Etude du langage de commande et de contrôle pour le réseau d'ordinateurs SOC (Système d'Ordinateurs Connectés)

Rita de Caluwe

THESE

présentée à

L'Université Scientifique et Médicale de Grenoble

pour obtenir

le grade de docteur de troisième cycle

«informatique»

par

RITA DE CALUWE

ETUDE DU LANGAGE DE COMMANDE ET DE CONTROLE POUR LE RESEAU D'ORDINATEURS SOC

(Système d'Ordinateurs Connectés)

Soutenue le 15 septembre 1973 devant la commission d'examen

MM.	Noël GASTINEL	Président
	Louis BOLLIET	Examinateur
	Michaël GRIFFITHS	Rapporteur
	Jean DU MASLE	Invité
Mme	Monique SOMIA	Invitée

UNIVERSITE SCIENTIFIQUE
ET MEDICALE DE GRENOBLE

LISTE DES PROFESSEURS

Président : Monsieur Michel SOUTIF
Vice-Président : Monsieur Gabriel CAU

PROFESSEURS TITULAIRES

MM.	ANGLES D'AURIAC Paul	Mécanique des fluides
	ARNAUD Georges	Clinique des maladies infectieuses
	ARNAUD Paul	Chimie
	AUBERT Guy	Physique
	AYANT Yves	Physique approfondie
Mme	BARBIER Marie-Jeanne	Electrochimie
MM.	BARBIER Jean-Claude	Physique expérimentale
	BARBIER Reynold	Géologie appliquée
	BARJON Robert	Physique nucléaire
	BARNOUD Fernand	Biosynthèse de la cellulose
	BARRA Jean-René	Statistiques
	BARRIE Joseph	Clinique chirurgicale
	BENOIT Jean	Radioélectricité
	BERNARD Alain	Mathématiques Pures
	BESSON Jean	Electrochimie
	BEZES Henri	Chirurgie générale
	BLAMBERT Maurice	Mathématiques Pures
	BOLLIET Louis	Informatique (IUT B)
	BONNET Georges	Electrotechnique
	BONNET Jean-Louis	Clinique ophtalmologique
	BONNET-EYMARD Joseph	Pathologie médicale
	BONNIER Etienne	Electrochimie Electrométallurgie
	BOUCHERLE André	Chimie et Toxicologie
	BOUCHEZ Robert	Physique nucléaire
	BOUSSARD Jean-Claude	Mathématiques Appliquées
	BRAVARD Yves	Géographie
	BRISSONNEAU Pierre	Physique du solide
	BUYLE-BODIN Maurice	Electronique
	CABANAC Jean	Pathologie chirurgicale
	CABANEL Jean	Clinique rhumatologique et hydrologie
	CALAS François	Anatomie
	CARRAZ Gilbert	Biologie animale et pharmacodynamie
	CAU Gabriel	Médecine légale et Toxicologie
	CAUQUIS Georges	Chimie organique
	CHABAUTY Claude	Mathématiques Pures
	CHARACHON Robert	Oto-Rhino-Laryngologie
	CHATEAU Robert	Thérapeutique
	CHENE Marcel	Chimie papetière
	COEUR André	Pharmacie chimique
	CONTAMIN Robert	Clinique gynécologique
	COUDERC Pierre	Anatomie Pathologique
	CRAYA Antoine	Mécanique

Mme	DEBELMAS Anne-Marie	Matière médicale
MM.	DEBELMAS Jacques	Géologie générale
	DEGRANGE Charles	Zoologie
	DESRE Pierre	Métallurgie
	DESSAUX Georges	Physiologie animale
	DODU Jacques	Mécanique appliquée
	DOLIQUE Jean-Michel	Physique des plasmas
	DREYFUS Bernard	Thermodynamique
	DUCROS Pierre	Cristallographie
	DUGOIS Pierre	Clinique de Dermatologie et Syphiligraphie
	FAU René	Clinique neuro-psychiatrique
	FELICI Noël	Electrostatique
	GAGNAIRE Didier	Chimie physique
	GALLISSOT François	Mathématiques Pures
	GALVANI Octave	Mathématiques Pures
	GASTINEL Noël	Analyse numérique
	GEINDRE Michel	Electroradiologie
	GERBER Robert	Mathématiques Pures
	GIRAUD Pierre	Géologie
	KLEIN Joseph	Mathématiques Pures
Mme	KOFLER Lucie	Botanique et Physilogie végétale
MM.	KOSZUL Jean-Louis	Mathématiques Pures
	KRAVTCHENKO Julien	Mécanique
	KUNTZMANN Jean	Mathématiques appliquées
	LACAZE Albert	Thermodynamique
	LACHARME Jean	Biologie végétale
	LAJZEROWICZ Joseph	Physique
	LATREILLE René	Chirurgie générale
	LATURAZE Jean	Biochimie pharmaceutique
	LAURENT Pierre-Jean	Mathématiques appliquées
	LEDRU Jean	Clinique médicale B
	LLIBOUTRY Louis	Géophysique
	LOUP Jean	Géographie
Mle	LUTZ Elisabeth	Mathématiques Pures
MM.	MALGRANGE Bernard	Mathématiques Pures
	MALINAS Yves	Clinique obstétricale
	MARTIN-NOEL Pierre	Seméiologie médicale
	MASSEPORT Jean	Géographie
	MAZARE Yves	Clinique médicale A
	MICHEL Robert	Minéralogie et Pétrographie
	MOURIQUAND Claude	Histologie
	MOUSSA André	Chimie nucléaire
	NEEL Louis	Physique du solide
	OZENDA Paul	Botanique
	PAUTHENET René	Electrotechnique
	PAYAN Jean-Jacques	Mathématiques Pures
	PEBAY-PEYROULA Jean-Claude	Physique
	PERRET René	Servomécanismes
	PILLET Emile	Physique industrielle
	RASSAT André	Chimie systématique
	RENARD Michel	Thermodynamique
	REULOS René	Physique industrielle
	RINALDI Renaud	Physique
	ROGET Jean	Clinique de pédiatrie et de puériculture
	SANTON Lucien	Mécanique
	SEIGNEURIN Raymond	Microbiologie et Hygiène
	SENGEL Philippe	Zoologie
	SILBERT Robert	Mécanique des fluides
	SOUTIF Michel	Physique générale

MM.	TANCHE Maurice	Physiologie
	TRAYNARD Philippe	Chimie générale
	VAILLAND François	Zoologie
	VALENTIN Jacques	Physique nucléaire
	VAUQUOIS Bernard	Calcul électronique
Mme	VERAIN Alice	Pharmacie galénique
M.	VERAIN André	Physique
Mme	VEYRET Germaine	Géographie
MM.	VEYRET Paul	Géographie
	VIGNAIS Pierre	Biochimie médicale
	YOCCOZ Jean	Physique nucléaire théorique

PROFESSEURS ASSOCIES

MM.	BULLEMER Bernhard	Physique
	HANO JUN-ICHI	Mathématiques Pures
	STEPHENS Michaël	Mathématiques appliquées

PROFESSEURS SANS CHAIRE

MM.	BEAUDOING André	Pédiatrie
Mme	BERTRANDIAS Françoise	Mathématiques Pures
MM.	BERTRANDIAS Jean-Paul	Mathématiques appliquées
	BIAREZ Jean-Pierre	Mécanique
	BONNETAIN Lucien	Chimie minérale
Mme	BONNIER Jane	Chimie générale
MM.	CARLIER Georges	Biologie végétale
	COHEN Joseph	Electrotechnique
	COUMES André	Radioélectricité
	DEPASSEL Roger	Mécanique des fluides
	DEPORTES Charles	Chimie minérale
	GAUTHIER Yves	Sciences biologiques
	GAVEND Michel	Pharmacologie
	GERMAIN Jean-Pierre	Mécanique
	GIDON Paul	Géologie et Minéralogie
	GLENAT René	Chimie organique
	HACQUES Gérard	Calcul numérique
	JANIN Bernard	Géographie
Mme	KAHANE Josette	Physique
MM.	MULLER Jean-Michel	Thérapeutique
	PERRIAUX Jean-Jacques	Géologie et Minéralogie
	POULOUJADOFF Michel	Electrotechnique
	REBECQ Jacques	Biologie (CUS)
	REVOL Michel	Urologie
	REYMOND Jean-Charles	Chirurgie générale
	ROBERT André	Chimie papetière
	DE ROUGEMONT Jacques	Neurochirurgie
	SARRAZIN Roger	Anatomie et chirurgie
	SARROT-REYNAULD Jean	Géologie
	SIBILLE Robert	Construction mécanique
	SIROT Louis	Chirurgie générale
Mme	SOUTIF Jeanne	Physique générale

MAITRES DE CONFERENCES ET MAITRES DE CONFERENCES AGREGES

Mle	AGNIUS-DELORD Claudine	Physique pharmaceutique
	ALARY Josette	Chimie analytique
MM.	AMBLARD Pierre	Dermatologie
	AMBROISE-THOMAS Pierre	Parasitologie
	ARMAND Yves	Chimie
	BEGUIN Claude	Chimie organique
	BELORIZKY Elie	Physique
	BENZAKEN Claude	Mathématiques appliquées
	BILLET Jean	Géographie
	BLIMAN Samuel	Electronique (EIE)
	BLOCH Daniel	Electrotechnique
Mme	BOUCHE Liane	Mathématiques (CUS)
MM.	BOUCHET Yves	Anatomie
	BOUVARD Maurice	Mécanique des fluides
	BRODEAU François	Mathématiques (IUT B)
	BRUGEL Lucien	Energétique
	BUISSON Roger	Physique
	BUTEL Jean	Orthopédie
	CHAMBAZ Edmond	Biochimie médicale
	CHAMPETIER Jean	Anatomie et organogénèse
	CHIAVERINA Jean	Biologie appliquée (EFP)
	CHIBON Pierre	Biologie animale
	COHEN-ADDAD Jean-Pierre	Spectrométrie physique
	COLOMB Maurice	Biochimie médicale
	CONTE René	Physique
	COULOMB Max	Radiologie
	CROUZET Guy	Radiologie
	DURAND Francis	Métallurgie
	DUSSAUD René	Mathématiques (CUS)
Mme	ETERRADOSSI Jacqueline	Physiologie
MM.	FAURE Jacques	Médecine légale
	GENSAC Pierre	Botanique
	GIDON Maurice	Géologie
	GRIFFITHS Michaël	Mathématiques appliquées
	GROULADE Joseph	Biochimie médicale
	HOLLARD Daniel	Hématologie
	HUGONOT Robert	Hygiène et Médecine préventive
	IDELMAN Simon	Physiologie animale
	IVANES Marcel	Electricité
	JALBERT Pierre	Histologie
	JOLY Jean-René	Mathématiques Pures
	JOUBERT Jean-Claude	Physique du solide
	JULLIEN Pierre	Mathématiques Pures
	KAHANE André	Physique générale
	KUHN Gérard	Physique
	LACOUME Jean-Louis	Physique
Mme	LAJZEROWICZ Jeannine	Physique
MM.	LANCIA Roland	Physique atomique
	LE JUNTER Noël	Electronique
	LEROY Philippe	Mathématiques
	LOISEAUX Jean-Marie	Physique nucléaire
	LONGEQUEUE Jean-Pierre	Physique nucléaire
	LUU DUC Cuong	Chimie organique
	MACHE Régis	Physiologie végétale
	MAGNIN Robert	Hygiène et Médecine préventive
	MARECHAL Jean	Mécanique
	MARTIN-BOUYER Michel	Chimie (CUS)

MM.	MAYNARD Roger	Physique du solide
	MICHOULIER Jean	Physique (IUT A)
	MICOUD Max	Maladies infectieuses
	MOREAU René	Hydraulique (INP)
	NEGRE Robert	Mécanique
	PARAMELLE Bernard	Pneumologie
	PECCOUD François	Analyse (IUT B)
	PEFFEN René	Métallurgie
	PELMONT Jean	Physiologie animale
	PERRET Jean	Neurologie
	PERRIN Louis	Pathologie expérimentale
	PFISTER Jean-Claude	Physique du solide
	PHELIP Xavier	Rhumatologie
Mlle	RIERY Yvette	Biologie animale
MM.	RACHAIL Michel	Médecine interne
	RACINET Claude	Gynécologie et obstétrique
	RENAUD Maurice	Chimie
	RICHARD Lucien	Botanique
Mme	RINAUDO Marquerite	Chimie macromoléculaire
MM.	ROMIER Guy	Mathématiques (IUT B)
	SHOM Jean-Claude	Chimie générale
	STIEGLITZ Paul	Anesthésiologie
	STOEBNER Pierre	Anatomie pathologique
	VAN CUTSEM Bernard	Mathématiques appliquées
	VEILLON Gérard	Mathématiques appliquées (INP)
	VIALON Pierre	Géologie
	VOOG Robert	Médecine interne
	VROUSSOS Constantin	Radiologie
	ZADWORNY François	Electronique

MAITRES DE CONFERENCES ASSOCIES

MM.	BOUDOURIS Georges	Radioélectricité
	CHEEKE John	Thermodynamique
	GOLDSCHMIDT Hubert	Mathématiques
	SIDNEY STUARD	Mathématiques Pures
	YACOUD Mahmoud	Médecine légale

CHARGES DE FONCTIONS DE MAITRES DE CONFERENCES

Mme	BERIEL Hélène	Physilogie
Mme	RENAUDET Jacqueline	Microbiologie

Fait le 30 mai 1972.

Nous tenons à remercier sincèrement

Monsieur le Professeur N.GASTINEL, pour nous avoir fait l'honneur de vouloir présider le jury de notre thèse,

Monsieur le Professeur L.BOLLIET, qui a acepté d'en faire partie,

Monsieur M.GRIFFITHS, Maître de Conférences, pour l'intérêt qu'il a porté à notre travail,

Monsieur J.du MASLE, Ingénieur au CNRS, qui nous a inlassablement guidés et encouragés dans notre recherche,

Madame M.SOMIA, Responsable du projet SOC au Développement Scientifique de la Compagnie IBM France, pour son infaillible disponibilité dans la résolution des problèmes de coordination au sein de l'équipe SOC.

Egalement, nous voulons remercier chaleureusement

Toutes celles et tous ceux à l'IMAG qui nous sont toujours gentiment venus en aide avec beaucoup de bonne humeur, dans nos petits problèmes techniques quotidiens,

Ainsi que l'équipe qui s'est occupée avec beaucoup de soin de la réalisation matérielle de cette thèse.

TABLE DES MATIERES

Page

INTRODUCTION

SOC ("Système d'Ordinateurs Connectés") est un projet réseau expérimental, réalisé en collaboration, par l'Institut des Mathématiques Appliquées de Grenoble, l'Ecole Nationale Supérieure des Mines de Paris à Fontainebleau, le Commissariat à l'Energie Atomique à Saclay, le Centre Interdiscipline Régional de Calcul Electronique à Orsay, le Centre Scientifique IBM de Grenoble à La Tronche et le Centre Scientifique IBM de Paris.

Le but de la réalisation est de permettre aux utilisateurs d'accéder à toutes les ressources disponibles sur l'un quelconque des ordinateurs du réseau. Ces ressources peuvent être des programmes, des données, des unités périphériques, de la mémoire centrale, des performances particulières par exemple. Dans un réseau, l'utilisateur ne s'adresse pas à un ordinateur implicite, mais il s'adresse au réseau. Le langage de contrôle et de commande pour un réseau diffère donc du langage de commande d'un ordinateur par le fait qu'il y a la notion d'ordinateur en plus à exprimer.

Le langage de contrôle et de commande pour le réseau SOC mis à la disposition de l'utilisateur est le "langage externe" (dénoté LE). C'est un langage de haut niveau. Il est traduit par le compilateur en des séquences d'ordres en "langage interne" (LI), destinés à être interprétés sur l'ordinateur à qui ils sont adressés. A ce niveau il sera tenu compte des caractéristiques de l'ordinateur concerné et de son mode d'exploitation. Actuellement, dans le réseau SOC, les ordinateurs connectés sont tous des IBM des mêmes séries (360 et 370), mais dans sa conception le réseau est hétérogène.

L'étude du LE a été commencée vers la fin de 1970 et les premiers résultats ont fait l'objet d'un rapport de D.E.A. [DECA]. Le LE et son mode d'implémentation présentés ici sont le résultat de la continuation de ce travail. Par rapport à la première définition, certaines modifications ont été in-

troduites, et la présente étude nous a appris qu'il vaut mieux parler de LE.1 que de LE, LE.2 étant déjà en perspective.

Dans ce qui suit, nous discuterons successivement de l'étude préliminaire qui a mené à la conception et la définition du LE.1, des éléments du LE.1, des exemples de jobs-réseau; nous ferons le point sur les limitations actuelles et les extensions à prévoir; nous décrirons l'implémentation, de la grammaire du LE.1 à la compilation des jobs-réseau; finalement nous donnerons nos réflexions sur le langage ALGOLW, comme langage d'écriture d'un prototype du compilateur.

1. ETUDE PRELIMINAIRE

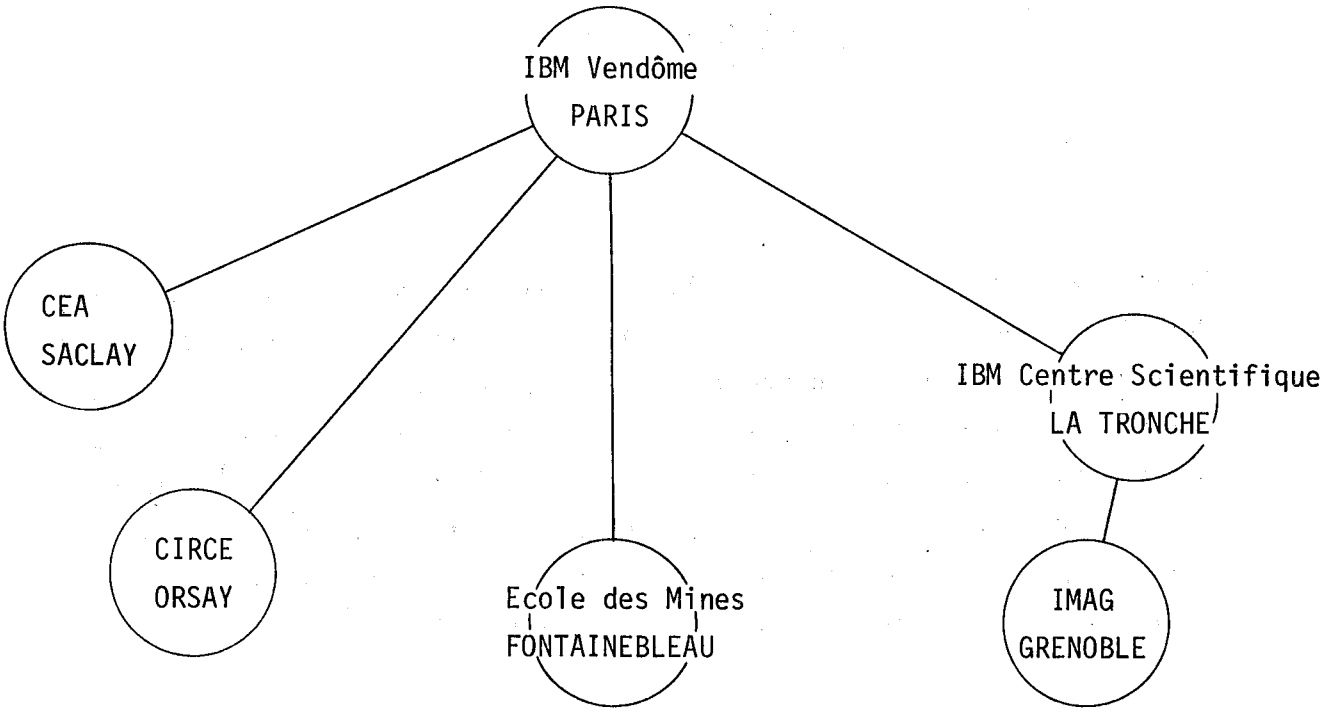
Le LE.1 est un langage très proche des réalités physiques qu'il décrit. Derrière chaque variable du langage il y a une entité physique bien précise, et à chaque instruction correspond une fonction qui traite de ces entités. Les entités, les fonctions, sont les mêmes que l'on rencontre chez les ordinateurs de diverses constructions, dans les divers systèmes d'exploitation sous lesquels ils opèrent. On les rencontre tout naturellement dans les études de langages de contrôle et de commande qui ont été faites avant la nôtre. Nous avons étudié ces aspects en vue de définir LE.1.

1.1. Architecture générale du réseau SOC

Le réseau SOC est un réseau général, qui dans sa conception regroupe plusieurs ordinateurs différents, mais qui, fortuitement, dans sa configuration expérimentale actuelle n'est constitué que d'ordinateurs IBM compatibles des séries 360 et 370.

Il a été décidé de réunir les ordinateurs par des équipements de télécommunication (unités de contrôle, modems et lignes de transmission) dans ce qui est appelé une configuration distribuée, c'est-à-dire que chaque ordinateur est relié à un nombre quelconque d'autres ordinateurs du réseau, de sorte qu'il existe toujours au moins une liaison entre deux ordinateurs quelconques du réseau.

La configuration expérimentale se présente schématiquement comme suit :



Le fait que les liaisons entre chaque paire d'ordinateurs sont uniques, tient à des raisons financières et n'enlève rien à la généralité du concept. L'ordinateur à IBM-Vendôme n'a aucun rôle particulier, mais simplement, il assurera un plus grand trafic de messages que les autres.
Le réseau est géré par un système-réseau.
L'utilisateur se servira des ordinateurs du réseau par l'intermédiaire du langage externe de contrôle et de commande (LE.1). Le travail-réseau qu'il constitue et soumet au système-réseau à partir d'un ordinateur quelconque, est pris en charge par un processus lecteur-compilateur qui le traduit en autant de séquences en langage interne (LI) qu'il y a d'ordinateurs concernés par ce travail-réseau. Une fois toutes compilées, ces séquences sont transportées par le processus lecteur interne à chacun des ordinateurs à qui elles sont destinées. Sur chaque ordinateur, un interpréteur du langage interne s'en charge maintenant, les interprète et les transmet au système d'exploitation local. Le processus initiateur-terminateur les démarre et en récupère les sorties.

L'utilisateur récup rera les résultats des travaux ainsi que les résultats des commandes au niveau du réseau (liste, éventuellement avec messages d'erreurs par exemple). Le processus de sortie se charge de produire ces derniers.
Les processus au niveau de l'utilisateur (lecteur-compilateur, lecteur interne, interpréteur et writer) sont démarrés dans le système-réseau par le processus initiateur [SOMI].

1.2. Entités physiques à manipuler.

D'une façon ou d'une autre, ce qui nous intéresse est de manipuler des données (éventuellement des programmes,...) sous forme de fichiers.
Un fichier de données, "data set" (auquel nous associerons dans LE.1 la variable de type DSN), se trouve sur une bande, sur un disque,...en termes généraux sur un volume, "volume" (d'où la variable de type VOLUME), qui selon sa nature sera ou est monté sur une unité d'un certain type, le "unit" (variable UNIT) qui est périphérique à l'unité centrale d'un ordinateur, "central processing unit" (variable CPU). Un fichier de données peut être de telle envergure qu'il nécessite plusieurs volumes pour le brocher et ainsi pour le définir complètement, il faut disposer de la liste de ces volumes, "volume list" (variable VOLLIST). D'un autre côté, un fichier important peut être constitué de plusieurs parties accessibles séparément, les membres, "member" (variable MBR), ou manipulées regroupées par plusieurs, décrites par une liste de membres, "member list" (variable LMBR).
Pour pouvoir les distinguer, ces fichiers portent des noms, qui représentent une valeur à manipuler. Les fichiers de données se trouvent à un certain endroit sur les volumes (la façon de définir cet endroit dépend de la nature du volume). Soit on connaît soi-même les coordonnées physiques de cet endroit et on les précise à chaque fois que l'on veut atteindre le fichier, soit ils ont été confiés une fois pour toutes au catalogue du système d'exploitation, auquel on peut s'adresser pour repérer le fichier ultérieurement.

Physiquement, les fichiers se composent d'enregistrements d'une certaine longueur (variable LRECL), qui sont accessibles séparément ou d'une façon ou d'une autre en blocs (variable RECFM) d'un certain nombre d'enregistrements, donc d'une longueur multiple de la longueur de leurs enregistrements (variable BLKSIZE).

Les enregistrements des fichiers sont accessibles d'une certaine façon, par exemple uniquement séquentiellement s'il s'agit de bandes, d'une façon éventuellement aléatoire s'il s'agit de disques, de tambours (variable DSORG).
La taille d'un fichier doit pouvoir être exprimée (variable SPACE): emplacement demandé, alloué, utilisé tout ou en partie. La demande peut être ferme ("primary space"), ou optionnnellement avec des allocations ultérieures s'il y a besoin ("secondary space") ou encore avec définition d'un nombre d'entrées dans le dictionnaire qui contient le nom des membres des fichiers partitionnés ("directory").

Certains fichiers ont une signification particulière dans le LE.I, étant une capacité de stockage propre au système-réseau : les fichiers "spool"(Simultaneous Peripheral Operation On Line) (variable DIS). Leur gestion est inspirée sur celle des files d'attente du système ASP.

1.3. Fonctions à exprimer.

On revient toujours au problème de pouvoir désigner, créer, supprimer et manipuler des fichiers (qui contiennent des données, des programmes etc.) sur les ordinateurs faisant partie du réseau et exprimer des ordres de transfert dans le réseau.
Les manipulations les plus courantes font l'objet des fonctions (communément appelées "utilitaires" dans les systèmes) de copie, de concaténation, d'impression, de maintenance du catalogue. Au fait, la plupart de ces fonctions se ramènent à des copies d'une façon ou d'une autre, si on raisonne sans tenir compte des types d'unité qui y interviennent. Dans le système MTS [FLAN], cette idée a d'ailleurs été valorisée.

Ces fonctions seront directement invoquables par des instructions exécutables LE.1 appropriées.

Pour laisser la porte ouverte à toutes les fonctions qui peuvent se présenter dans un système d'exploitation particulier et qui ne peuvent pas avoir d'équivalent général ou qui ne présentent pas assez d'intérêt pour faire l'objet d'une fonction générale, on doit avoir la possibilité de les définir dans le travail-réseau d'une façon indépendante des instructions LE.1, en code, mais de les faire exécuter et de récupérer leurs résultats dans l'environnement-réseau. En particulier, la soumission de travaux locaux pourra être faite de cette façon.

Toutes ces fonctions revêtent en plus une nouvelle dimension, puisque dans le réseau l'exécution passera aussi bien en local que sur un autre ordinateur du réseau. Des problèmes de transfert et de synchronisation y sont afférents.

1.4. Etude comparative des langages de contrôle et de commande de quelques réseaux et systèmes d'exploitation existants.

Outre les fonctions reprises dans les langages, l'allure du langage nous a intéressés. En général elle est, bien sûr, une façon adéquate d'exprimer les fonctions, mais très souvent, ni souple, ni commode.

En matière de réseaux, mis en oeuvre avant SOC, CYBERNET [SHAD][GOTT] et ARPA [ELI1][ELI2][CARR][OSUL][CROC] ont été les plus importants. Des langages de contrôle et de commande ont été conçus pour eux.

Dans le réseau CYBERNET, le langage de contrôle et de commande SHADE (implémenté sous le système-réseau SHADOW, qui tourne sur un réseau d'ordinateurs CDC de type 3300 et 3500 essentiellement, en coexistance avec l'operating system MASTER) on trouve exactement les fonctions utilitaires citées plus haut, avec en plus la possibilité de consulter et de modifier les fichiers de données dans le réseau.

La forme du langage est une séquence d'ordres avec des paramètres mot-clé et des paramètres positionnels, séparés par des virgules: ses valeurs par défaut peuvent être affectées à certaines variables.
Remarquons au passage que dans le système SHADOW, la protection des fichiers de données est très étudiée, ce qui se reflète dans les paramètres du langage SHADE.

En ce qui concerne le réseau hétérogène ARPA, nous n'avons trouvé que peu de documentation disponible sur TELNET, qui est beaucoup plus un protocole de transmission qu'un langage. En tout cas la forme des commandes y est très élémentaire.

Comme projet-réseau, il nous faut encore citer un projet plus récent et de moindre envergure que les précédents, pour lequel un langage de commande de style tout à fait différent a été conçu, qui nous a intéressé plus particulièrement, parce que dans sa conception il est proche de LE.1 du projet SOC.
Il s'agit d'un projet de l'Université d'Illinois [GRAH] qui n'interconnecte que deux ordinateurs, un ILLIAC IV et un BURROUGHS. Pour ce système de réseau très réduit et spécifique, le langage de contrôle ICL ("Illiac Control Language") a été conçu de type Algol 60.

En regardant du côté des langages de contrôle et de commande des systèmes conversationnels CP/CMS[CP67], TSS[TSSC] et MTS[FLAN] sur IBM, INTERCOM2[ICOM] sur CDC et des langages de contrôle des systèmes OS/360 et autres rencontrés au passage, on retrouve toujours les mêmes fonctions exprimées avec plus ou moins de possibilités permises, mais en gros toujours dans une forme élémentaire semblable, comme une séquence d'ordres, écrits avec des paramètres mot-clé et des paramètres positionnels. La forme externe est de type assembleur ou macro-assembleur. Nous mentionnons ici tout de même une possibilité intéressante dans GEORGE 3 sur ICL 1900 [ICL] qui nous a frappés par son originalité vis à vis des autres langages de contrôle desquels nous avons pû prendre connaissance et qui va dans le sens des langages évolués qui nous intéressent. Il s'agit de la possibilité de décider du choix des ordres suivants à exécuter si les précédents se terminent anormalement.
On écrit :

IF FAIL THEN 1)

où plus loin 1) est défini comme

1) ..... (action)

Pour avoir une idée de l'allure des commandes dans ces langages, prenons l'exemple de la création d'un fichier. On écrit :

En SHADE :

EST,P,TESTFILE,****,****,80,500,10

(Commande ESTABLISH d'un fichier à accès PUBLIC de nom TESTFILE, sans spécification des codes d'accès -l'omission est indiquée par les *- de 500 enregistrements à 80 caractères chacun, numérotés de 10 en 10 à partir de 10).

En CP/CMS :

Création implicite, lors d'une autre commande par exemple

COMBINE TESTFILE TYPE P1 REFFILE TYPE P1

(Le nouveau fichier TESTFILE a les spécifications de celui qu'on y copie , REFFILE dans l'exemple)

En MTS :

$CREATE -TESTFILE

(Création d'un fichier temporaire sur disque. Le fichier ainsi créé, à un emplacement choisi par le système avec des caractéristiques standard, est temporaire du fait que son nom débute par le signe -. Par des paramètres falcutatifs, en général non nécessaires, on peut lui fixer des attributs ou une localisation précise).

En INTERCOM2 :

En intéractif dans le mode "command" en répondant successivement

SETUP.

SYSTEM/GENERAL (ou autre)

NEW/TESTFILE (ou autre)

En LE.1 l'écriture suivante :

```
CPU  A:=IMAG.67/2314/IMAG80/FELIX;
DSCB SPECIF:=((CYL,(2,1),(FB,80,800),PS);
CREATE  FELIX        (SPECIF);
```

a pour effet :

1- De déclarer en le localisant un fichier de nom FELIX.
2- De définir dans la déclaration DSCB :
   . l'espace à allouer (2 cylindres d'espace primaire et éventuellement 15 fois un cylindre d'espace additionnel)
   . le type des enregistrements - image de cartes de 80 caractères bloqués par 10
   . l'organisation du fichier séquentiel.
   La déclaration de variable DSCB peut être omise en partie si l'on désire utiliser des valeurs par défaut.
3- Création effective par l'ordre CREATE.

1.5. Idées générales retenues pour le langage externe

Le LE.1 se voudra tout d'abord un outil facile pour l'utilisateur. Dans les langages étudiés, les règles d'écriture de style macro-assembleur paraissent trop contraignantes, pour les appliquer convenablement. Le LE.1 sera défini comme un langage de haut niveau qui alliera la souplesse de sa forme à la puissance des autres langages de contrôle et de commande. Ces idées n'ont pas toutes été entièrement développées dans la première version du langage LE.

L'étude de la structure de divers langages de programmation (qui seront cités ultérieurement) et celle des systèmes étudiés dans le paragraphe 1.4 sont à l'origine des concepts généraux retenus pour LE.1, dont les plus importants suivent.Des types de variables qui correspondent aux réalités physiques, seront définis, les fonctions exprimées d'une façon simple. Des facilités pour concaténer à moindres frais des programmes en LE.1 seront mises à la

disposition de l'utilisateur (noms, synonymes pour une variable). La définition des programmes en code, intégrés à un programme-réseau, sera bien étudiée. Les ordres de contrôle à la charge de l'utilisateur LE.1 seront aussi réduits que possible. Les programmes écrits en LE.1 par l'utilisateur seront compilés en un code interprétatif (le langage interne LI).

En outre, un travail-réseau s'exécutera pas à pas, si bien que, en cas d'erreur dans un pas, les pas (correctement) exécutés avant celui qui conduit en erreur, n'auront pas à être repris.

2. LE LANGAGE EXTERNE DE CONTROLE ET DE COMMANDE

Dans ce qui suit nous décrivons l'allure générale du langage externe et nous en donnons la définition. Des exemples illustrent son utilisation. Nous faisons le point sur LE.1 en vue de la définition de LE.2.

2.1. Concepts généraux.

De par sa forme, le LE.1 est un langage de haut niveau, qui comprend, d'une part des ordres de contrôle, destinés à établir la connexion de l'utilisateur à SOC et d'autre part des instructions, déclaratives et exécutables.
On appellera par la suite travail-réseau, toute suite convenable d'un ordre de connexion (NETIN), d'une séquence d'instructions LE.1 et d'un ordre de déconnexion (NETOUT). Un travail-réseau dépouillé de ses ordres de contrôle sera appelé un programme-réseau.
Les ordres et les instructions s'écrivent à l'aide de mots-clé en tête, exception faite pour l'instruction arithmétique. Dès le début, l'option a été prise, que toute variable utilisée dans une instruction exécutable doit avoir été préalablement déclarée dans une instruction déclarative. Toutefois, pour autant que l'on respecte cette dernière règle, instructions déclaratives et instructions exécutables peuvent être mélangées dans un programme LE.1.
Par exemple, si l'on a besoin de faire monter le disque 20MVT3 à l'IMAG, on déclarera une variable, soit V, de type VOLUME et de valeur 20MVT3. La forme de la déclaration permet d'exprimer également que le type de l'unité correspondant à ce disque est 2314 et que l'ordinateur est celui de l'IMAG. Après cette déclaration nécessaire, on pourra écrire l'instruction de montage du disque.

Au moment de la déclaration d'une variable on lui affecte une valeur, soit implicitement, soit explicitement. Cette obligation n'existe cependant pas pour les variables d'un type sur lequel des opérations arithmétiques sont définies, parce qu'il est possible de leur affecter une valeur ultérieurement, à la suite de l'évaluation d'une expression arithmétique.

Les variables peuvent être désignées par plusieurs noms synonymes, comme le permet aussi PL360 [WIR2]. Une notion importante est celle de travail en langage de commande d'une machine quelconque du réseau inséré dans le flot d'entrée. C'est la façon d'introduire des travaux locaux complets autres que des fonctions utilitaires, dans le flot d'entrée d'un travail-réseau, avec pour but, de les envoyer à l'ordinateur de destination, de les y faire exécuter en batch, à partir du programme-réseau et de recueillir les résultats de cette exécution dans le programme-réseau. Cette notion dans le LE.1 a deux aspects : D'un côté, par son intermédiaire, on arrive à pallier toutes les lacunes du LE.1, de l'autre la déclaration d'un travail inséré dans le flot d'entrée suivi d'une instruction d'exécution est un programme-réseau complet et ne diffère donc guère du travail en traitement par lots lui-même.

La notion de procédure en code dans un langage de haut niveau n'est pas nouvelle. On la trouve dans de nombreux compilateurs Algol 60 (sur IBM 7044-BOUSSARD [BOUS], KDF9, RANDELL et RUSSELL[RAND]). Entre l'espace des noms des travaux insérés et celui des variables du LE.1 aucune relation ne sera établie, si bien que si l'on utilise un même fichier dans les deux environnements, il faut faire deux fois des déclarations rigoureusement conformes.

Aux erreurs à l'exécution on associe un niveau de sévérité d'erreur et dans le programme en LE.1 un niveau permis est défini. Toute erreur qui a un niveau supérieur au niveau permis entraine la terminaison immédiate du travail. Le niveau permis à défaut de déclaration explicite est le niveau zéro, c'est-à-dire, aucune erreur n'est tolérée.

2.2. Les éléments du langage externe LE.1.

Nous définissons les symboles de base, mentionnons les différents types de variables, discutons les valeurs que les différents types de variables peuvent avoir et les noms qu'on peut leur donner et faisons le tour des instructions et ordres de contrôle du langage. Néanmoins, la sémantique des instructions et des ordres sera examinée en détail dans le chapitre suivant, à l'occasion de la discussion du compilateur.

Dans ce qui suit on remarquera que les mots clés (dont nous avons déjà vu NETIN et NETOUT) sont en anglais bien qu'il s'agisse d'un projet français. Ce choix, qui est d'ailleurs celui des constructeurs français, a pour but de rendre le langage LE.1 aussi familier que possible aux utilisateurs de systèmes d'exploitation utilisés sur le réseau (OS/360 option MVT). La suite des lettres qui constituent le symbole de base NETIN est un artifice de représentation externe du symbole netin (en reprenant les notations usuelles d'Algol 60).

2.2.1. Les symboles de base.

Ce sont :

Les chiffres 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Les lettres et le point
A | B | C | D | E | F | G | H | I | J | K | L | M | N |
O | P | Q | R | S | T | U | V | W | X | Y | Z | .

Les autres
+ | − | * | / | ; | , | ' | (|) | :=

Les mots-clé

AT | BY | ON | TO | NETIN | NETOUT | CPU | UNIT | VOLUME |
VOLLIST | DSN | MBR | LMBR | SPACE | ALLOC | CYL | TRK |
RECFM | F | FB | V | VB | U | VS | VBS | LRECL | BLKSIZE | DSORG |
PS | DA | IS | PO | REC | DSCB | INP | LIST | MSG | COMMENT |
QUALIFY | DEFAULT | LEVEL | DUMP | MOUNT | DISMOUNT |
RING | NORING | SL | NL | BLP | EMPTY | DELETE | CATALG |
UNCTLG | ADD | COPY | CREATE | RUN

On trouvera en annexe la traduction des mots-clé.

2.2.2. Les entiers, les valeurs-chaîne, les identificateurs, les chaînes de caractères.

Les entiers sans signe

Un entier est une suite de chiffres.

Une valeur-chaîne

est une suite d'éléments, chaque élément étant soit une lettre, soit un chiffre, soit un point.

(Par exemple un nom de fichier en OS : SYS1.LINKLIB, un type d'unité 2314, SYSDA etc...)

Les identificateurs

Un identificateur est une suite d'éléments, chaque élément étant soit une une lettre, soit un chiffre, soit un point et qui commence par une lettre.

Les textes ou chaînes de caractères

Un texte est une suite d'éléments, chaque élément étant soit un chiffre, une lettre, un point ou un autre symbole de base précédé et suivi d'une apostrophe simple (').

Les apostrophes qui figurent dans la suite d'éléments doivent être doublées.

2.2.3. Les variables du langage externe.

Puisque dans le langage externe toutes les fonctions portent sur des fichiers (ou membres de fichiers), les variables qui y sont définissables servent, soit à décrire la localisation des fichiers, soit à décrire les spécifications des fichiers.

2.2.3.1. *Types.*

Les variables de localisation sont de type unité centrale (CPU), unité périphérique (UNIT), volume(VOLUME), liste de volumes (VOLLIST), nom de fichier (DSN), membre (MBR) et liste de membres (LMBR). Les variables de spécifications sont du type format d'enregistrement (RECFM), longueur logique d'enregistrement (LRECL), taille de bloc(BLKSIZE), espace(SPACE), définition d'enregistrement (REC), organisation de fichier(DSORG), descripteur de fichier (DSCB).

Un fichier même correspond à une variable de type nom de fichier (DSN), un membre d'un fichier partitionné à une variable de type membre (MBR),une liste de membres d'un fichier partitionné à une variable de type liste de membre (LMBR).

Pour localiser complètement un fichier il faut connaître le ou les volumes, l'unité et l'ordinateur associés, c'est-à-dire les variables de types VOLUME ou VOLLIST, UNIT, CPU.
Pour spécifier complètement un fichier, il faut les renseignements contenus dans son descripteur physique. Ils sont exprimés par des variables de type DSCB ou SPACE, REC, DSORG ou SPACE, RECFM, LRECL, BLKSIZE, DSORG.
(A un travail inséré dans le flot d'entrée, et qui est stocké dans un fichier spécial du système d'entrées/sorties simultanées (SPOOL) est associée une variable de type descripteur interne de fichier d'entrée-sortie (DIS) . Ce type de variable n'est pas accessible explicitement au niveau LE.)

2.2.3.2. *Valeurs.*

Les valeurs que peuvent prendre les variables du langage externe sont soit des valeurs-chaîne, soit des valeurs d'entiers, soit des valeurs de référence à une autre variable, chacune selon sa définition.

Examinons à l'aide d'exemples les valeurs qui peuvent être affectées aux variables du langage externe.

2.2.3.2.1. Variables de localisation.

CPU (Unité centrale de traitement)

A chaque ordinateur faisant partie du réseau correspond une chaîne prédéfinie et réservée, valeur d'une variable de type CPU. Par exemple, la chaîne-valeur IMAG.67 représente en LE.1, si cette chaîne est rencontrée dans le contexte CPU, l'ordinateur de l'IMAG car il a été défini sous ce nom dans le compilateur. Les valeurs des variables de localisation, sauf CPU, sont à considérer comme des valeurs relatives. En effet, elles ne peuvent être considérées isolément et apparaissent toujours à un certain niveau de la structure hiérarchique suivante :

CPU UNIT VOLUME (VOLLIST) DSN (MBR, LMBR)

La valeur absolue d'une variable est l'ensemble de sa valeur relative et des valeurs relatives des variables à sa gauche dans cette séquence. Ainsi, par exemple, les deux variables DSN à valeur relative égale SYS.CMPLR ont une valeur absolue différente si elles font partie de l'une ou de l'autre des séquences de définition suivantes :

IMAG.67	2314	IMAG80	SYS.CMPLR
SAC.91	2314	SACOO1	SYS.CMPLR

(Spécifications égales implicites dans l'exemple).
Quand par la suite nous parlons de chaînes réservées nous sous-entendons réservées dans un certain contexte, comme indiqué plus loin par une table.

UNIT (Unité périphérique)

Type de variable à deux composantes :

- Une valeur-chaîne prédéfinie et réservée correspondant au type d'unité, par exemple : 2314, 2400.
  La liste des valeurs-chaîne est connue par le compilateur. Nous remarquons qu'il y a des chaînes qui s'écrivent de la même façon que les entiers. Il est possible de les distinguer par contexte.

- Une valeur référence à une variable de type CPU, correspondant à l'ordinateur auquel l'unité est attachée.

VOLUME (Volume)

Type de variable à deux composantes :

- Une valeur-chaîne, correspondant au nom physique du volume utilisé, par exemple : 111111 , 20MVT3 , IMAG80.
  Ces valeurs-chaîne ne sont pas prédéfinies. Elles sont reconnues dans le contexte et deviennent réservées par rapport à VOLUME à partir de leur déclaration.
- Une valeur-référence à une variable de type UNIT, correspondant à l'unité sur laquelle est ou sera monté le volume.

VOLLIST (Liste de volumes)

Type de variable à un nombre variable de composantes.
Une (ou plusieurs) valeur-référence à une variable de type VOLUME désignant le(s) volume(s) qui font partie de la liste.

DSN (Nom de fichier)

Type de variable à trois composantes :

- Une valeur-chaîne indiquant le nom physique du fichier.
  (Les valeurs-chaîne ne sont pas connues d'avance par le compilateur, mais deviennent réservées à partir de leur déclaration par rapport à DSN, MBR et LMBR).
  Cette chaîne peut être abrégée dans l'écriture d'un programme LE.1.
  Dans l'écriture d'un programme LE.1, deux cas sont donc à distinguer :
  - Le premier caractère de la chaîne est une lettre
    La valeur est considérée comme valeur à part entière.
    Par exemple : S5161.P0580.DECALUWE.CMPLR

- Le premier caractère de la chaîne est **un point.**
  La valeur est considérée comme écrite sous forme abrégée, à compléter par des renseignements définis antérieurement dans le programme réseau.
  Par exemple, si la valeur-chaîne

  .CMPLR

  a été précédée de l'instruction (qui sera décrite plus loin)

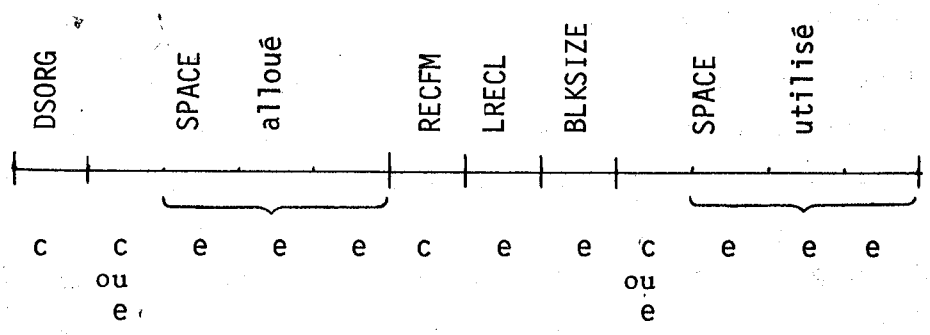
  QUALIFY <cpu name> BY S5161.PO58O.DECALUWE;

  la valeur chaîne interne sera :

  S5161.PO58O.DECALUWE.CMPLR

- Une valeur-référence à variable de type VOLLIST, qui désigne le volume sur lequel réside le fichier.
  Nous remarquons qu'un volume déclaré seul est considéré comme une liste de volumes à un élément et que de ce fait entre DSN et VOLUME, il y a toujours VOLLIST dans la représentation interne.

- Une partie qui contient toutes les spécifications qui se trouvent dans le descripteur physique du fichier, dans laquelle il y a donc des valeurs entier et chaîne, rangées comme le montre le schéma joint.



c ≡ valeur-chaîne

e ≡ entier

MBR (Membre de fichier partitionné)
Type de variable à deux composantes :

- Une valeur-chaîne, correspondant au nom du membre, par exemple : PHASEA .
  Ces chaînes deviennent réservées par rapport à DSN, MBR et LMBR dès leur déclaration.
- Une valeur-référence à une variable de type DSN indiquant le fichier duquel le membre fait partie.

LMBR (Liste de membres de fichiers partitionnés)
Type de variable à nombre variable de composantes :

Une(ou plusieurs) valeur(s) référence indiquant les membres qui font partie de la liste. Les membres dans la liste doivent tous faire partie d'un même fichier.

2.2.3.2.2. Variables de spécification.

Les variables de spécification ont pour fonction de permettre la création et l'utilisation des fichiers ou plus précisément d'ensembles de données. Un ensemble de données porte en lui les informations décrivant son contenu de telle sorte que tout programme qui sait lire cette partie descriptive est à même d'exploiter correctement le contenu. Ces informations sont regroupées dans la variable DSCB-bloc de contrôle d'ensemble de données. Ayant choisi un volume (support d'information) et un nom pour un ensemble de données, la première opération a réaliser consiste à lui allouer de l'espace sur le volume (opération d'allocation). Il est usuel d'allouer l'espace en plusieurs tranches :

- l'espace primaire
- les espaces additionnels
- éventuellement l'espace nécessaire à l'annuaire si le fichier est de type bibliothèque.

Les détails d'une demande d'espace sont indiqués par la variable SPACE. L'espace alloué à un ensemble de données n'est pas toujours tout utilisé, d'où la distinction qu'il y a à faire entre l'espace alloué et l'espace utilisé. Selon les emplois prévus d'un ensemble de données, il est possible de lui donner une organisation particulière définie par la variable DSORG (organisation d'ensemble de données) : séquentielle, indexée-séquentielle, partitionnée, à accès direct. Cette organisation limite par la suite les accès permis à l'ensemble de données.

Ainsi, quelle que soit l'organisation, il est possible d'avoir un accès séquentiel à un ensemble de données; par contre il n'est pas possible d'avoir un accès sélectif rapide à un fichier séquentiel. Une fois le fichier alloué et son organisation fixée, il faut encore définir en détail le mode d'enregistrement de l'information.
Physiquement, l'écriture ou la lecture se font par bloc (quantité d'information vehiculée par une opération d'entrée/sortie). Un bloc est caractérisé par sa longueur maximum (variable BLKSIZE-taille de bloc) qui permet au système d'allouer les mémoires-tampon.

Le programmeur travaille au niveau des enregistrements logiques d'une longueur indiquée par la variable LRECL (longueur d'enregistrement logique). Enfin, ayant fixé une taille de bloc et une taille d'enregistrement logique on doit pouvoir préciser l'arrangement des enregistrements logiques sur les blocs. Un choix est possible entre plusieurs méthodes usuelles définies par la variable RECFM (format d'enregistrement, ou plus précisément technique d'enregistrement). Ici encore, la technique d'enregistrement ne présume pas du mode de lecture, ainsi un fichier séquentiel organisé en blocs de 10 images de cartes (80 col) peut très bien être considéré comme un fichier non bloqué contenant des enregistrements de longueur 800 caractères sauf pour le dernier bloc qui peut être incomplet.

Nous pouvons maintenant aborder la description des variables de spécification.

DSORG (Organisation de fichier)

Type de variable à valeur-chaîne, qui indique comment le fichier est organisé.
Ces chaînes sont prédéfinies.
Ce sont :

PS	pour séquentiel
IS	pour indexé-séquentiel
DA	pour accès direct
PO	pour fichier partitionné

RECFM (Format d'enregistrement)

Type de variable à valeur-chaîne, qui indique le type des enregistrements et de leur groupement.
Ces chaînes sont prédéfinies.
Ce sont :

F	pour enregistrements à longueur fixe
FB	pour enregistrements à longueur fixe, groupés
V	pour enregistrements à longueur variable
VB	pour enregistrements à longueur variable, groupés
VS	pour enregistrements logiques à longueur variable, qui peuvent s'étendre sur plusieurs enregistrements physiques.
VBS	idem que VS, groupés
V	pour enregistrements de type non-défini

LRECL (Longueur d'enregistrement logique)

Type de variable à valeur d'entier, qui indique la longueur d'un enregistrement.

BLKSIZE (Taille de bloc)

Type de variable à valeur d'entier, qui indique la longueur maximum du bloc, en cas de groupement.

REC (Technique d'enregistrement)

Type de variable à trois composantes, à

- une valeur-référence à une variable de type RECFM
- une valeur-référence à une variable de type LRECL
- une valeur-référence à une variable de type BLKSIZE

SPACE (Espace)

Type de variable à quatre composantes, à

- une valeur-chaîne, soit CYL ou TRK, pour des fichiers sur des mémoires magnétiques secondaires, exprimées en cylindres ou en pistes,
  ou bien
  une valeur d'entier, pour des fichiers exprimés en taille moyenne de blocs.
- une valeur d'entier qui indique l'espace primaire
- une valeur d'entier qui indique l'espace secondaire
- une valeur d'entier indiquant le nombre d'entrées dans la "directory" pour des fichiers partitionnés.

DSCB (Bloc de contrôle -descripteur- d'ensemble de données)

Type de variable à trois composantes, à

- une valeur-référence à une variable de type DSORG
- une valeur-référence à une variable de type REC
- une valeur-référence à une variable de type SPACE

Nous remarquons que les variables de type REC et de type DSCB sont de structure analogue à celle de "scalar" dans le langage PASCAL.

2.2.3.3. *Noms*.

Une variable peut être indiquée par des noms synonymes c'est-à-dire que plusieurs identificateurs peuvent être associés à une même variable.

2.2.3.4. *Distinction entre les noms et les valeurs-chaîne.*

Lexicographiquement la distinction est impossible. Au niveau de l'analyse syntaxique, certains cas peuvent être résolus, d'autres ne peuvent l'être qu'au niveau de l'analyse sémantique. Les cas d'ambiguité sont traités de la façon suivante : pour distinguer un nom d'une valeur-chaîne à un endroit où les deux sont permis, la première occurrence dans le programme-réseau d'une chaîne de caractères est considérée comme valeur-chaîne.

EXEMPLE :

Dans la déclaration d'unité suivante :

```
CPU  A:=IMAG.67/B:=C:=SYSDA;
```

le nom SYSDA pourrait être un nom synonyme de B et C au cas où il y aurait eu une déclaration préalable telle que :

```
UNIT  D:=SYSDA:=2314;
```

ou une valeur d'unité s'il n'y a pas eu d'autre déclaration.

2.2.3.5. *Valeurs initiales.*

En principe, chaque déclaration de variable comprend une affectation de valeur(s) initiale(s). Par la forme de la déclaration les références sont connues et les valeurs entier ou chaîne sont données soit explicitement, soit par défaut. Les variables LRECL, BLKSIZE et SPACE peuvent être déclarées sans affectation de valeur initiale. (Des opérations arithmétiques sont définies sur elles, qui permettent d'affecter une valeur à ces variables ultérieurement).

2.2.3.6. *Définitions.*

Par la suite nous appelons valeur relative d'une variable, la valeur qu'elle a, en la considérant comme une variable isolée et nous appelons valeur absolue d'une variable l'ensemble de sa valeur relative et des valeurs relatives de ses composantes et éventuellement des composantes de ces dernières et ainsi de suite, où on substitue successivement à chaque valeur référence à variable, la valeur ou le groupe des valeurs de la variable correspondante.

Une variable pour laquelle la notion de valeur absolue coïncide avec la notion de valeur relative, c'est-à-dire qui a un sens quand elle est isolée, est appelée variable simple. Une variable qui n'est pas une variable simple est appelée variable structurée.

2.3. Les instructions.

Toutes les instructions du LE.1 commencent par un mot-clé, distinct, sauf les instructions arithmétiques. Entre elles, deux instructions sont séparées par un point-virgule. Parmi les instructions on distingue : l'instruction vide, les commentaires, les instructions déclaratives, les instructions exécutables.

2.3.1. Les instructions déclaratives.

Les instructions déclaratives servent à déclarer :

- Les variables utilisées dans un programme LE.1 en leur affectant en même temps une valeur initiale; l'affectation d'une valeur initiale n'est pas obligatoire pour les variables de type LRECL, BLKSIZE et SPACE. (Nous remarquons l'analogie avec "block data" de FORTRAN).

- Les "TIFE", travaux insérés dans le flot d'entrée.

- Les valeurs qui sont données par défaut aux variables de spécification de type SPACE, REC et DSORG.

- Les préfixes nécessaires pour compléter les formes abrégées des noms physiques des fichiers, pour les variables de type DSN.

- La valeur du niveau de sévérité permettant de décider de l'abandon du programme-réseau, lorsque l'exécution d'une instruction élémentaire se fait d'une façon anormale.

La forme d'une instruction déclarative est différente, selon qu'il s'agit de la déclaration d'une variable de localisation ou d'une variable de spécification.

Excepté pour la déclaration d'un travail inséré dans le flot d'entrée, la règle suivante est applicable : plusieurs déclarations de variables d'un même type peuvent se faire en une seule instruction déclarative, séparées par des virgules; le mot-clé en tête de l'instruction n'est pas répété.

Pour pouvoir définir complètement une variable composée, dans un programme LE.1, il faut qu'elle soit déclarée, et, en même temps, toutes les variables, composées ou simples, desquelles dépend sa valeur absolue.
A chaque instant, dans une instruction déclarative, on peut associer un ou plusieurs noms synonymes à une variable.

2.3.1.1. *Déclaration des variables.*

2.3.1.1.1. Déclaration des variables de localisation.

Les instructions déclaratives pour les variables de localisation se composent, d'une ou de plusieurs déclaration(s) partielle(s). Une déclaration partielle est soit une déclaration explicite, soit une référence à une variable. Les déclarations partielles sont séparées entre elles par des barres de division.

La signification physique de A/B est "B est sur A"; par exemple un disque B est sur une unité de type 2314 A. Dans le même ordre que :

CPU UNIT VOLUME (VOLLIST) DSN (MBR,LMBR),

on écrit la variable de type CPU le plus à gauche dans une instruction déclarative, la variable de type DSN (ou MBR, ou LMBR) le plus à droite. Dans une instruction déclarative, chaque partie de l'instruction qui se trouve à gauche de la déclaration partielle pour la variable que l'on veut déclarer en définitive, peut être remplacée d'un seul trait par un des noms qui refèrent à cette partie et qui lui a été attribué dans une déclaration antérieure.

Le mot-clé en tête correspond au type de la variable qui est immédiatement à sa droite, et ne correspond donc pas au type de la variable définie en définitive (exception faite pour les variables de type CPU).

EXEMPLES :

1. CPU IMAG.67/2314/2OMVT3/DATASET:=SYS2.AWLOAD

Déclaration d'un fichier de données(existant ou à créer) SYS2.AWLOAD au nom de DATASET, se trouvant sur le volume disque 2OMVT3, monté sur une unité de type 2314 sur l'ordinateur IMAG.67.

2. CPU C1:=SAC.91/2314/V1:=SACOO1;

Déclaration d'un volume disque SACOO1 au nom de V1, monté sur une unité de type 2314 sur l'ordinateur SAC.91 au nom de C1.

3. CPU C2:=C1/2314/V2:=SACOO2;

Déclaration analogue à la précédente. Un nom synonyme C2 pour C1 est donné à l'ordinateur SAC.91.

4. VOLUME V1/D1 :=TESTFILE1, V2/D2:=TESTFILE2;

Déclaration deu deux fichiers de données se trouvant sur les deux volumes déclarés dans les exemples précédents.

5. DSN DATASET/(MBR1, MBR2);

Déclaration des membres MBR1 et MBR2 du fichier SYS2.AWLOAD du premier exemple.

REMARQUE.

Depuis notre rapport de DEA [DECA], deux modifications importantes ont été apportées à la forme des instructions déclaratives des variables de localisation, notamment l'écriture de gauche à droite des déclarations quant à la hiérarchie des types et subséquemment l'emploi d'un mot-clé différent de celui correspondant à la variable déclarée finalement. Ces modifications sont discutables puisque les instructions y ont perdu en clarté. Elles ont été dictées par des raisons d'implémentation. En effet, pour implémenter le langage tel que défini dans [DECA], le compilateur a besoin d'une pile puisque les décisions qu'il prend ne peuvent porter que sur des valeurs absolues. Avec la définition actuelle, il n'en est plus rien, ce qui réduit la zone de travail du compilateur et facilite la compilation. Il est évident que la gestion de cette pile ne pose pas de problème théorique. Cette forme de déclaration a été adoptée à titre temporaire pour avancer plus rapidement l'implémentation du compilateur. D'autre part, le mot-clé en tête actuellement est moins représentatif, mais son choix est imposé en conséquence de la première décision, comme nous le montrons à l'aide de l'exemple suivant :

Déclaration ancienne forme *(1)* VOLUME V:=2OMVT3/2314/IMAG.67;
Déclaration nouvelle forme *(2)* CPU IMAG.67/2314/V:=2OMVT3;
Nouvelle forme prohibée *(3)* VOLUME IMAG.67/2314/V:=2OMVT3;

Dans *(1)*, avant de pouvoir décider que V n'a pas déjà été déclaré, il faut remonter jusqu'au CPU correspondant (il n'est pas exclu, qu'il y ait un volume de même nom physique sur un autre ordinateur), donc il faut empiler les renseignements de la déclaration avant de pouvoir faire les tests pour la génération.

Dans *(2)*, on sait tout de suite que la première valeur que l'on rencontre est celle d'un CPU (en général celle correspondant au mot-clé en tête).

On peut vérifier immédiatement sa déclaration. En progressant dans l'analyse de l'instruction, pas à pas, on peut vérifier, en se basant sur l'information qui précède immédiatement. Le fait que le mot-clé CPU ne correspond pas effectivement à la déclaration de volume qu'on est en train de faire n'est à éviter qu'au prix d'une pile, dont nous avons justement voulu nous débarasser.

En effet, dans *(3)*, le mot-clé porte sur le dernier, mais à ce moment on ne peut pas décider tout de suite à quel type de variable correspond la première valeur. On ne peut le faire qu'après avoir compté les barres.

2.3.1.1.2. Déclaration des variables de spécification.

Pour les variables composées, dont les composantes sont des références (à des variables d'un autre type), il est permis de remplacer ces références par une déclaration complète des variables référencées, au mot-clé près, qu'on ne met pas dans ce cas. Par contexte on connait en effet le type de la variable ainsi déclarée. L'affectation d'une valeur initiale peut être explicite, implicite pour les variables de type SPACE, REC, DSORG (valeurs par défaut) ou omise pour les variables de type SPACE, LRECL, BLKSIZE. Les variables auxquelles on associe une valeur par défaut ne doivent pas être écrites dans la déclaration, mais le séparateur entre une telle variable et la composante qui suit est gardé; s'il s'agit d'une composante en dernière position dans la déclaration, le séparateur n'est pas nécessaire.
L'affectation d'une valeur peut être explicite :

- soit par affectation d'une valeur

  A(nom):=B(valeur);

- soit par référence à une entité qui dans sa hiérarchie comporte un niveau du type de la variable affectée (dans la 1ère version l'entité est de type DSN)

  A(nom):=C(nom d'une entité);

Si ce type d'affectation explicite n'est pas utilisé, une valeur par défaut peut être fournie.

Cette valeur par défaut est, soit une valeur prévue dans le compilateur pour ce type de variable, soit une valeur indiquée pour ce type de variable dans une instruction DEFAULT.

Les valeurs référence à variable peuvent être soit des références directes, soit des références à des variables de type DSN qui contiennent les valeurs correspondantes à la variable de type recherché dans leur partie spécifications.

Les valeurs par défaut implicites sont :

```
SPACE:=(800,(100,50,2))
RECFM:=FB
LRECL:=80
BLKSIZE:=800
DSORG:=PS;
```

EXEMPLES :

1. `RECFM  RF:=VBS;`
2. `LRECL:=80;`
3. `BLKSIZE:=800;`
4. `REC  RE:=(FB,80,800);`
   équivalent à
   `REC  RE:=(FB,LR,BL);`
   en utilisant les déclarations des exemples 2 et 3.
5. `SPACE  SP:=(TRK,(100,20,2));`
6. `DSORG  DO:=PS;`
7. `DSCB  DC1:=(SP,RE,DO);`
   ou
   `DSCB  DC1:=((TRK,(100,20,2)),(FB,80,800),PS);`

8. DSCB DC2:=(SP,(F,80,80));
Déclaration d'une variable DC2 de type DSCB pour fichier avec les mêmes spécifications que celui correspondant à DC1 de l'exemple précédent au stockage près.

9. DSCB DC3:=DATASET;
Si DATASET a été déclaré comme nom d'une variable de type DSN représentant un fichier existant, DC3 prend les valeurs correspondantes dans les spécifications du fichier référencé (on suppose qu'il n'existe aucune variable de type DSCB au nom de DATASET).

2.3.1.2. *Déclaration d'un travail inséré dans le flot d'entrée.*

La déclaration entraine la construction d'un fichier spécial (dans le système d'entrées/sorties simultanées), avec sa variable de contrôle associée (de type DIS, non accessible à l'utilisateur). Dans la déclaration un nom est donné à la variable associée au fichier. Le fichier-travail étant destiné à être pris en charge par le système de contrôle de l'ordinateur où on demandera son exécution, sa partie cartes contrôle devra respecter leurs règles d'écriture, de carte en carte avec les tabulations nécessaires. En plus, comme le contenu d'un travail inséré peut être quelconque (cartes de données entre autres), la fin d'un tel travail n'est repérable ni lexicographiquement, ni syntaxiquement. La reconnaissance de la fin du travail se fait par comparaison d'une chaîne donnée en tête du travail avec une chaine donnée en fin de travail. Deux paramètres indiquent en quelle position de l'image de carte on peut attendre le début de la chaine finale et, sur quelle longueur il faut comparer les deux chaînes. Par défaut, la comparaison se fera à partir de la première position de l'image de carte, sur une longueur égale à la longueur de la première chaîne, étendue à droite avec des blancs au besoin.

EXEMPLES :

1.
```
INP  TRAVAIL1:='**' (3,2);
//TMVT1  JOB(5161,580),'DECALUWE',MSGLEVEL=(1,1),
//       REGION=250K,CLASS=C
//       EXEC   ASMGC
//ASM.SYSIN  DD  DSN=SYS2.AWSOURCE(OSCOMP),
//       DISP=OLD,VOL=SER=20MVT3,UNIT=2314
//
   **
```

2.
```
INP  TRAVAIL2:='FIN DE TRAVAIL';
//TMVT2  JOB(5161,580),'DECALUWE',MSGLEVEL=(1,1),
//       REGION=250K,CLASS=C
//       EXEC   ASMGC
//ASM.SYSIN  DD  DSN=SYS2.AWSOURCE(OSCOMP),
//       DISP=OLD,VOL=SER=20MVT3,UNIT=2314
//
FIN DE TRAVAIL
```

Dans les exemples, deux travaux insérés dans le flot d'entrée sont déclarés, avec comme noms logiques, respectivement TRAVAIL1 et TRAVAIL2.

2.3.1.3. *Déclaration des valeurs par défaut.*

La fonction de ce type de déclaration est de permettre de fixer des valeurs par défaut autres que celles définies en standard dans le compilateur. Cette instruction déclarative commence par le mot-clé DEFAULT suivi du mot clé qui correspond au type de la variable déclarée par défaut. La valeur associée s'écrit d'une façon analogue à celle des autres instructions déclaratives.

Les déclarations par défaut peuvent être répétées au courant du programme LE.1. La dernière valeur indiquée est utilisée subséquemment. Des valeurs par défaut peuvent être déclarées pour les variables de type SPACE, LRECL et BLKSIZE.

EXEMPLE :

```
DEFAULT  LRECL:=800,  BLKSIZE:=7200;
```

2.3.1.4. *Déclaration des préfixes.*

L'instruction QUALIFY a pour but de simplifier l'écriture des noms de fichiers. Les noms de fichiers sont en général complexes.
Par exemple, en OS/360 sur le système de l'IMAG, le nom identifie le propriétaire et a la forme suivante :

```
S<s>.P<p>.<n>.<x>
```

Où s,p et n représentent respectivement le numéro de service, numéro de programmeur et le nom du propriétaire. Seule la partie x est réellement intéressante pour le programmeur. Le préfixe S<s>.P<p>.<n> est long à écrire et de ce fait source d'erreur. Il est par ailleurs dépendant des conventions propres à une exploitation donnée. Il est donc pratique de pouvoir le déclarer par un préfixe pour chaque ordinateur du réseau par l'instruction.

```
QUALIFY <nom d'unité centrale> <préfixe> ;
```

Un préfixe déclaré dans une instruction QUALIFY pour un ordinateur donné, est considéré comme faisant partie de chaque nom physique de fichier, s'il est écrit sous une forme abrégée c'est-à-dire commençant par un point. Les déclarations QUALIFY peuvent apparaître en différents points du programme LE.1. En ce cas, la dernière rencontrée lexicographiquement est considérée.

EXEMPLE :

QUALIFY IMAG.67 BY S5161.P0580.DECALUWE
permet de référencer le fichier
S5161.P0580.DECALUWE.CMPLR
seulement par
.CMPLR

2.3.1.5. *Déclaration du niveau de sévérité.*

L'exécution d'un programme LE se fait séquentiellement. A la fin de chaque instruction, une valeur entière est fournie par l'interpréteur pour indiquer comment s'est déroulée l'instruction. La valeur 0 indique que tout s'est passé normalement, une valeur 16 indiquant une erreur fatale.
L'instruction LEVEL permet au programmeur de fixer le niveau de code retour au dessus duquel il veut abandonner l'exécution.

Le mot-clé en tête de cette instruction déclarative est LEVEL, suivi d'un entier. L'entier représente le niveau de sévérité toléré à l'exécution (les codes retour ne devront pas le dépasser).
A défaut d'une déclaration LEVEL la valeur zéro est prise, ce qui fait qu'aucune erreur n'est tolérée.
Les instructions LEVEL peuvent apparaître en différents points du programme, dans ce cas la dernière rencontrée fixe la valeur du code retour admissible.

EXEMPLE :

LEVEL 4;

2.3.2. Les instructions exécutables.

Les instructions exécutables servent à :

- faire exécuter des fonctions utilitaires portant sur des fichiers qui se trouvent sur un ou plusieurs ordinateurs du réseau.
- faire exécuter un travail inséré dans le flot d'entrée.
- faire évaluer des expressions arithmétiques, dont les variables sont de type LRECL, BLKSIZE ou SPACE et les constantes des nombres entiers.
- faire monter ou démonter des volumes.
- envoyer des messages.

Chaque variable utilisée dans une instruction exécutable doit avoir été préalablement déclarée sauf les variables de type CPU, si l'on n'utilise que les chaînes valeurs-chaîne associées.

2.3.2.1. *Les fonctions utilitaires.*

2.3.2.1.1. Créer un fichier.

La déclaration d'un nouveau fichier ne suffit pas à son existence. On est obligé de créer explicitement le fichier par une instruction CREATE dans laquelle on associe la localisation du fichier à ses spécifications.

Un fichier existant peut être manipulé, après simple déclaration (A l'exécution du programme LE.1, les spécifications nécessaires du fichier, sont cherchées dans son descripteur physique et passées aux variables du programme LE.1).

EXEMPLE :

```
QUALIFY IMAG.67 BY S5161.P0580.DECALUWE;
CPU    IMAG.67/2314/IMAG80/DATA:=.XREF;
DSCB   SPECIF:=((CYL,(2,1,)),(VB,200,1600),DA);
CREATE  DATA (SPECIF);
```

Schématiquement on peut représenter le pointeur physique sur le premier enregistrement libre du fichier comme suit :

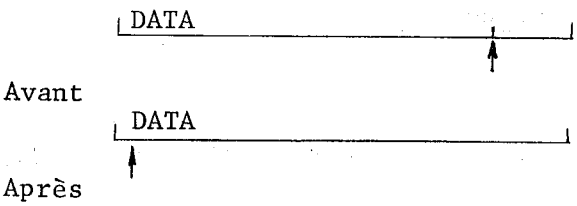
2.3.2.1.2. Vider un fichier.

L'instruction EMPTY sert à effacer le contenu d'un fichier, sans que son descripteur physique ne disparaisse.(A fortiori, s'il est catalogué, il le reste).

EXEMPLE :

EMPTY DATA;

Schématiquement



2.3.2.1.3. Détruire un fichier.

Cela se fait par l'instruction DELETE; le descripteur physique du fichier est détruit.

EXEMPLE :

DELETE DATA;

2.3.2.1.4. Cataloguer et décataloguer dans le catalogue du système local.

Par les instructions CATALG et UNCATALG .

EXEMPLE :

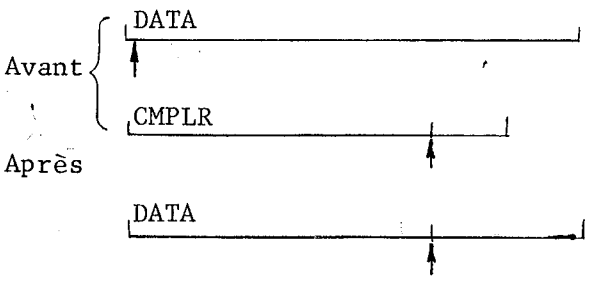
```
CATALG   DATA ;
UNCTLG   DATA   ON   V1;
```

2.3.2.1.5. Copier un fichier (membre ou liste de membres).

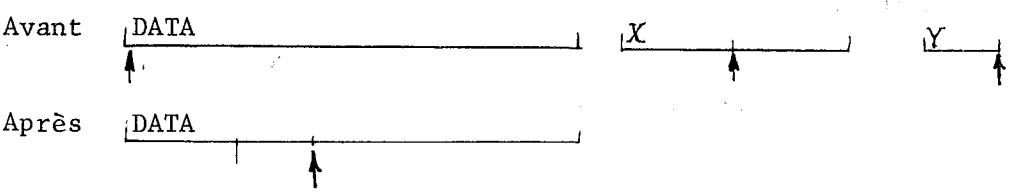
On peut copier un fichier, un membre ou une liste de membres sur un fichier créé par l'instruction COPY.

EXEMPLES :

1. `COPY  CMPLR  TO  DATA;`



2. `COPY  (X, Y)  TO  DATA;`



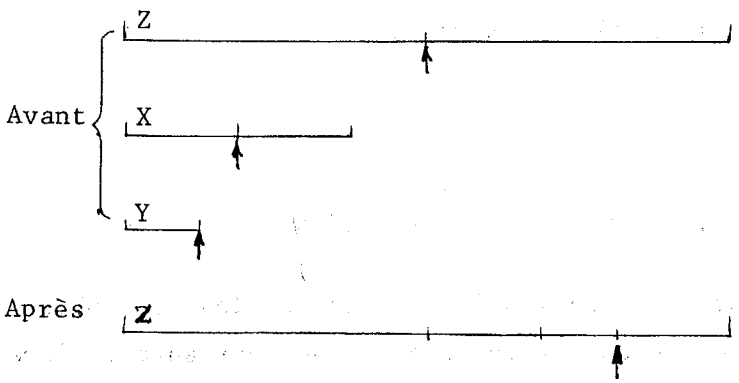
2.3.2.1.6. Concaténer un fichier (membre ou liste de membres) à un fichier.

Au fait la concaténation est une copie, au besoin répétée .Le fichier final (le récepteur), qui est l'ensemble des fichiers concaténés, se trouve à l'endroit du premier fichier, qui doit donc être prévu assez grand lors de sa création.

EXEMPLE :

ADD (X,Y) TO Z;
(L'endroit où se trouve Z est supposé suffisamment grand)

Schéma



2.3.2.1.7. Imprimer un fichier (membre ou liste de membres).

Sans spécification de l'ordinateur qui l'exécute, l'impression est faite en local, mais on peut demander l'impression de n'importe quel fichier se trouvant sur n'importe quel ordinateur. Nous remarquons que l'instruction LIST est également une demande de copie, au sens large du terme, d'une unité périphérique sur une autre.

EXEMPLE :

LIST DATA AT IMAG.67;

2.3.2.2. *L'exécution des travaux insérés dans le flot d'entrée.*

L'instruction RUN commande l'exécution d'un travail inséré dans le progra réseau, sur un ordinateur défini. Optionnellement on peut demander que la liste sorte sur un ordinateur autre que celui où s'exécute le travail.

EXEMPLES :

```
1.          RUN  INPJOB1  AT  SAC.91  LIST  AT  IMAG.67;

2.          RUN  INPJOB2  AT  SAC.91;
```

Dans le second exemple la liste sortira sur l'ordinateur SAC.91 où est exécuté le job. L'instruction RUN peut être soumise par un troisième ordi teur, CIRCE.75 par exemple.

2.3.2.3. *Instructions arithmétiques.*

On peut faire de l'arithmétique sur les variables de type LRECL, BLKSIZE et SPACE. A une variable ou à une constante entière, on peut ajouter une série de termes, chaque terme étant soit une variable, soit un entier que multiplie une variable, en additionnant ou en soustrayant.

Dans le cas des variables SPACE on n'admet de l'arithmétique que sur des SPACE ayant une première composante de même valeur, c'est-à-dire corresp dant à des espaces exprimés d'une même façon, soit en cylindres, soit en pistes, soit en taille moyenne de blocs.

EXEMPLES :

```
1.          LRECL  LO , L1:=DATA1; L2:=DATA2;  L3:=4;
            LO:=L1+2*L2-L3;
```

2. BLKSIZE BO;
BO:=1600;

3. SPACE SO;
SPACE S1:=(TRK,(100,20,2));
SPACE S2:=DATA1;
SO:=DATA1+2*S1;
(DATA1 est le nom d'un fichier par exemple ; si le SPACE correspondant à DATA1 est exprimé en blocks;
2*S1 est équivalent à un SPACE de valeur (TRK,(200,40,4))).

2.3.2.4. *Les instructions de montage et de démontage d'un volume.*

Elles servent à demander explicitement le montage ou le démontage d'un volume; s'il s'agit d'une bande , au montage on spécifiera comment on la veut montée (en écriture ou en protection d'écriture) et comment elle est étiquettée (standard, non étiquetée, ou étiquetée sans qu'on utilise l'étiquette).

Dans spécifications,l'option "non étiqueté" et/ou l'option "protection d'écriture" sont prises en compte.

EXEMPLES :

1. MOUNT D2OMVT3;
Montage d'un disque (si par ex. CPU .../2314/D2OMVT3:=2OMVT3;)
A cause des déclarations, il n'y a pas ambiguïté entre la demande de montage d'un disque ou d'une bande.

DISMOUNT , D2OMVT3;
Démontage du disque.

2. MOUNT B000238(RING,SL);
Montage de la bande 000238 étiquetée standard, en écriture,

DISMOUNT B000238;
Démontage de la bande.

3. MOUNT B000218 (,SL);
MOUNT B002004 (RING,);

4. MOUNT B000236 (,);
Equivalent à
MOUNT B000236;

2.3.2.5. *Les envois de messages.*

L'instruction MSG peut être utilisée à n'importe quel endroit du programme, puisque la seule variable que l'on y utilise peut aussi bien être désignée par son nom que par sa valeur puisqu'il s'agit de la variable de type CPU. Cette instruction entraine l'envoi du message qu'elle contient à l'opérateur de l'ordinateur à qui le message est adressé.

EXEMPLE : MSG IMAG.67 'MESSAGE' ;

2.3.3. L'instruction vide

Elle n'a aucun effet sur l'exécution du programme LE.1.

EXEMPLE :

MOUNT B218 ; ; ; DISMOUNT B218

deux instructions vides

2.3.4. Les commentaires

Ils sont sans effet sur l'exécution.

EXEMPLE :

```
COMMENT 'LE.1  EST  DEFINI  PAR UNE GRAMMAIRE LL(1);';
```

2.4. Exemples de programmes.

① Travail-réseau, soumis à SAC.91 contenant un travail inséré OS/360 dans le texte à faire exécuter par IMAG.67,avec listing du travail à CIRCE.75 (Le listing du travail-réseau est produit à SAC.91, où il est soumis).

```
NETIN(5001,9589,1),(DECALUWE,FSTPROG);
INP  ALPHA:='DELIMITER'(1,3)
//JOB  JOB(5161,0580),'DECALUWE', MSGLEVEL=(1,1),
//     PRTY=6, REGION=150K
//     EXEC   PGM=IEBUPDTE, PARM=MOD
//SYSPRINT  DD  SYSOUT=A
//SYSUT1    DD  DSNAME=P5161.S0580.DECALUWE.ACOMP, UNIT=2314, DISP=(OLD,KEEP),
                VOLUME=SER=IMAG80, DCB=(RECFM=F, LRECL=80, BLKSIZE=1600)
//SYSIN   DD   *
./     CHANGE  NAME=PHASEA, LIST=ALL, UPDATE=INPLACE
./     NUMBER  SEQ1=ALL, NEW1=100, INCR=10
TEXTE DE REMPLACEMENT POUR LA CARTE 36                              00000036
/*
//
DEL   ;
RUN  ALPHA  AT  IMAG.67  LIST AT CIRCE.75;
NETOUT
```

Cet exemple correspond à la soumission d'un travail sur un site distant, cas usuel des terminaux lourds.

(2) Transfert d'un fichier sur disque de IMAG.67 à SAC.91.

```
NETIN(5161,0580,2),(DECALUWE, SNDPROG);
COMMENT LE FICHIER A EXISTE A IMAG.67, ON LE DECLARE;
CPU  IMAG.67/2314/IMAG80/A:=S5161.P0580.DECALUWE.IN;
COMMENT LE FICHIER B EST A DECLARER ET A CREER AVANT DE POUVOIR RECEVOIR
LA COPIE DE A;
CPU  SAC.91/2314/SAC005/B:=SAC5001.DECALUWE(OUT)
SPACE  SPA:=A;
DSCB DB (SPA,(FB,80,800),PS);  CREATE  B  (DB) ;
COPY  A  TO  B;
NETOUT
```

(3) Travail-réseau soumis à IMAG.67, contenant un travail inséré OS/360 à exécuter à SAC.91 et qui utilise un fichier qui est la concaténat: d'un fichier résidant à IMAG.67 et d'un fichier résidant à CIRCE.75. On récupère le listing à IMAG.67 et on transfère le fichier avec les résulta de l'exécution à CIRCE.75 et à IMAG.67. Puis on détruit le fichier avec résultats à SAC.91.

```
NETIN(5161,0580,3),(DECALUWE,TRDPPROG);
CPU  A:=IMAG.67, B:=CIRCE.75, C:=SAC.91;
QUALIFY  A  BY  S5161.P0580.DECALUWE;
QUALIFY  C  BY  SAC5001.DECALUWE;
COMMENT  DEFINITION  DE  S5161.P0580.DECALUWE.FA
A  IMAG.67;
CPU  A/2314/MVT009/.FA;
COMMENT DEFINITION DU TRAVAIL INSERE DANS LE FLOT D'ENTREE ALPHA;
INP  ALPHA:='DELIMITER'(10,5)
//JOB  JOB...
...
//GO  EXEC  ASMGCLG
//GO.SYSIN  DD  UNIT=2314, VOL=SER=SAC009,
            DSN=SAC5001.DECALUWE.FAB,
            DISP=(OLD,DELETE)
```

```
//GO.SYSRES  DD  UNIT=2314, VOL=SER=SACO1O,
        DSN=SAC5OO1.DECALUWE.RESULTC,
        DISP=(NEW,KEEP),
//      SPACE=(CYL,(2O,1O,2)),DCB=(BLKSIZE=8OO,LRECL=8O,RECFM=FB)
//
          DELIM;
COMMENT      DECLARATION ET CREATION DE
    SAC5OO1.DECALUWE.FAB A SAC.91;
REC  RA:=(FB,8O,8OO);
  CPU  C/2314/SACOO9/INC:=.FAB;
SPACE  SPFAB , S1:=DSA , S2:=DSB;
SPFAB:=S1+S2;
CREATE  INC (SPFAB, RA, PS);
COMMENT TRANSFERT AVEC CONCATENATION IMPLICITE DES DEUX FICHIERS CONSTITU
COPY  (DSA, DSB) TO  INC;
COMMENT DEFINITION ET CREATION DES FICHIERS-RESULTATS;
CPU  A/2314/MVTOO4/RESA:=RESULTA;
CPU  B/2314/CIROO5/RESB:=RESEAU.RESULTB;
CPU  C/2314/SACO1O/RESC:=.RESULTC;
SPACE  SPRES:=(CYL,(2O,1O,2));
DSCB  SPECIFS:=(SPRES, RA, PS );
CREATE  RESA (SPECIFS);
CREATE  RESB (SPECIFS);
CREATE  RESC (SPECIFS);
COMMENT EXECUTION;
RUN ALPHA AT SAC.91 LIST AT IMAG.67
COMMENT TRANSFERT DES FICHIERS-RESULTATS;
COPY  RESC  TO  RESA;
COPY  RESC  TO  RESB;
COMMENT DESTRUCTION DU FICHIER-RESULTATS  A  SAC.91.
DELETE  RESC;
NETOUT
```

2.5. Limitations actuelles et extensions à prévoir.

Il est essentiel de voir le LE.1 comme un processeur d'un langage plus élaboré et plus puissant. Actuellement, dans le LE.1, la structure de blocs se réduit à un seul bloc qui comprend des ordres à exécuter séquentiellement sans possibilité de sauts.

Outres les particularités au cours de sa description, quelques points sont à discuter. Les options par défaut, les références, permettent d'exprimer des valeurs "les mêmes que" et d'aucunes façons des valeurs "les mêmes que, sauf". Peut-être y a-t-il moyen d'exprimer la négation ou l'exception sous une forme simple en LE, le point est à examiner.

Les fichiers que l'on traite explicitement dans le LE.1 ne sont accessibles qu'en entier ou membre par membre (s'il s'agit de fichiers partitionnés). En particulier les transferts se font ainsi et peuvent donc être coûteux.

Quant aux travaux insérés dans le texte, l'établissement des liens pour un fichier déclaré ainsi, et le reste du programme externe, est laissé aux soins du programmeur-réseau. Il lui convient donc de répéter exactement les spécifications du fichier en LE.1 et en langage de contrôle de l'ordinateur de destination. Notons au passage la dualité de ces deux formes d'écriture.

Alors qu'en OS/360 la durée de vie d'un fichier est spécifiée par le sous-paramètre 'DISPOSITION' de l'instruction DD (data definition)

```
//SYSUT1  DD  DSN=X,SPACE=(...), UNIT=...,VOL=SER=...
              DISP=(NEW,DELETE)
```

en LE.1 il faut écrire :

```
CPU  ORDINATEUR/UNITE/VOLUME/X;
CREATE X;
...
DELETE X;
```

Par ailleurs, le contrôle conditionnel de l'exécution d'un travail par la seule instruction LEVEL est rudimentaire. Une écriture conditionnelle plus élaborée serait nécessaire. Les variables sur lesquelles devraient porter

les tests étant d'une part les codes retour des instructions précédemment exécutées et d'autre part les attributs des ordinateurs du réseau.
Aussi pourrait-on envisager un partage des charges automatique entre les ordinateurs. Pour les travaux insérés il y a à chercher encore comment introduire des commandes d'exécution de travaux qui ont besoin de données qui se trouvent ailleurs.

Une prochaine version du LE.1 pourra peut-être aussi paraître plus hétérogène si d'autres ordinateurs que des IBM/360 feront effectivement partie du réseau.

La version actuelle par contre se prête bien à l'adaptation du compilateur à une version conversationnelle.

3. IMPLEMENTATION

Nous discutons ci-dessous la façon d'enregistrer la grammaire du LE.1 et l'analyseur utilisé. La génération des instructions et des ordres est détaillée. Egalement, la compilation est située dans le cadre de l'implémentation du réseau SOC lui-même.

Quelques réflexions sur nos expériences avec le langage ALGOLW, utilisé pour l'écriture d'un prototype du compilateur LE.1, sont données.

3.1. La grammaire, enregistrement et méthodes d'analyse et de génération.

3.1.1. Enregistrement de la grammaire et analyse syntaxique.

Du point de vue de l'implémentation du LE.1, deux facteurs sont importants; l'encombrement mémoire de l'ordinateur et la rapidité de l'algorithme d'analyse qui sera appliqué. En particulier, dans le cadre du projet SOC, c'est le premier facteur qui importe le plus. Mais les investigations vont plus loin, puisqu'il faut tenir compte du fait que n'importe quelle méthode n'est pas appliquable à n'importe quel langage, ou peut aussi exiger beaucoup de transformations préalables du langage, souvent d'une façon artisanale.

Le langage LE.1, défini par une grammaire strictement LL(1),utilise un algorithme d'analyse descendante approprié [GRIF]. La rapidité de l'algorithme tient au fait que la grammaire n'est non seulement LL(1), mais qu'en outre elle possède la propriété que chaque règle de grammaire et chaque alternative d'une règle commencent par un symbole terminal. La méthode choisie est décrite en détail plus loin.

En ce qui concerne l'encombrement mémoire, nous avons fait des estimations sur ce qu'auraient pû donner d'autres méthodes traditionnelles, notamment quelques méthodes de précédence (auxquelles on pourrait songer en raison de leur performance) et nous avons pû constater que notre méthode est très compétitive.

Approximativement, la grammaire comporte 45 règles, avec 60 symboles terminaux et 45 symboles non-terminaux. Procédant par réductions successives, les méthodes de précédence se basent sur des relations dites de précédence, établies entre les différents symboles de la grammaire. Deux symboles quelconques de la grammaire sont reliés par une des relations de précédence suivantes : absence de relation, $\lessdot$,=, $\gtrdot$. La façon de définir les relations dépend de la méthode de précédence particulière utilisée. Si on décide de représenter ces relations respectivement par 0,1,2,3, il faut au moins deux bits par relation de précédence. En admettant qu'on représente un symbole de la grammaire par un octet (le maximum de 256 symboles n'est pas atteint ici), pour représenter 45 règles comportant 8 symboles en moyenne chacune, on compte 360 octets auxquels il faudrait cependant ajouter des descripteurs et/ou des pointeurs pour chaque règle en vue de la consultation. Ainsi on arrive à environ 500 octets. Remarquons encore que la grammaire s'allonge en moyenne d'un huitième par les transformations qu'on est amené à lui faire subir, pour la rendre de précédence. Nous avancerons donc 600 octets.

Pour la méthode de précédence selon Floyd [FLOY], on a besoin d'enregistrer la grammaire et une matrice de précédence. Cette matrice comprend les relations entre les symboles terminaux, c'est-à-dire 60x60relations=3600relations, ce qui revient à 7200 bits ou 900 bytes. Coût total en mémoire :
600 octets + 900 octets = 1500 octets

Pour les méthodes de précédence selon Wirth [WIRT] et Colmerauer [COL1][COL2] il faut enregistrer, en outre de la grammaire, une matrice comprenant les relations entre les symboles terminaux et non-terminaux, c'est-à-dire de 105 x 105 relations. Coût total en mémoire : 2200 octets.

Une variante de cette méthode de représentation des relations de précédence

(utilisation des fonctions de précédence), [AHO] permet de ramener le nombre de N^2 relations à 2N relations et donc de diminuer sensiblement l'encombrement mémoire de la matrice de précédence, mais la transformation n'est pas toujours possible. En plus, elle est moins efficace quant à la localisation des erreurs syntaxiques.

Mentionnons encore au passage une méthode de précédence récente qui serait plus facile à appliquer que les autres, mais pour laquelle il faut compter un encombrement mémoire comparable à celui exigé par les méthodes de Colmerauer et de Wirth. Notons que dans les méthodes de précédence l'analyseur lui-même est très peu encombrant.

Avec notre méthode, nous arrivons à environ 1700 octets de mémoire, mais les appels aux fonctions sémantiques qui s'y insèrent automatiquement aux occurrences des symboles terminaux dans les règles de grammaire, y sont déjà comptés. Pour l'enregistrement proprement dit on peut compter environ 1350 octets (notons encore que les appels aux fonctions sémantiques dans les grammaires de précédence s'insèrent moins facilement que dans une grammaire de type LL(1)).

En outre, notre méthode est passible d'améliorations dans l'enregistrement de la grammaire, ce qui réduirait encore légèrement sa taille en mémoire, mais permettrait en même temps une analyse plus rapide. Cette modification sera discutée après la présentation de la méthode actuellement implémentée. La grammaire utilisée pour l'implémentation du LE.1 est donnée en annexe. Pour vérifier que cette grammaire est bien LL(1), nous nous sommes servis d'un programme de MM. Griffiths et Peltier disponible à l'IMAG.

3.1.2. Méthode d'enregistrement choisie.

L'algorithme d'analyse déterministe décrit plus loin est une consultation d'arbre de haut en bas, de gauche à droite. Dans cette optique, la grammaire est enregistrée dans l'esprit de Cheatham et Sattley [CHEA] .

Pour un symbole terminal ou non-terminal apparaissant à un endroit quelconque de la grammaire, nous notons :

1°/ Si ce symbole est succédé ou non par un autre dans la règle de grammaire

2°/ Si ce symbole possède une alternative ou non dans la règle de grammaire (notons que seulement les symboles terminaux peuvent en avoir une, puisque toutes les règles et toutes les alternatives d'une règle commencent par un symbole terminal).

Les symboles terminaux et non-terminaux (une centaine) sont représentés, cha par un code numérique tenant dans un octet.

Pour les terminaux, l'indication d'alternative se fait par son adresse en 3 octets, dont le bit de gauche est réservé pour indiquer la présence ou non d'un successeur, qui le cas échéant, est immédiatement attenant en mémoire, s'il n'y a pas d'alternative, l'adresse est zéro.

Pour les non-terminaux, une adresse de 3 octets pointe immédiatement sur la règle de grammaire qui les définit; celle-ci commence nécessairement par un terminal.

Pour l'enregistrement on considère comme terminal: mot-clé, identificateur, entier, chaîne et élément nul de la grammaire. L'élément nul de la grammai apparaît, à la fin des règles, s'il y a lieu et est noté explicitement (par "RIEN"). Les appels aux fonctions sémantiques sont générés automatiquement à chaque occurrence d'un symbole terminal dans la grammaire, sauf à l'occurrence de l'élément nul, sous forme d'une étiquette notée sur 2 bytes à laquelle on saute quand on la rencontre à l'analyse.

Prenons un exemple pour illustrer la méthode d'enregistrement de la grammai Prenons un langage L décrit par une grammaire G_o. G_o est défini par le quadruplet (N_o, T, N, R), où N_o représente l'axiome de G_o, T l'ensemble des symboles terminaux de G_o, N l'ensemble des symboles non-terminaux de G_o et I l'ensemble des règles de la grammaire.

Nous notons :

$$G_o = (N_o, T, N, R)$$

N_o axiome

$$T = \{t_1, t_2, \ldots, t_u\}$$

$$N = \{N_1, N_2, \ldots, N_v\}$$

$$R = \{R_1, R_2, \ldots, R_w\}$$

et τ élément nul de la grammaire.
Soit Φ l'ensemble des fonctions sémantiques

$$\Phi = \{\varphi_1, \varphi_2, \ldots, \varphi_r\}$$

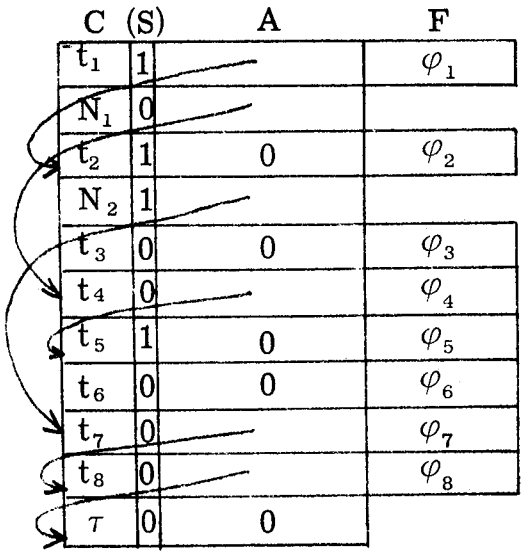
Le langage L est défini par la grammaire sous forme normale de Backus (BNF) comme suit :

$$\left.\begin{array}{l} N_o ::= t_1\ N_1 \\ \quad\quad | \ t_2\ N_2\ t_3 \end{array}\right\} \equiv R1$$

$$\left.\begin{array}{l} N_1 ::= t_4 \\ \quad\quad | \ t_5\ t_6 \end{array}\right\} \equiv R2$$

$$\left.\begin{array}{l} N_2 ::= t_7 \\ \quad\quad | \ t_8 \\ \quad\quad | \ \tau \end{array}\right\} \equiv R3$$

L'enregistrement de cette grammaire peut être représenté par le schéma suivant :



C : 1 octet
S : 1 octet
A : 3 octets
F : 2 octets

3.1.3. Méthode d'analyse syntaxique appliquée.

L'analyse se fait, suivant le parcours de l'arbre de la grammaire, de haut en bas. Grâce au fait que la grammaire est LL(1) et que toutes les productions des règles de grammaire commencent par un symbole terminal (symboles nécessairement différents pour toutes les alternatives d'une règle), on est sûr que pour décider de la validité d'une production, il suffit d'enregarder le premier élément : l'analyse est déterministe. Tout symbole non-terminal qui exige donc le développement d'une règle de grammaire doit mener à la reconnaissance d'un symbole terminal en début d'une alternative de la règle, ou, éventuellement, au vide.

L'éditeur du compilateur - LE.1 lit un programme-source en LE.1 et en produit une chaîne codée, comprenant des unités syntaxiques des types identificateur, entier, chaîne, mot-clé. Ci-après nous donnons l'algorithme d'analyse utilisé.

Dénotons la chaîne codée $\Gamma = \gamma_1 \ \gamma_2 \ \ldots \gamma_q$

Avec γ_j soit identificateur,
soit entier,
soit chaîne,
soit mot-clé

et

$$G = \{N_o, T, N, \tau\} \equiv \{g_1, g_2, .., g_p\}$$

A chaque moment de l'analyse, dans le parcours de l'arbre qui correspond à la grammaire, nous avons à comparer un élément g_i de G avec un élément γ_j de Γ.

Deux cas peuvent se produire :

1. g_i est un symbole non-terminal ($g_i \in N$); dans ce cas, g_i est défini par une règle de grammaire, dont l'adresse du premier élément figure à côté du code de g_i; on saute à cette règle, et on remplace g_i par le premier élément de la règle (qui est nécessairement un élément terminal); on compare le nouveau g_i avec γ_j.

2. g_i est un symbole terminal ($g_i \in T$);

2.1. si $g_i \equiv \gamma_j$ (mot-clé identique, ou correspondance de type pour identificateur, entier ou chaîne) on progresse comme suit : on prend l'élément suivant γ_{j+1} de la chaîne codée et l'élément g_{i+1} de G, défini de la façon suivante :

- si g_i a un successeur, ce successeur est nommé g_{i+1}
- si g_i n'a pas de successeur, ce qui veut dire qu'un production complète de la règle de grammaire qu'on est en train d'explorer, vient d'être utilisée, on reprend l'analyse dans la règle explorée précédemment; on retourne au symbole non-terminal où l'on a quitté la règle en dernière instance; si cet élément a un successeur, ce successeur est nommé g_{i+1}; sinon, on recommence cette procédure, tant qu'il n'y ait pas de successeur.

2.2. Si $g_i \neq \gamma_j$, alors

2.2.1. si $g_i \equiv \tau$, celà veut dire que la règle de grammaire explorée résulte en une production vide; on reprend l'analyse dans la règle explorée précédemment, au point où on l'a quittée, c'est-à-dire au symbole non terminal;

- on regarde si ce symbole a un successeur et si oui ce successeur remplace g_i; on compare de nouveau g_i avec γ_j
- s'il n'y a pas de successeur, une production complète de la règle de grammaire où l'on se trouve vient d'être utilisée et on reprend l'analyse dans la règle explorée avant celle-ci; on retourne au symbole non-terminal où on l'a quittée en dernière instance; si cet élément a un successeur, celui-ci remplace g_i; sinon, on recommence cette procédure tant qu'il n'y ait pas de successeur; on compare le nouveau g_i avec γ_j.

2.2.2. si g_i a une alternative (qui est également un symbole terminal), cet élément remplace g_i; le nouveau g_i est comparé avec γ_j.

2.2.3. si g_i n'a pas d'alternative, il y a erreur syntaxique à l'élément γ_j dans la chaîne codée et donc dans le programme source en LE.1.

Prenons la grammaire de l'exemple du paragraphe précédent et examinons la chaîne codée t_2t_3. Nous indiquons schématiquement sur la représentation de la grammaire le parcours suivi à l'analyse de la chaîne.

Grammaire

N_0 ::= t_1 ① N_1

| t_2 ② N_2 ③ t_3 ⑦

N_1 ::= t_4

| t_5 t_6

N_2 ::= t_7 ④

| t_8 ⑤

| τ ⑥

Chaîne codée

$t_2 t_3$

La chaîne codée correspond à un programme correct.

3.1.4. Les appels aux fonctions sémantiques.

Une fonction sémantique peut entrainer plusieurs actions sémantiques. Chacune des étiquettes générées avec l'enregistrement de la grammaire est distincte et reflète l'endroit du symbole terminal correspondant dans la grammaire. Une étiquette comprend une partie index numérique qui est formée du numéro de la règle de la grammaire, du numéro de l'alternative dans laquelle le symbole apparaît et de son numéro d'ordre dans l'alternative.(La première production dans une règle est à considérer comme alternative 0 dans ce contexte). A l'étiquette se trouve détaillée la fonction sémantique. Le plus fréquemment elle se compose d'actions élémentaires ou d'appels à des fonctior entrainant des actions élémentaires, qui sont explicitées ailleurs parce qu'elles reviennent fréquemment. L'option que nous avons prise de n'introduire les appels qu'aux occurrences des symboles terminaux a pour but d'éviter au maximum de générer du code dans le cas où l'analyse détecte une erreur On remarque en effet que l'analyseur syntaxique descendant utilisé ici, ne fait progresser l'analyse de la chaîne codée qu'aux occurrences des symboles terminaux reconnus comme valables. L'option prise correspond aussi assez bier à la réalité de l'implémentation, puisque dans la pratique presque à chaque étiquette il y a des actions sémantiques à faire et à l'occurrence des symboles non-terminaux le besoin d'introduire des fonctions sémantiques ne s'est pas fait sentir.

Surtout, formellement, la grammaire est complètement indépendante de la sémantique de cette façon. Cela peut être très facile pour faire des petites modifications plus tard et dans la phase de mise au point, c'est très appréci ble.

3.1.5. L'encombrement de la grammaire en mémoire.

Des 350 symboles (environ) constituant la grammaire, nous comptons un tiers de symboles non-terminaux et deux tiers de symboles terminaux, dont une trentaine d'éléments nuls. Pour chaque symbole, il nous faut un octet pour son code, et 3 octets pour indiquer alternative et successeur. Simplifions les comptes en disant que 200 symboles terminaux entrainent la génération d'une étiquette de 2 octets pour des appels aux fonctions sémantiques. Notre grammaire requiert donc en mémoire 150 x 4 octets + 200 x 6 octets, soit 1800 octets, appels compris.

3.1.6. Modification proposée.

L'amélioration qu'on peut proposer dans cette méthode d'enregistrement de la grammaire, en vue d'appliquer un algorithme d'analyse plus performant, est inspiré par une méthode de Brooker, MacCallum , Morris et Rohl [BROO], [DUMA].

Reprenons notre exemple de grammaire, qui s'écrirait

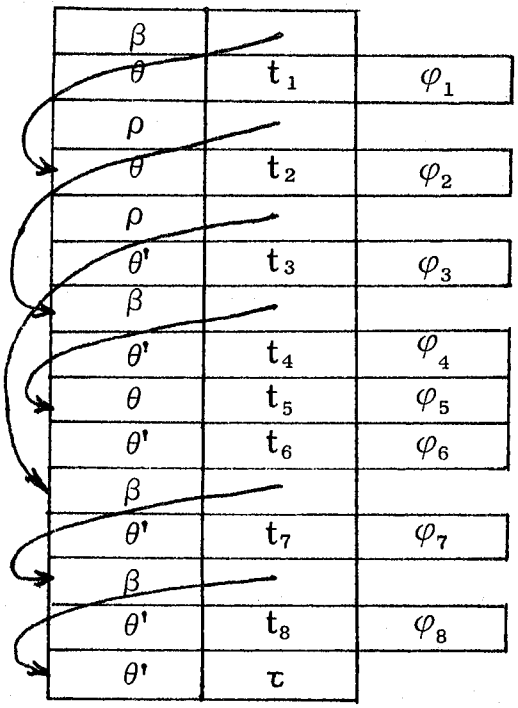
$$N_o \quad ::= \quad \overbrace{\beta\ t_1\ N_1\ .\ t_2}\ N_2\ t_3.$$

$$N_1 \quad ::= \quad \overbrace{\beta\ t_4\ .\ t_5}\ t_6.$$

$$N_2 \quad ::= \quad \overbrace{\beta\ t_7\ .}\ \overbrace{\beta\ t_8\ .}\ \tau.$$

où les β signifient la présence d'une alternative de la règle, en même temps qu'ils pointent dessus, et les points la fin d'une règle ou d'une alternative de règle.

On enregistre :



L'octet en tête de chaque bout de règle de grammaire est un descripteur du type du symbole qui suit.

On a β pour indiquer que suit une adresse relative, c'est-à-dire un déplacement exprimé en nombre d'octets par rapport à l'adresse du début de l'enregistrement de la grammaire. (En pratique ce déplacement est toujours inférieur à 255 octets).

On a θ et θ' pour indiquer que suit un symbole terminal, θ'_1 étant réservé aux symboles terminaux qui terminent une règle ou une alternative d'une règle de grammaire. Eliminant ainsi la codification explicite du point.

On a ρ pour indiquer l'occurrence d'une règle dans une définition. Le marqueur ρ doit être suivi d'un pointeur vers la règle correspondante de la grammaire. Ce pointeur peut être un déplacement relatif par rapport à l'origine de la grammaire et occupe en général deux octets (déplacement $\leqslant$ 64K). En distinguant des pointeurs en avant (ρ+) et en arrière (ρ-) on peut indiquer seulement le déplacement relatif par rapport au point d'appel. Comme pour les marqueurs θ il y a lieu de compacter l'information de fin de règle ou d'alternative dans ρ en adoptant les symboles ρ et ρ'.

Un troisième octet apparaît dans le cas des symboles terminaux θ et θ'(sauf pour l'élément nul, comme avant) et contient un code numérique qui est le numéro de la fonction de génération appelée à cet endroit.

La description faite à l'aide de l'exemple est valable jusqu'à concurrence de 256 symboles terminaux de grammaire, 256 fonctions de génération. En réalité, s'il y a besoin de travailler avec une grammaire plus importante, on définit des codes β et β_1.
En distinguant entre les déplacements $\leqslant$ 255 et ceux $\leqslant$ 64K par des marqueurs différents ρ (suivi d'un octet) et ρ_1 suivi de deux octets), on peut espérer un gain de place appréciable. Dans la pratique on aura en fait 8 marqueurs différents

β $\quad$ β_1
ρ $\quad$ ρ' $\quad$ ρ_1 $\quad$ ρ_1'
θ $\quad$ θ'

Une codification appropriée permet de les traiter comme argument d'une instruction 'CASE OF' d'Algol W.
Raisonnablement, on peut s'attendre à ce que la plupart des règles de grammaire aient leurs symboles constituants dans le même bloc et qu'il y ait donc une prépondérance d'adresses courtes.
Une estimation pour l'implémentation de notre grammaire de cette façon donne :

200 terminaux à 3 octets chacun,
150 non-terminaux dont 140 à 2 octets,
10 à 3 octets,
90 β dont 80 à 2 bytes et 10 à 3 octets,
soit au total environ 100 octets.

Quant aux facilités vis à vis de l'algorithme d'analyse, on peut citer que le premier octet permet de décider immédiatement du type du symbole disponible et au besoin de faire un branchement à une adresse qui est tout de suite sous la main. L'algorithme d'analyse associé à ce mode d'enregistrement d'une grammaire LL(1) est un fait plus simple que celui professé par SPG qui traite une classe plus large de grammaires. C'est cette méthode qui a été implémentée dans la version opérationnelle du compilateur écrite en Assembleur 360.

Rappelons que la méthode implémentée, avec le bit de successeur, nécessite l'extraction de ce bit, ce qui est une opération onéreuse en temps machine et qu'elle est assez fréquente.

3.2. La compilation.

Partant d'un programme-source LE.1 nous montrons comment sont analysées et générées des séquences à transférer aux et à exécuter par les différents ordinateurs concernés. Le prototype du compilateur que nous avons écrit transforme tout d'abord le programme-source en une chaîne codée, qui est reprise pour l'analyse et la génération, élément par élément. Finalement, ce procédé est moins avantageux que celui d'un éditeur qui dans son traitement du programme-source délivre à chaque fois qu'il a trouvé, une unité syntaxique, cet élément à l'analyseur. On évite ainsi des recherches inutiles dans des tables d'identificateurs, d'entiers...; surtout cela permet d'adapter plus facilement le compilateur pour travailler en mode intéractif.

Pour les séquences générées pour les différents ordinateurs, le problème de la synchronisation, se pose à l'exécution ainsi que le problème de reprise des programmes en cas d'interruption inattendue.

Les séquences générées se composent en principe de pas , avec des demandes de synchronisation en tête. Toujours l'un ou l'autre des ordinateurs qui participent à l'exécution du programme est actif et peut demander qu'un autre prenne la relève. Si une interruption se produit, la sauvegarde des résultats déjà obtenus se fait au niveau des pas et non au niveau du programme-réseau. Un pointeur de reprise est sauvegardé qui permet de redémarrer le programme au début du pas qui était en cours d'exécution.

Nous discutons ci-dessous l'implémentation de l'éditeur, de l'analyseur et la génération des différentes instructions.

Les règles de grammaire que nous y utilisons pour définir la syntaxe du langage appartiennent à la grammaire de représentation.

La grammaire d'implémentation (donnée en annexe), est dérivée de celle-ci en réunissant plusieurs règles afin de minimiser l'encombrement mémoire.

3.2.1. Analyse.

L'analyse se fait à trois niveaux: lexicographique, syntaxique et sémantique.

3.2.1.1. *Analyse lexicographique*

L'éditeur lit des images cartes en entrée et attribue aux unités syntaxiques un code correspondant à un des types suivants :

1. Identificateur : chaîne de 1 à n caractères alphanumériques plus le point.
2. Entier : chaîne de 1 à n caractères numériques
3. Chaîne : chaîne de n caractères entre apostrophes(une apostrophe faisant partie de la chaîne doit être représentée par deux apostrophes consécutives).
4. Mot-clé : opérateurs + - *
   séparateurs , := ( ) ; / '
   NETIN, CPU etc...

3.2.1.2. *Analyse syntaxique*.

L'analyse syntaxique est faite suivant l'algorithme décrit dans 3.1.3. Une pile est utilisée pour retenir l'adresse des éléments non-terminaux que l'on rencontre pendant le parcours; chaque adresse est retenue jusqu'à ce que l'élément non-terminal correspondant résulte en une production qui ne comprend que des éléments terminaux ou bien qui est vide; à ce moment on enlève l'adresse de la pile et on continue l'analyse.

L'analyse commence par la comparaison du premïer élément de la chaîne codée avec le premier élément enregistré de la grammaire, c'est-à-dire le premier élément de la première production de l'axiome, c'est-à-dire NETIN. L'analyse d'un programme correct laisse la pile vide à sa fin. Un élément suivant de la chaîne codée est amené par la procédure NEXTITEM. Un pointeur i pointe sur

l'élément de la grammaire à comparer; les adresses des éléments de la grammaire à retenir sont empilées dans ST. Nous notons dans l'organigramme suivant,

ST le contenu du sommet de la pile (j adresse du sommet; j:=j+1 ajout d'un élément à la pile; j:=j-1 enlèvement d'un élément de la pile).

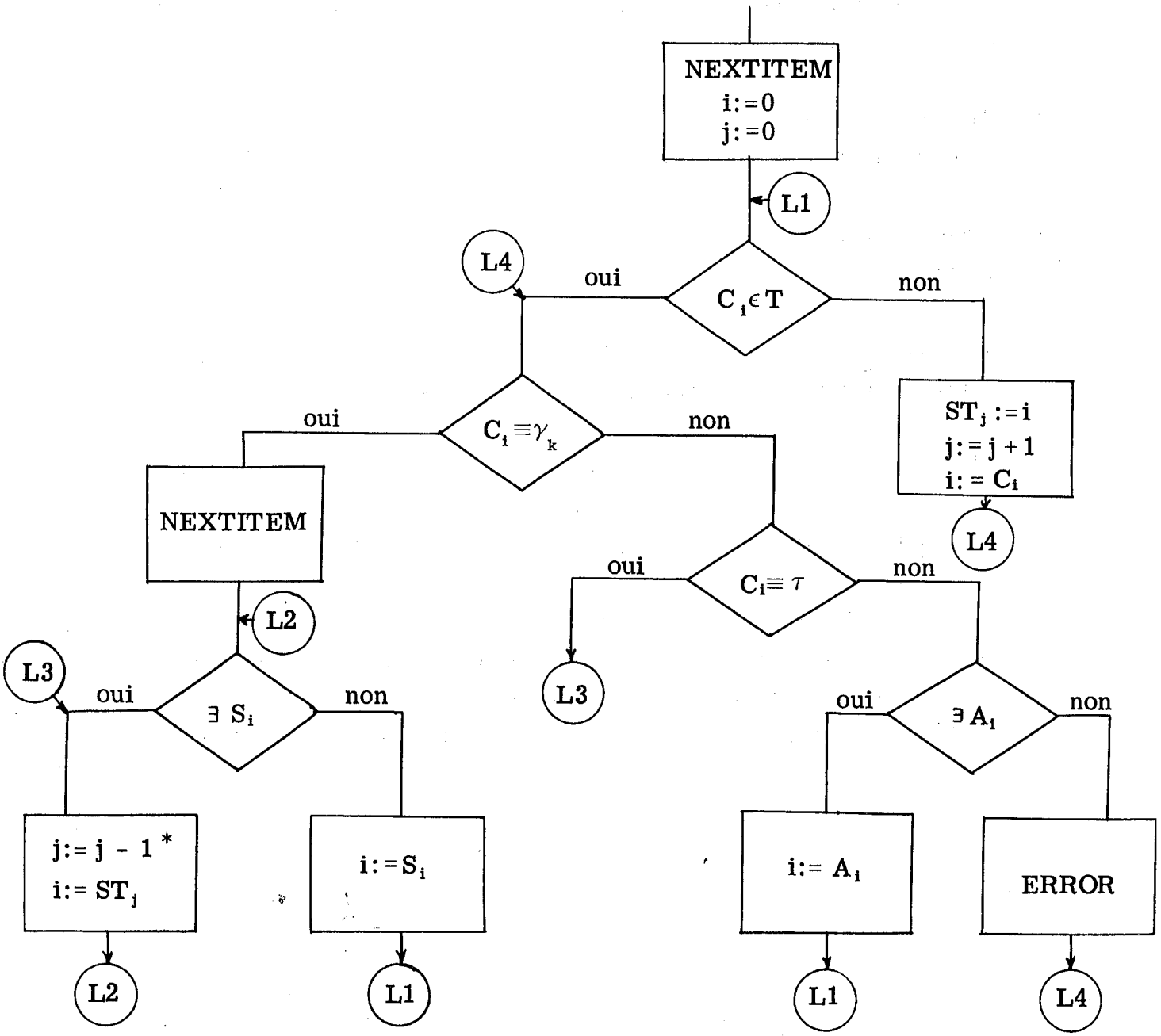
C pour le code numérique d'un symbole terminal ou non-terminal de la grammaire .

A l'adresse de l'alternative du terminal ou du premier élément de la première production du non-terminal, selon le cas.

S l'indication de la présence (=1) ou de l'absence (=0) d'un successeur.

C_i, S_i, A_i représentent les adresses des éléments C,S,A. Les erreurs syntaxiques, signalées, ne provoquent pas l'arrêt de l'analyse, mais la font reprendre à l'instruction LE.1 suivante (après le ;); la procédure ERROR sort le message approprié et met le pointeur dans la chaîne codée au début de la prochaine instruction (ou possiblement, NETOUT ou fin de chaîne codée) et le pointeur dans la grammaire en début de la liste d'instructions.

Organigramme de l'analyseur syntaxique :



Les éléments de la chaîne codée sont dénotés γ_k ;

C_i, S_i, A_i correspondent avec g_i dans la notation de 3.1.3.,

T avec la classe des terminaux, N avec la classe des non-terminaux.

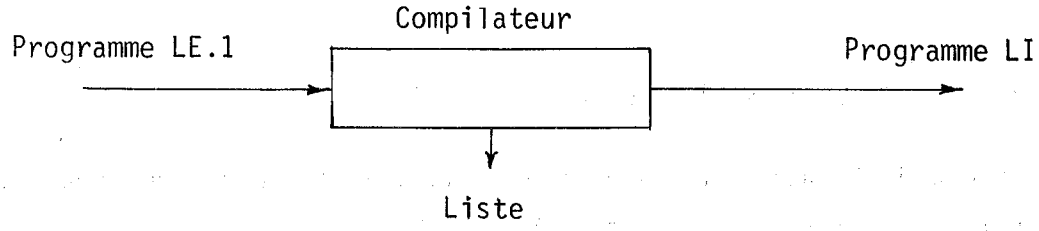
L'étoile indique qu'un contrôle est fait sur le débordement.

3.2.1.3. *Analyse sémantique.*

Parallèlement à l'analyse syntaxique la génération est faite, se basant sur le fait que chaque symbole reconnu comme valable le sera définitivement.
Les vérifications sémantiques qui précèdent à la génération peuvent aussi détecter des erreurs. L'analyse reprend à l'instruction LE.1 suivante (après le ;).

3.2.2. Génération.

Le compilateur traduit le langage externe LE.1 en langage interne LI et produit une liste de compilation.



Le compilateur fournit à l'interpréteur du LI sur chaque ordinateur participant des renseignements généraux, un bloc de zones de travail et un bloc d'instructions exécutables. Lui-même a besoin de ses propres zones de travail à la génération.
Une particularité du compilateur LE.1 est qu'une partie du code généré sert à lui-même en même temps comme zone de travail.

Pour éviter des confusions, si nécessaire, dorénavant nous parlerons de E-variables et E-instructions quand nous aurons à désigner des variables et des instructions au sens du LE.1 et de I-variables et de I-instructions pour désigner des variables et des instructions au sens du LI.

3.2.2.1. *Structure générale du programme généré.*

3.2.2.1.1. Les renseignements généraux.

Ils regroupent des informations pour le chargeur du LI(en cas d'initiation ou de reprise s'il y a redemarrage du système) et des I-variables de contrôle pour l'exécution. On y trouve :

1. La longueur totale du LI généré
2. Un pointeur vers la prochaine I-instruction à exécuter (initialement:la première; à l'exécution ce pointeur est constamment remis à jour).
3. La I-variable LEVEL, contenant le niveau de sévérité à respecter à l'exécution. (Si le code retour dépasse ce niveau, il y a erreur).
4. Un pointeur vers le point de reprise en cas d'interruption anormale.
5. Une I-variable destinée à contenir des données de contrôle lors des transmissions.

3.2.2.1.2. Le bloc des zones de travail LI.

Chaque déclaration d'une variable entraine l'ajout d'une zone associée à la variable. Il y a des zones CPU,... UNIT,...
Les variables de localisation sont en même temps des E-variables et des I-variables, tandis que les variables de spécification ne sont que des E-variables. Uniquement les zones créées pour les I-variables formeront les zones de travail LI et font partie du code généré passé à l'interpréteur. L'ensemble de toutes les zones créées fait partie des zones de travail LE.1 (les renseignements contenus dans les déclarations de spécification sont destinés à être copiés dans la I-variable DSN).

De façon générale on peut dire qu'une zone associée à une E-variable contient un code numérique qui indique le type de la variable, la valeur de la variable ou un pointeur vers elle, les pointeurs de chaînage nécessaires et éventuellement des renseignements complémentaires selon le type de la variable.

Au cours de la compilation, le compilateur construit et consulte une table, sous forme de liste qui contient les noms synonymes pour les variables et un pointeur vers la zone associée à ces variables. On l'appelle la liste des noms.

3.2.2.1.3. Le bloc des instructions exécutables LI.

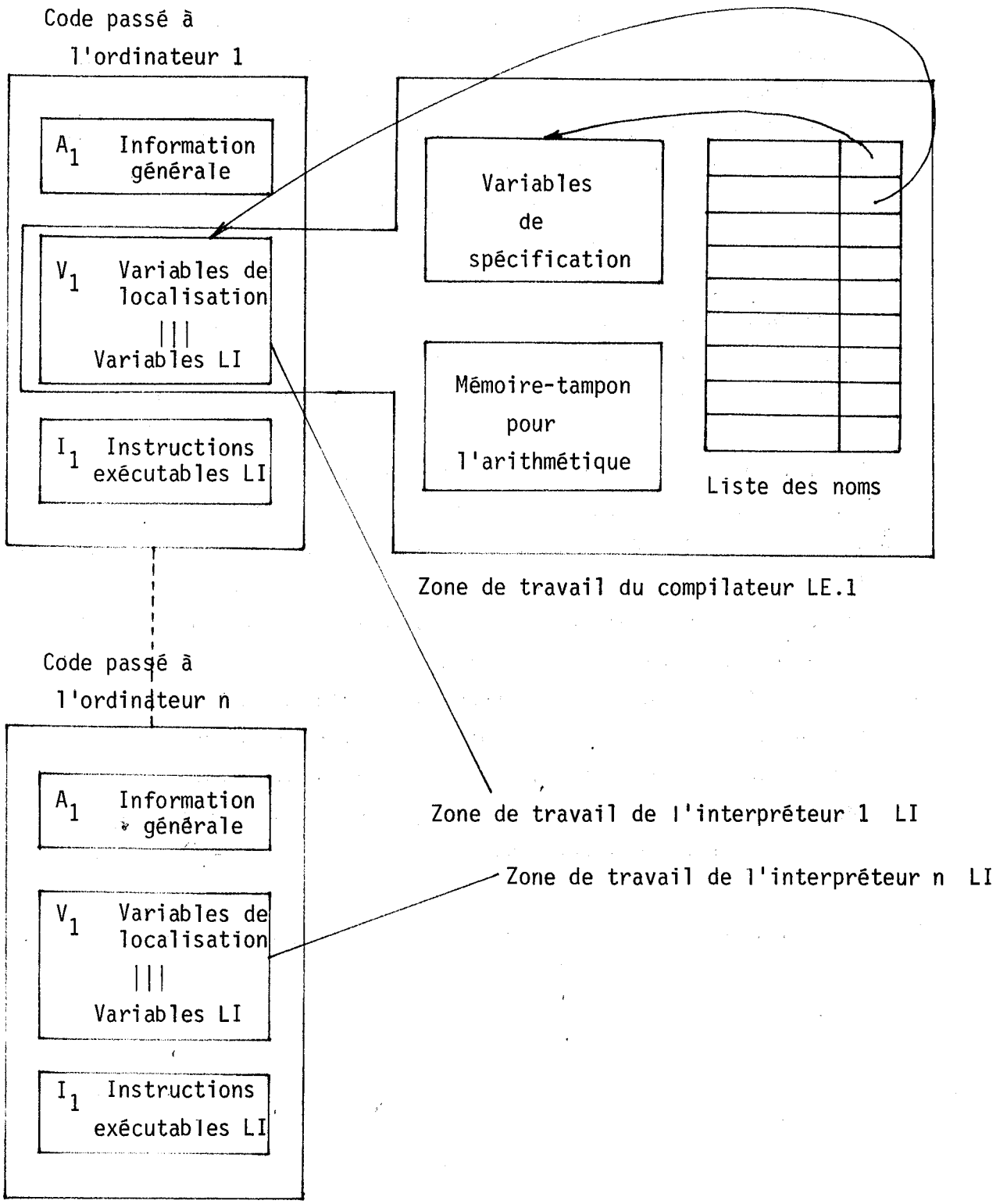
Chaque E-instruction exécutable entraîne la génération d'une ou de plusieurs I-instructions exécutables. Le format des I-instructions est de 2, 4, 6 ou 8 octets. Elles contiennent toujours en tête le code de l'instruction, ensuite la longueur de l'instruction et après une ou plusieurs valeurs ou pointeurs, selon l'instruction. Il y a des I-instructions correspondant directement aux fonctions exprimées par les E-instructions, il y en a d'autres qui servent à régler les transferts (ordres de transmission et de synchronisation) et il y en a relatives à la gestion de mémoire.

A chaque ordinateur du réseau qui aura à exécuter une partie du programme réseau, il sera adressé du code généré comprenant des renseignements généraux; un bloc de zones de travail et un bloc d'instructions exécutables.

La liaison entre les I-instructions exécutables et zones de travail LI est exprimée par les pointeurs se trouvant dans les I-instructions. Ils posent un problème à la génération. Dans la définition du LI, ce problème n'a pas été suffisamment abordé. En effet, les adresses sont actuellement données par un déplacement vis à vis du début des zones de travail LE.1. Comme les séquences de zones de travail LI ne sont qu'une sélection dans les zones de travail LE.1 le calcul d'adresses est à refaire ou bien on doit opter pour ne pas faire la sélection et recopier en entier les zones de travail LE.1 des E-variables de déclaration;(option pour le premier essai d'implémentation). Une redéfinition du système d'adressage est à envisager pour le LI. On pourrait s'inspirer de l'adressage par registre de base et déplacement en 360.

Le LI destiné à un ordinateur constituera un fichier spool, qui sera transmis.

3.2.2.1.4. Schéma de la génération.



3.2.2.2. *Génération des instructions déclaratives.*

La forme générale d'une partie déclarative d'une E-instruction est celle d'une affectation multiple au sens Algol 60 du terme :

$$n_1 := n_2 := .... := p ;$$

où les n_i sont des noms synonymes et p soit une valeur, soit une liste de valeurs entre parenthèses, soit un nom de variable déjà existante, dont le type est correspondant (1) à celui de la déclaration ou (2) à un autre type qui permet de trouver une valeur du type en question, par exemple

(1) DSORG A:=PS ; PS est une valeur pour la variable A de type DSORG.

(2) DSN B:= ... ;
DSORG A:=B ; affecte à la variable A la valeur du paramètre DSORG prise dans la partie spécifications de la I-variable B de type DSN.

Il y a deux aspects à la génération. En premier lieu, il y a la partie vérification qui décide de l'unicité des noms et de l'existence des variables. En second lieu, il y a la partie génération dans laquelle on génère la variable de la déclaration si elle n'existe pas encore et on complète éventuellement la liste des noms.

Pour procéder aux vérifications, le compilateur dispose d'un jeu de procédures logiques paramétrées qui permettent de détecter si un identificateur de variable examiné est un nom ou une valeur-chaîne déjà utilisé pour une variable de type correspondant. En plus, ces procédures délivrent en même temps le pointeur sur la zone associée à la variable si celle-ci existe, ou sur le premier emplacement libre en mémoire où l'on peut construire la nouvelle zone associée dans le cas contraire.

Remarque

En LE.1, les variables de types différents peuvent porter un même nom pour autant que l'on puisse distinguer le nom de ces variables par contexte. Dans le cadre de l'implémentation actuelle, les homonymes sont admis comme indiqué dans le premier tableau joint (case cochée correspond à validité). Une seconde version correspondra au second tableau.(La restriction apparente devra permettre un système de référençage plus puissant. Dans les instructions déclaratives de spécification notamment, non seulement une variable de même type que la déclaration, ou, des valeurs correspondantes dans la partie spécifications d'une variable de type DSN, pourront être référencées, mais, n'importe quel type de variable structurée qui dans la hiérarchie des variables est supérieure à la variable déclarée; dans ce cas il sera en références successives en opérant de haut en bas dans la hiérarchie).

	CPU	UNIT	VOLUME	VOLLIST	DSN	MBR	LMBR	INP	DSCB	SPACE	REC	RECFM	LRECL	BLKSIZE	DSORG
CPU															
UNIT															
VOLUME															
VOLLIST															
DSN															
MBR															
LMBR															
INP															
DSCB															
SPACE															
REC															
RECFM															
LRECL															
BLKSIZE															
DSORG															

Tableau 1: Implémentation actuelle

Exemple :

Une variable de type REC peut avoir le même nom qu'une variable de type CPU,UNIT,VOLUME,VOLLIST,DSCB,SPACE,RECFM,LRECL,BLKSIZE,DSORG.

	CPU	UNIT	VOLUME	VOLLIST	DSN	MBR	LMBR	INP	DSCB	SPACE	REC	RECFM	LRECL	BLKSIZE	DSORG
CPU		|	|	|	|	|	|	|	|	|	|	|	|	|	|
UNIT	|		|	|	|	|	|	|	|	|	|	|	|	|	|
VOLUME	|	|			|	|	|	|	|	|	|	|	|	|	|
VOLLIST	|	|			|	|	|	|	|	|	|	|	|	|	|
DSN	|	|	|	|											
MBR	|	|	|	|											
LMBR	|	|	|	|											
INP	|	|	|	|											
DSCB	|	|	|	|											
SPACE	|	|	|	|											
REC	|	|	|	|											
RECFM	|	|	|	|											
LRECL	|	|	|	|											
BLKSIZE	|	|	|	|											
DSORG	|	|	|	|											

Tableau 2: Projet d'implémentation

Exemple :

Une variable de type REC peut avoir le même nom qu'une variable de type CPU, UNIT, VOLUME, VOLLIST.

3.2.2.2.1. Génération des variables de localisation.

En règle générale et sauf pour la variable de type CPU pour laquelle les valeurs absolues et relatives coïncident, on ne peut commencer une déclaration par une partie déclarative dont la partie droite (p) est une valeur. Puisque cette valeur est forcément relative elle pourrait être interprétée d'une façon ambigüe.

3.2.2.2.1.1. Les séparateurs des parties déclaratives.

Il importe donc au compilateur au moment des tests de vérification, d'avoir l'information disponible qui indique que la partie déclarative en cours de

compilation débute une déclaration ou non. Une variable de travail est réservée à cet effet. Selon la valeur de cette variable,les résultats des tests peuvent être interprétés différemment . En plus à chaque moment il lui faut connaître le type de la variable qu'il est en train d'implémenter. Le mot-clé du début indique déjà le premier. Si d'autres variables suivent, séparées entre elles par des barres, au passage de la barre, le compilateur note qu'il a à faire à un nouveau type de variable dans la hiérarchie. Une variable de travail existe où on retient cela. Sauf par une barre, des parties déclaratives peuvent être encore séparées par des virgules. Une virgule signifie pour le compilateur qu'il y a une nouvelle déclaration et qu'elle débute comme l'indique le mot-clé en début de l'instruction. Le choix de cette signification pour la virgule mérite quelque réflexion. On peut en juger à l'aide de l'exemple suivant où l'on déclare trois fichiers de données DATA1, DATA2, et DATA3, les premiers se trouvant sur un même volume.

En LE.1 nous écrivons actuellement par exemple :

```
C1:=IMAG.67/U1:=2314/V1:=20MVT3/D1:=DATA1,
C1/U1/V1/D2:=DATA2,
SAC.91/2314/111111/D3:=DATA3;
```

ou le plus court possible

```
IMAG.67/2314/V1:=20MVT3/D1:=DATA1,
SAC.91/2314/111111/D3:=DATA3;
VOLUME V1/D2:=DATA2;
```

La virgule sépare ici deux déclarations de CPU. Il serait très concevable et peut-être cela s'avérera beaucoup plus pratique par la suite d'attacher à la virgule la signification du séparateur de deux variables de même type vis à vis de la barre précédente, en considérant tout ce qui précède cette barre comme partie déclarative commune à ces deux variables.
Ainsi nous écririons l'exemple précédent :

```
CPU IMAG.67/2314/20MVT3/D1:=DATA1,D2:=DATA2,
CPU SAC.91/2314/111111/D3:=DATA3;
```

A ce moment, probablement on ne pourra admettre ce style de parties déclaratives, que dans les parties d'extrême droite des déclarations, si l'on veut garder des significations sémantiques simples aux déclarations.

Nous remarquons encore que la syntaxe des déclarations permet d'écrire plusieurs barres consécutives. L'omission des UNIT et VOLUME permet de déclarer des fichiers catalogués. Le compilateur génère des blocs vides pour eux.

EXEMPLE :

```
CPU  IMAG.67///FICHIER;
```

3.2.2.2.1. 2. Les variables de type CPU (Unité centrale de traitement).

Syntaxe

CPU <DCLCPU><STDCLCPU> ;

<DCLCPU> ::= <ID> <IDCL> .....

<IDCL> ::= := <ID> <IDCL>
| <RIEN>

<STDCLCPU> ::= , <DCLCPU> <STDCLCPU>
| <RIEN>

Sémantique

Les variables de type CPU servent à référencer les ordinateurs appartenant au réseau.

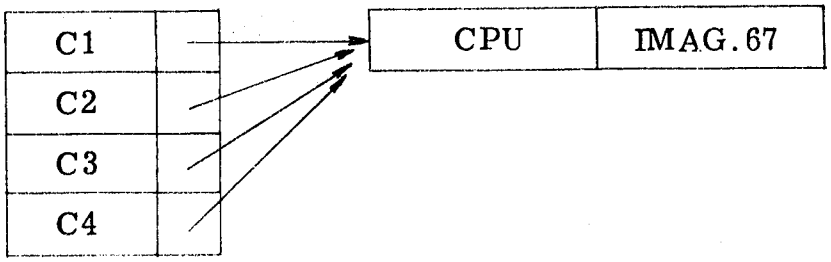
Les valeurs possibles des variables de type CPU existent sous forme de liste, passée au compilateur (IMAG.67, SAC.91 etc ...) Toutefois, ce n'est qu'à l'occasion d'une déclaration de CPU que la zone associée à la variable CPU en question sera créée.
Les écritures suivantes sont valables :

```
CPU  IMAG.67 ...                    (1)
CPU  C1:=C2:=IMAG.67 ...            (2)
CPU  C3:=C4:=C1 ...                 (3)
CPU  C1 ...                         (4)
```

Le compilateur fait les vérifications nécessaires sur l'unicité des noms synonymes et crée des entrées pour eux dans la liste des noms.
La partie de la forme générale est soit une valeur *(1),(2)* soit un non *(3) (4)* et le compilateur examine s'il n'y a pas déjà été construit une zone CPU avec la valeur ou la valeur référencée indiquée. Si p est un nouveau nom il y a erreur. Si nécessaire il construit un bloc associé CPU et remplit les pointeurs dans la liste des noms. Toutes les déclarations *(1) (2) (3) (4)* dans cet ordre peuvent exister ensemble dans un programme; mais dans l'ordre *(4) (1) (2) (3)* il y aurait erreur dès la première déclaration, parce qu'alors on référencierait une variable CPU qui n'existe pas. Ceci est une limitation imposée pour simplifier le compilateur, mais dont l'élimination ne pose pas de problème théorique.

Code généré

Le compilateur construit pour les déclarations de l'exemple une zone associée CPU et quatre entrées dans la liste des noms. Schématiquement, on peut représenter cela par :



3.2.2.2.1.3. Les variables de type UNIT (Unité périphérique).

Syntaxe

UNIT <DCLUNIT> <STDCLUNIT> ;

<DCLUNIT> ::= <ID> := <PARTUNIT> .....

<PARTUNIT> ::= := <VALUNIT>
| <RIEN>

<VALUNIT> ::= <ID> <PARTUNIT>
| <ENTIER>

<STDCLUNIT> ::= , <DCLUNIT> <STDCLUNIT>
| <RIEN>

Sémantique

Les variables de type UNIT servent à référencer les unités périphériques d'une unité centrale. Les valeurs usuelles des variables de type UNIT peuvent être :

- des numéros du catalogue du constructeur
  2400, 2415 pour des dérouleurs de bandes
  2311, 2314, 3330 pour des disques magnétiques
  2301 pour des tambours magnétiques

- des noms symboliques
  TAPE9, SYSSQ pour un dérouleur de bande 9 pistes
  DISK, SYSDA pour des disques
  DRUM, SYSDA pour des tambours

Le compilateur dispose d'une liste des valeurs UNIT possibles. A l'examen de la partie p il rencontre soit un nom, soit une valeur. Si le nom existe déjà pour une variable UNIT et qu'en conséquence il y a un pointeur sur une zone associée UNIT correspondante, il n'y a que le traitement des noms synonymes. Si le nom est nouveau, il y a erreur .
Si p est une valeur, valable, on ne regarde pas si une zone associée UNIT a déjà été construite pour cette valeur, d'office on en construit une nouvelle que l'on chaîne à la zone associée CPU correspondante.

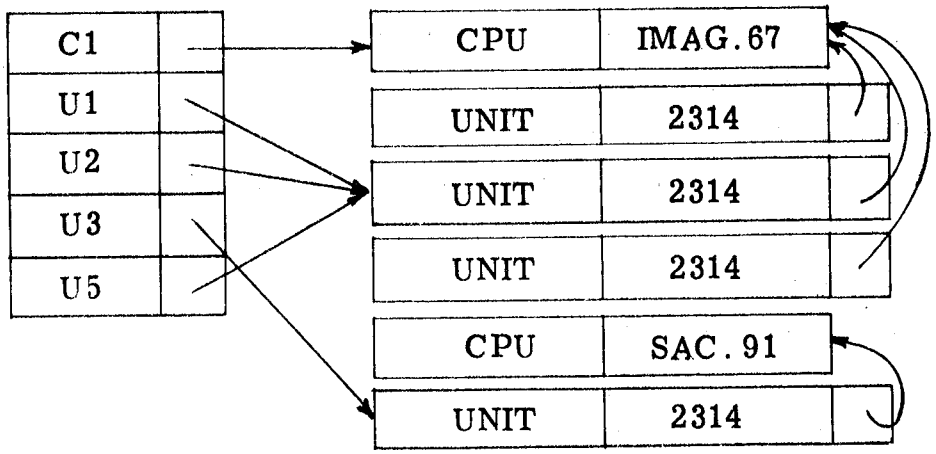
Prenons les exemples suivants :

```
CPU  C1:=IMAG.67/2314{...;   /...   ,...        (1)
CPU  C1/U1:=2314...                             (2)
CPU  C1/U1 ...                                  (3)
CPU  C1/U2:=U1 ...                              (4)
CPU  C1/ 2314  ...                              (5)
CPU  SAC.91/U3:=2314...                         (6)
CPU  C1/U4 ...                                  (7)
UNIT U5:=U1                                     (8)
```

A l'exception de l'exemple *(7)* près, toutes ces déclarations prises dans l'ordre sont valables.

Code généré

Le compilateur construit pour elles en ce qui concerne les zones associées UNIT et trois entrées dans la liste des noms, représentées schématiquement de la façon suivante :



Remarque

Des déclarations comme *(1)*, *(3)* et *(5)* prises en tant que telles n'ont pas d'intérêt pratique, mais peuvent être utiles si elles sont complétées par d'autres parties déclaratives, par exemple :

```
CPU  C1:=IMAG.67/2314/20MVT3/D1:=SYSLOAD1;        (1')
```

La création d'une nouvelle zone associée UNIT à chaque fois que l'on rencontre une valeur UNIT est motivée par des cas comme dans l'exemple *(1')* *(2')* *(6')*

```
CPU  C1/U1:=2314/20MVT4/D2:=SYSLOAD2;             (2')
CPU  SAC.91/U3:=2314/SYS001/D3:=LOADSET;          (6')
```

où les déclarations sont faites par exemple dans le but de faire une copie par l'instruction

```
COPY  (D1,D3)    TO    D2;
```

et où il importe à ce moment que les volumes 20MVT3 et 20MVT4 soient montés sur des unités différentes. C'est à l'interpréteur du langage interne de prendre en charge l'allocation d'unités en tenant compte de ces différences forcées par le langage externe.

3.2.2.2.1.4. Les variables de type VOLUME (Volume).

Syntaxe

VOLUME <DCLVOL> <STDCLVOL>;

<DCLVOL> ::= <ID> := <PARTVOL>.....

<PARTVOL> ::= := <LVOL>
| <RIEN>

<LVOL> ::= <ID> <PARTVOL>
| <ENTIER>
| (<LIDENT>)

<LIDENT> ::= <ID> <LID>
| <ENTIER> <LID>

<LID> ::= ,<ENTID> <LID>
| <RIEN>

<STDCLVOL> ::= ,<DCLVOL> <STDCLVOL>
| <RIEN>

<ENTID> ::= <ID>
| <ENTIER>

Sémantique

Les noms donnés aux volumes servent au système et aux opérateurs à distinguer les volumes des uns des autres. Les noms de volumes sont en tous points analogues au numéro matricule d'une automobile. Dans une installation donnée un nom de volume doit être unique. Il se peut cependant qu'au hasard des conventions de deux installations différentes un même nom désigne en fait deux volumes distincts.

Nous rappelons que les noms et les valeurs des variables de type VOLUME ne sont pas distinguables à priori. En outre, dans une seule partie déclarative on arrive à exprimer les VOLUME et les listes de VOLUME (symboliquement: VOLLIST).

Les vérifications sur les noms synonymes se font donc dans l'environnement de l'ensemble des noms des VOLUME et des noms des VOLLIST.

Faisons l'étude à l'aide des exemples suivants :

```
CPU  C1:=IMAG.67/U1:=2314/2OMVT2 ...        (1)
UNIT U1/V1:=2OMVT3...                       (2)
UNIT U1/V1...                               (3)
UNIT U1/(2OMVT4)...                         (4)
UNIT U1/W1:=(2OMVT5)...                     (5)
CPU C1/U2:=2314/V2:=IMAG8O...               (6)
CPU C1/2314/W2:=(V2,IMAG81)...              (7)
UNIT U2/V3:=V2 ...                          (8)
UNIT U1/W1...                               (9)
CPU SAC.91/2314/2OMVT2...                   (10)
```

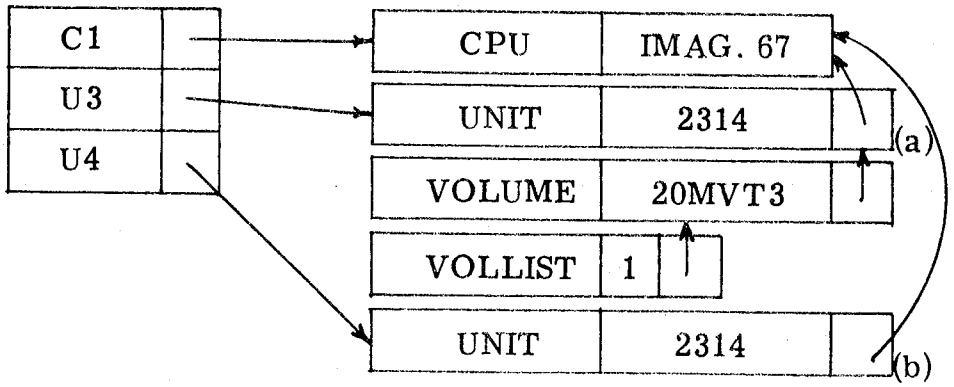
Nous supposons que les déclarations ont été faites dans l'ordre.
Syntaxiquement, il n'y a aucune différence entre *(1)*, *(3)* et *(9)* alors que dans *(1)* on a à faire à une valeur VOLUME, dans *(3)* à un nom de VOLUME, dans *(9)* à un nom de VOLLIST. Les valeurs VOLLIST par contre sont reconnaissables syntaxiquement, puisque une liste commence toujours par une parenthèse *(4)*.
Quand dans une partie p, on rencontre une chaîne pour la première fois, elle est considérée comme valeur de variable de type VOLUME *(1)*, *(2)*, *(4)*, *(6)*, *(7)*.
Dans la liste de VOLUME on peut rencontrer des valeurs VOLUME ou des noms VOLUME *(7)*, mais pas des noms de liste de VOLUME.
Chaque VOLUME déclaré seul correspond à une zone associée VOLUME et une zone associée VOLLIST construites par le compilateur.Le VOLLIST pointe sur le VOLUME. Un ou plusieurs VOLUME déclarés en liste de VOLUME correspondent à autant de zones associées VOLUME qu'il y a de volumes dans la liste et une seule zone associée VOLLIST qui contient un pointeur vers chaque VOLUME. La zone associée VOLLIST contient en outre le nombre de VOLUME sur laquelle elle pointe. Si un nom est associé au UNIT, il est réservé à la première des zones associées UNIT créées pour la liste. Pour décider de l'égalité de deux variables VOLUME on remonte le chaînage jusqu'au bloc CPU correspondant,

en contrôlant les valeurs et les noms UNIT (il faut que les valeurs soient les mêmes et qu'au plus un UNIT ait un nom) et en comparant les pointeurs sur le CPU des zones associées UNIT.
Dans les exemples suivants il y aurait ambiguïté et donc erreur :

```
CPU  C1:=IMAG.67/U3:=2314/20MVT3...        (11)
CPU  C1/U4:=2314/20MVT3...                 (12)
```

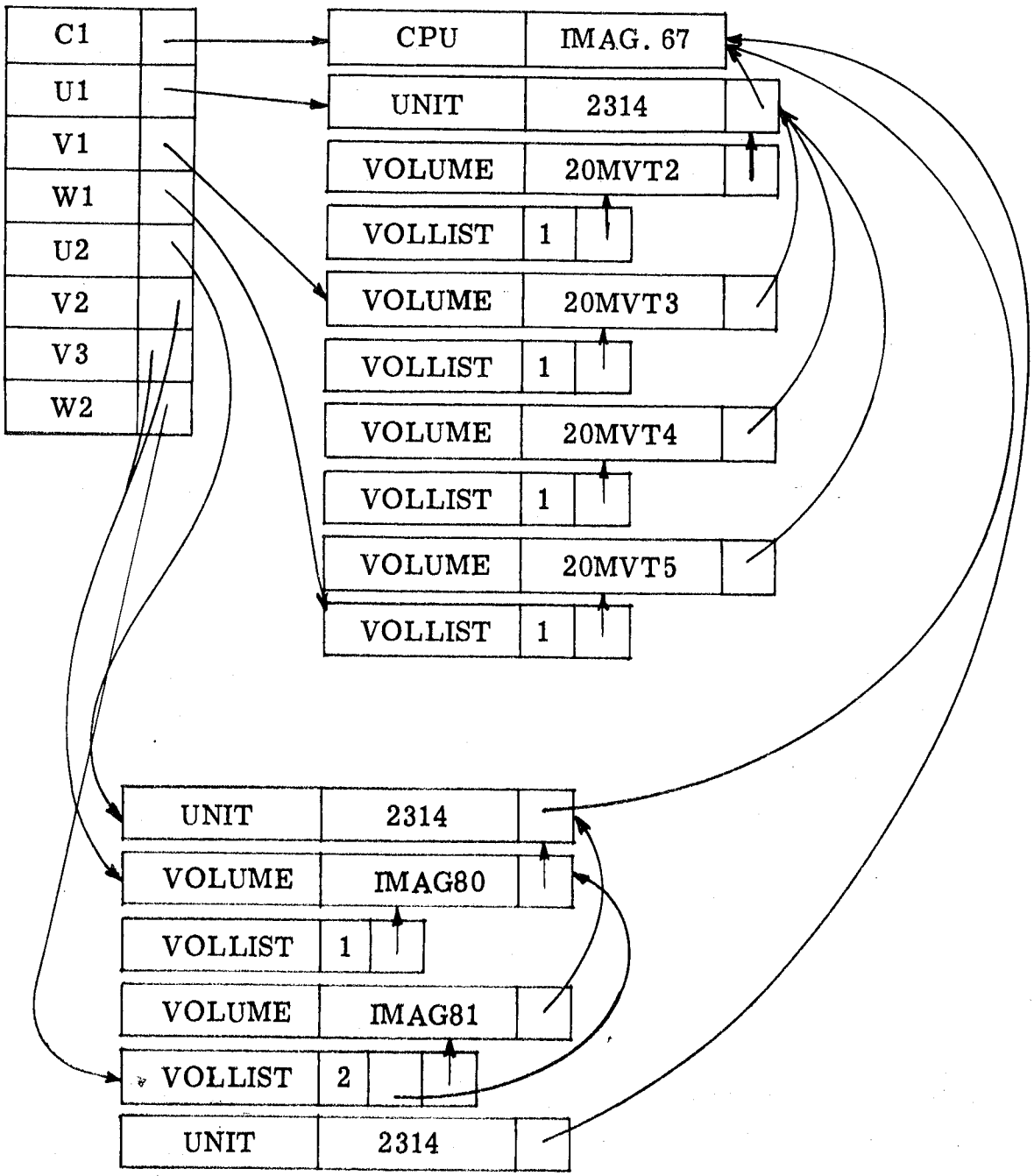
Code généré

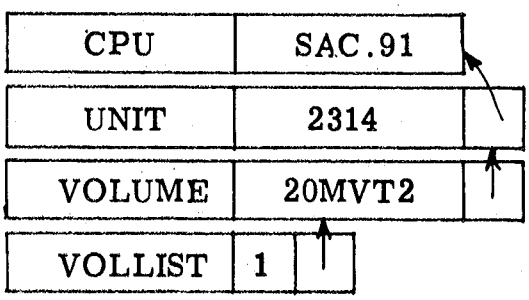
Le schéma montre l'organisation mémoire au moment de la deuxième utilisation de 20MVT3.



D'une part, 20MVT3 devrait être rattaché au bloc data UNIT (b) parce que c'est demandé explicitement dans la déclaration et d'autre part les contrôles sur le type d'unité et le pointeur vers CPU démontrent qu'il s'agit d'une même variable VOLUME. Comme il ne peut y avoir deux chaînes de zones associées pour une même variable, il y a erreur. Cependant, si U4 n'avait pas été associé à la zone (b), on aurait pû faire le chaînage avec la zone (a). La déclaration de la zone (b) peut à ce moment être considérée comme une maladresse du programmeur sans plus.
Pour décider de l'égalité de deux variables VOLLIST, un premier test est effectué sur le nombre de VOLUME dans la zone VOLLIST. S'il y a égalité on compare un à un les VOLUME pointés jusqu'à épuisement de la liste.

Dressons le schéma de l'organisation mémoire correspondant aux exemples *(1)* à *(10)*.





3.2.2.2.1.5. Les variables de types VOLLIST (Liste de volumes).

Syntaxe

```
                ::= VOLLIST <DCLVLL> <STDCLVLL> ;
<DCLVLL>        ::= <ID> <IDCL>.....
<STDCLVLL>      ::= ,<DCLVLL> <STDCLVLL>
                  | <RIEN>
```

Sémantique

La déclaration d'une variable de type VOLLIST peut être utilisée pour attacher un nouveau nom à une liste de volumes déjà déclarée.

EXEMPLE

Dans le contexte des exemples 3.2.2.2.1.4. on peut écrire :
VOLLIST WL := W2;

Code généré

Dans la liste des noms, le nom est ajouté et le pointeur correspondant rempli.

3.2.2.2.1. 6 Les variables de type DSN

Syntaxe

```
                    DSN <DCLDSN> <STDCLDSN> ;
<DCLDSN>        ::= <ID> <IDCL>.....
<STDCLDSN>      ::= , <DCLDSN> <STDCLDSN>
                  | <RIEN>
```

Sémantique

Un nom de fichier sert à identifier un fichier sur le (ou les) volume(s) sur le(s) quel(s) il est enregistré. Sur un volume donné, tous les noms de fichier doivent être différents. Certains noms de fichiers peuvent être listés dans un catalogue, en ce cas les noms doivent être unique dans la catalogue.

Dans la partie p chaque nouvelle chaîne est considérée comme une valeur. Sitôt qu'une chaîne commençant par un point apparaît, le CPU correspondant au **DSN** est cherché (chaînage de pointeurs connus) et dans la table des qualificateurs (voir instruction QUALIFY) on cherche le préfixe correspondant. S'il n'y en a pas, il y a erreur.

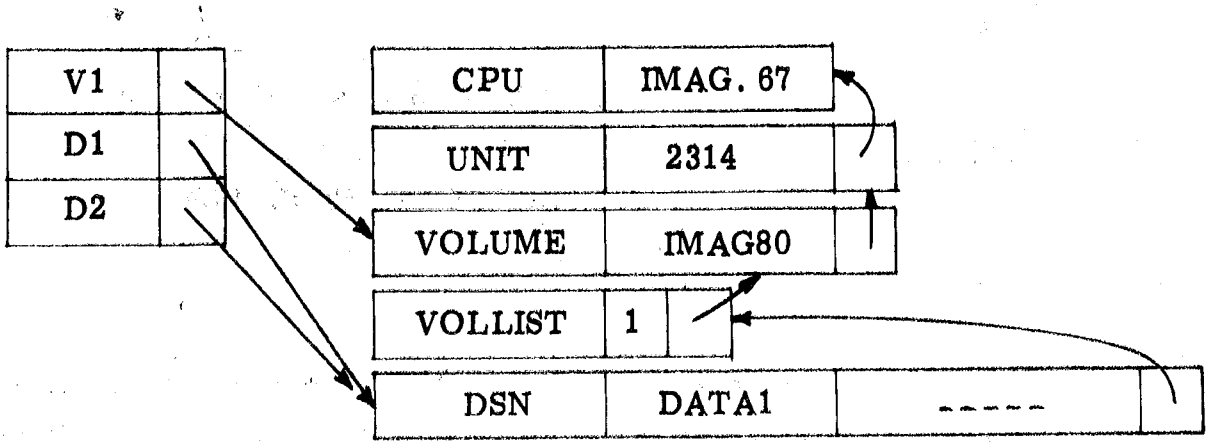
Les vérifications sur les noms synonymes sont triviales. Elles portent sur l'ensemble des noms de DSN, de MBR et de LMBR.
Si p est une valeur, elle est considérée comme une valeur DSN. Dans le cas où on trouve deux valeurs relatives DSN égales, l'examen des pointeurs de chaînage permet de décider si une zone associée a déjà été créée pour ce DSN.
Si p est un nom, ce nom doit obligatoirement avoir été associé à une zone DSN.

EXEMPLES :

```
CPU IMAG.67/3414/V1:=IMAG80/D1:=DATA1...        (1)
VOLUME V1/D2:=D1...                             (2)
```

Code généré

Schématiquement le code généré pour les exemples peut être représenté comme suit :



3.2.2.2.1.7. Les variables de type MBR (Membre).

Syntaxe

```
                    MBR <DCLMBR> <STDCLMBR> ;
<DCLMBR>        ::= <ID> <PARTMBR>
<PARTMBR>       ::= := <VALMBR>
<VALMBR>        ::= <ID> <PARTMBR>
                  | (<ID> <STID> )
<STID>          ::= , <ID> <STID>
                  | <RIEN>
<STDCLMBR>      ::= , <DCLMBR> <STDCLMBR>
                  | <RIEN>
```

Sémantique

Certains fichiers organisés en bibliothèque (organisation partitionnée) sont constitués d'un ensemble de fichiers séquentiels accessibles séparément. Chacun de ces fichiers séquentiels constitue un membre du fichier partitionné. Chaque nom de membre doit être unique dans un fichier partitionné.

L'unicité des noms est requise pour l'ensemble des noms, DSN, MBR, LMBR. La compilation est analogue à celle de la déclaration de VOLUME/VOLLIST Il faut cependant insister sur une différence; les éléments d'une liste de volumes sont ordonnés, ceux d'une liste de membres ne le sont pas (différents algorithmes de recherche).

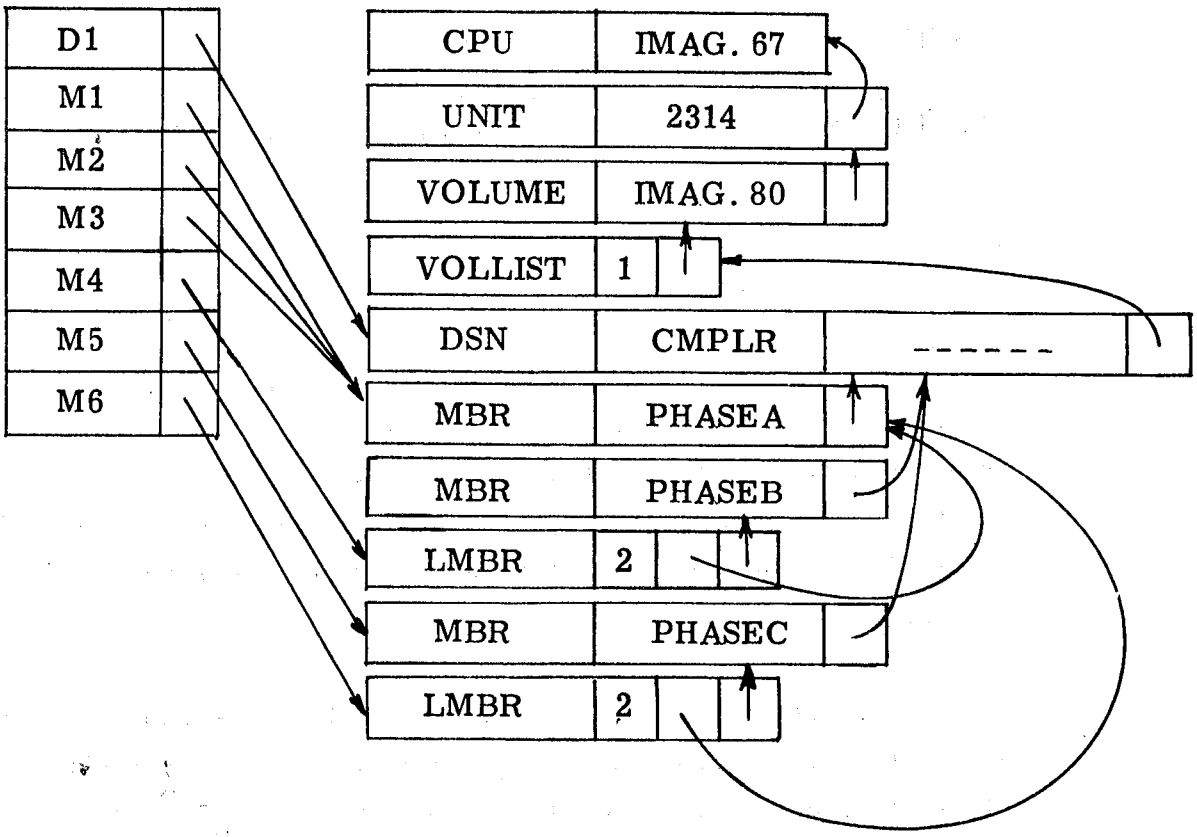
EXEMPLES :

```
CPU IMAG.67/2314/IMAG80/D1:=CMPLR/M1:=PHASEA;        (1)
DSN D1/M2:=M1 ;                                      (2)
DSN D1/PHASEB ;                                      (3)
MBR  M3:=M1 ;                                        (4)
```

```
DSN D1/M4:=(M1,PHASEB);                    (5)
DSN D1/M5:=PHASEC;                         (6)
MBR M6:=(M1,M5);                           (7)
```

Code généré

Schéma pour les exemples *(1)* à *(7)*



Notons que les noms dans les listes LMBR ne peuvent pas être des noms de LMBR, ni de DSN.

3.2.2.2.1.8. Les variables de type LMBR (Liste de membres).

Syntaxe

```
                    LMBR <DCLLMBR><STDCLLMBR>;
<DCLLMBR>      ::= <ID> := <ID><IDCL>
<STDCLLMBR>    ::= , <DCLLMBR><STDCLLMBR>
                 | <RIEN>
```

Sémantique

Dans la pratique on a souvent à manipuler une liste de membres d'un fichier partitionné. Pour simplifier l'écriture de listes de membre il a été créé dans LE.1 une variable de type liste de membres (LMBR).
La déclaration de LMBR sert à renommer des listes de membres. Elle s'impose surtout dans la grammaire par souci d'homogénéité.

EXEMPLE : Dans le contexte des exemples de 3.2.2.2.1.6.,

```
LMBR  M7.=M6;                                        (8)
```

Code généré

Une entrée dans la liste des noms est ajoutée.

3.2.2.2.1.9. Les travaux insérés dans le flot d'entrée.

Syntaxe

```
INP <ID> := '<TEXTE>'  <INP1> ;
```

suivi du travail inséré.

```
<INP1> ::= (<ENTIER> <ENTIER> <INP2>
         | <RIEN>
<INP2> ::= )
         | , <ENTIER>)
```

Sémantique

EXEMPLE :

```
INP   A:='DELIM'(1,5);
//....JOB
//

//
DELIM
```

Les deux entiers entre parenthèses indiquent :
1/ la colonne de cadrage de la chaîne dans une image de carte et
2/ la longueur effective de la chaîne servant de délimiteur d'images de cartes.

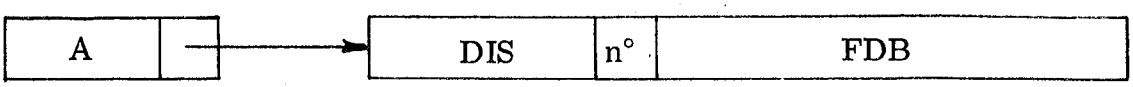
A noter que le ; est nécessaire pour permettre au compilateur de trouver simplement l'image de carte suivant l'instruction LE. Il serait souhaitable d'autoriser la référence à des fichiers OS/360 externes au Job LE, du genre

```
INP   <id>  :=  <dsname> ;
```

Enfin pour l'instant il n'est pas fait de relation entre les variables LE et celles apparaissant dans les images de cartes constituant le travail insér

Un fichier 'spool' est créé, contenant le travail inséré; il est caractérisé par la I-variable DIS.

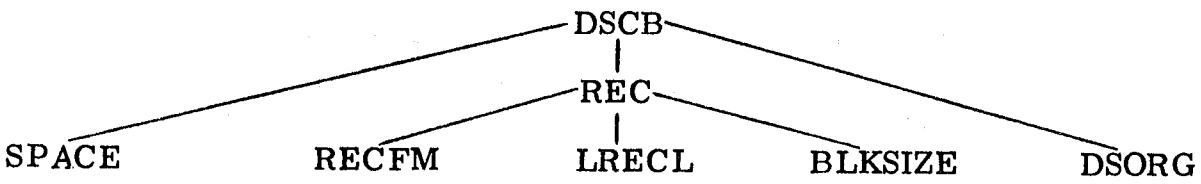
Code généré



Un octet est réservé à contenir le numéro de poste du fichier spool dans la NJCB ('network job control table' du système SOC).
FDB contient les spécifications du fichier; il n'est rempli que lorsque le fichier existe réellement.

3.2.2.2.2. Génération des variables de spécification.

Parmi ces variables qui sont uniquement des E-variables, il existe une imbrication, schématisée comme suit :



Les zones associées qui correspondent à ces variables ont des champs valeur ou pointeur (indiqué par un bit).(Les zones associées à
et à REC ont uniquement des champs pointeur).
Dans une déclaration, chaque partie déclarative a la forme :

$$n_1 := n_2 := .... := p$$

où les n_i sont des noms synonymes et p soit une valeur, soit un nom. Il faut au moins un n_i à la déclaration de plus haut niveau dans une partie déclarative de l'instruction. Si p est une valeur, une zone associée à la variable déclarée est créée. L'omission d'une valeur dans une déclaration entraîne l'affectation de la valeur par défaut implicite au compilateur, ou explicite si elle a été déclarée avant dans le programme-réseau.
Si p est un nom, il peut être soit le nom d'une variable existante, de même type que celle que l'on est en train de déclarer, soit le nom d'une variable de type DSN existante.

Il est à envisager d'étudier la validité des noms, en admettant que pour un p, le nom de n'importe quelle variable de plus haut niveau existante ou le nom de n'importe quelle variable de type DSN conviendrait.

Le compilateur a à chercher l'information correspondante dans la zone associée à la variable de même type, référencée, ou dans la partie spécification de la zone associée du DSN référence. En étendant la validité des noms, l'algorithme de recherche devient un peu plus complexe, puisqu'il y a des chaînages à parcourir. A chaque fois que le compilateur rencontre une valeur de l'un des types SRACE, RECFM, LRECL, BLKSIZE et DSORG, il crèe une zone associée appropriée de quelle façon que soit faite la déclaration autonome, ou à l'intérieur d'une déclaration de plus haut niveau s'il rencontre un nom dans la partie p, après les vérifications nécessaire il ajuste les pointeurs dans la zone associée à la variable en question.

La liste des noms est construite comme pour la déclaration des variables de localisation.

3.2.2.2.2.1. Les variables des types DSORG, RECFM, LRECL, BLKSIZE. (Organisation de fichiers, format d'enregistrement, longueur logique d'enregistrement, taille de bloc)

Ces quatre types sont similaires :

Syntaxe

RECFM <DCLRECFM><STDCLRECFM>;
<DCLRECFM> ::= <ID><PARTRECFM>
<PARTRECFM> ::= := <IDCLRM>
<IDCLRM> ::= <ID> <PARTRECFM>
| F
| FB
| V
| VB
| VS
| VBS
| U
<STDCLRECFM> ::= , <DCLRECFM><STDCLRECFM>
| <RIEN>

DSORG <DCLDSORG> <STDCLDSORG> ;

<DCLDSORG> ::= <ID> <PARTDSORG>

<PARTDSORG> ::= := <IDCLDS>

<IDCLDS> ::= <ID> <PARTDSORG>
| PS
| DA
| IS
| PO

<STDCLDSORG> ::= , <DCLDSORG> <STDCLDSORG>
| <RIEN>

LRECL <DCLLRECL> <STDCLLRECL> ;

<DCLLRECL> ::= <ID> <PARTUNIT>

<STDCLLRECL> ::= , <DCLLRECL> <STDCLLRECL>
| <RIEN>

BLKSIZE <DCLBLKSIZE> <STDCLBLKSIZE> ;

<DCLBLKSIZE> ::= <ID> <PARTUNIT>

<STDCLBLKSIZE> ::= , <DCLBLKSIZE> <STDCLBLKSIZE>
| <RIEN>

Sémantique

Rappelons que ces variables permettent de définir l'organisation d'un fichier (DSORG), son mode d'enregistrement (RECFM) et la taille des blocs physiques (BLKSIZE) et des enregistrements logiques (LRECL).

Les valeurs sont soit des chaînes réservées (RECFM, DSORG), soit des valeurs entier (LRECL, BLKSIZE).

EXEMPLES :

Nous supposons déclarées dans l'ordre :

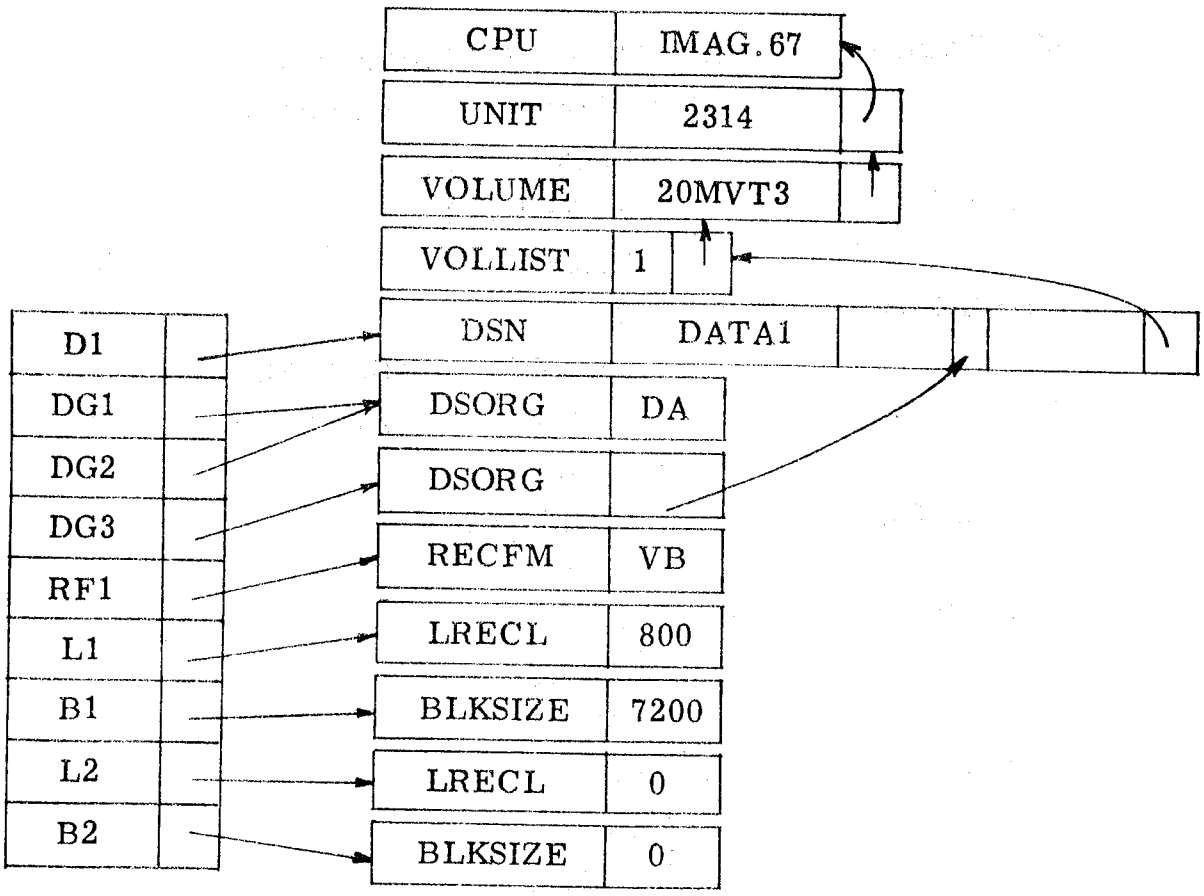
```
CPU IMAG.67/2314/20MVT3/D1:=DATA1;...          (1)
DSORG  DG1:=DA, DG2:=DG1, DG3:=D1;             (2)
RECFM  RF1:=VB;...                             (3)
LRECL L1:=800;                                 (4)
BLKSIZE B1:=7200;                              (5)
LRECL L2;  BLKSIZE B2;                         (6) (7)
```

Dans *(2)* on voit les trois façons possibles de déclarer une variable DSORG (en général RECFM, DSORG, LRECL ou BLKSIZE).
L'algorithme de recherche pour la partie p procède en s'interrogeant d'abord sur la valeur. S'il y a valeur, un bloc de données DSORG est créé. Ensuite, si p est un nom, il est d'abord regardé parmi les noms associés aux DSORG si c'est un nom connu, éventuellement on ajuste le pointeur du bloc de données DSORG, puis on regarde parmi les noms des DSN existants et si on trouve le nom correspondant, de même on ajuste le pointeur du bloc de données DSORG vers le champ DSORG dans l'extension du bloc de données DSN. Si le nom est nouveau il y a erreur. (Nous remarquons ici que les champs de la partie spécifications d'un DSN ne sont pas nécessairement remplis par leurs valeurs définitives à ce moment. Ils devront l'être à l'exécution, avant que l'on ne s'en serve).

Dans les exemples *(6) (7)* une zone associée à la variable LRECL est créée et une autre à la variable BLKSIZE. Leurs valeurs initiales sont zéro.

Code généré

Schéma pour les exemples :



3.2.2.2.2.2. Les variables de type SPACE (Espace)

Syntaxe

SPACE <DCLSPACE> <STDCLSPACE>;

<DCLSPACE> ::= <ID> <PARTSPACE>

<PARTSPACE> ::= := <LSPACE>

| <RIEN>

```
<LSPACE>      ::= <ID> <PARTSPACE>
                | ( <MESURE> <LSPACE1> )
                | ALLOC <ID>
<LSPACE1>     ::= , <LSPACE2>
                | <RIEN>
<LSPACE2>     ::= <ID>
                | ( <VENTID> <LSPACE3> )
<LSPACE3>     ::= , <VENTID> <LSPACE4>
                | <RIEN>
<LSPACE4>     ::= , <VENTID>
                | <RIEN>
<VENTID>      ::= <ID>
                | <ENTIER>
                | <RIEN>
<MESURE>      ::= CYL
                | TRK
                | <ENTIER>
<STDCLSPACE>  ::= , <DCLSPACE> <STDCLSPACE>
                | <RIEN>
```

Sémantique

Les variables de type SPACE (espace) servent à indiquer l'espace à allouer à un fichier. Cet espace s'exprime en diverses unités :

- pistes (TRK)
- cylindre (CYL)
- ou taille moyenne des blocs physiques

Une zone associée à une variable SPACE peut contenir quatre valeurs (la première étant chaîne ou entier, les autres des entiers) ou pointeurs.

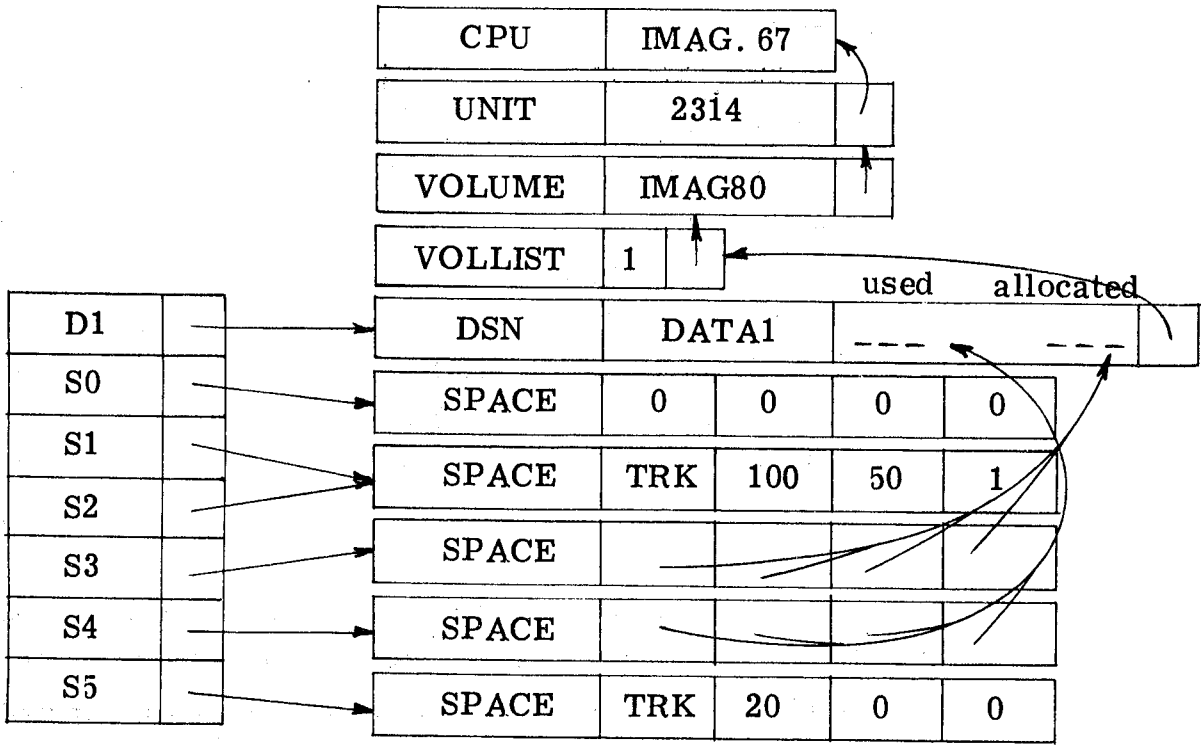
Les exemples suivants illustrent la génération.

```
CPU IMAG.67/2314/IMAG80/D1:=DATA1;                    (1)
SPACE S1;                                             (2)
SPACE S0:=(TRK,(100,50,1));                           (3)
SPACE S2:=S0;                                         (4)
SPACE S3:=D1;                                         (5)
SPACE S4:=ALLOC D1                                    (6)
SPACE S5:=(TRK,20)                                    (7)
SPACE S6:=(TRK,     )                                 (8)
SPACE S7:=(TRK,(20))                                  (9)
SPACE S8:=(TRK,(   ))                                 (10)
SPACE S9:=(TRK,(    ,10))                             (11)
SPACE S10:=(TRK,(   ,  ))                             (12)
SPACE S11:=(TRK,(   ,   ,2))                          (13)
SPACE S12:=(TRK,(    ,  ,   ,))                       (14)
```

Sans spécifications, les valeurs 'USED' (l'espace utilisé par un fichier de données) sont reprises, quand on référencie un DSN. Si 'ALLOC' est demandé ce sont les valeurs de l'espace alloué que l'on reprend, au sens IBM des termes.

Code généré

Schéma pour les exemples *(1) (2) (3) (4) (5) (6) (7)*



3.2.2.2.2.3. Les variables de type REC (Définition d'enregistrement).

Syntaxe

```
                        REC <DCLREC> <STDCLREC> ;
<DCLREC>        ::= <ID> <PARTREC>
<PARTREC>       ::= := <VALREC>
<VALREC>        ::= <ID> <PARTREC>
                  | ( <VIDCLRM> <BREC1> )
<BREC1>         ::= , <VIDCLV> <BREC2>
                  | <RIEN>
<BREC2>         ::= , <VIDCLV>
                  | <RIEN>
<STDCLREC>      ::= , <DCLREC> <STDCLREC>
                  | <RIEN>
<VIDCLRM>       ::= <ID> <PARTRECFM>
                  | <RIEN>
<VIDCLV>        ::= <ID> <PARTUNIT>
                  | <RIEN>
```

Sémantique

Les variables de type REC permettent de référencer un triplet :

- type d'organisation (DSORG)
- longueur logique d'enregistrement (LRECL)
- longueur des blocs physiques (BLKSIZE)

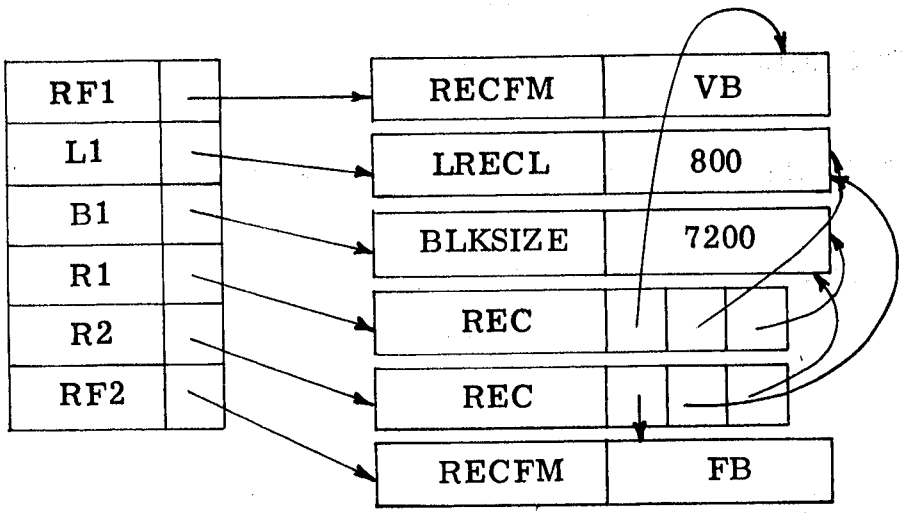
La zone associée à cette variable contient trois pointeurs, des valeurs de variables RECFM, LRECL, et BLKSIZE resp.

EXEMPLES

```
RECFM RF1:=VB;                          (1)
LRECL L1:=800;                          (2)
BLKSIZE B1:=7200;                       (3)
REC R1:=(RF1,L1,B1);                    (4)
REC R2:=(FR2:=FB,L1,B1);                (5)
```

Code généré

Schéma :



3.2.2.2.2.4. Les variables de type DSCB (Bloc de contrôle de fichier)

Ce type est similaire au type REC.

Syntaxe

DSCB <DCLDSCB> <STDCLDSCB> ;

<DCLDSCB> ::= <ID> <PARTDSCB>

<PARTDSCB> ::= := <VALDSCB>
| <RIEN>

<VALDSCB> ::= <ID> <PARTDSCB>
| (<VIDCLSP> <LDSCB1>)

<LDSCB1> ::= , <VIDCLREC> <LDSCB2>
| <RIEN>

<LDSCB2> ::= , <VIDCLDS>
| <RIEN>

<STDCLDSCB> ::= , <DCLDSCB> <STDCLDSCB>
| <RIEN>

<VIDCLSP> ::= <ID> <PARTSPACE>
| <RIEN>

<VIDCLREC> ::= <ID> <PARTREC>
| <RIEN>

<VIDCLDS> ::= <ID> <PARTDSORG>
| <RIEN>

Sémantique

Les variables de type DSCB permettent de référencer un triplet contenant :

- l'espace alloué à un fichier (SPACE)
- la définition d'enregistrement (REC)
- le type d'organisation (DSORG)

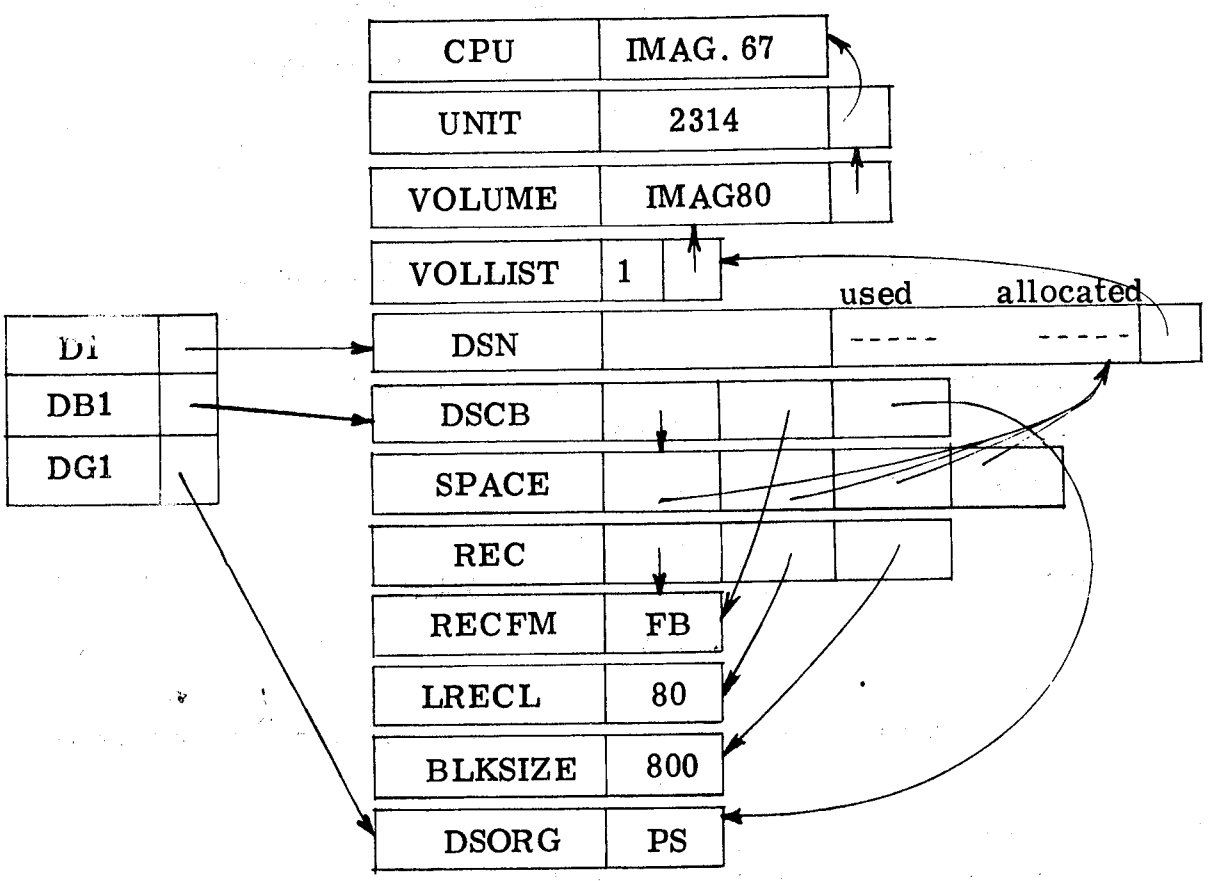
L'ensemble de ces informations est en général conservé dans le bloc descripteur d'un fichier et permet avec les informations de localisation la manipulation du contenu du fichier qui est alors considéré comme un ensemble de données ('DATA SET').

EXEMPLE :

```
CPU  IMAG.67/2314/IMAG80/D1:=DATA1;
DSCB DB1:=(ALLOC D1,(FB,80,800),DG1:=PS)
```

Code généré

Schéma :



3.2.2.2.3. Autres déclarations.

3.2.2.2.3.1. Instruction LEVEL.

Syntaxe

```
                    LEVEL <ENTIER> <OPTLEVEL> ;
<OPTLEVEL>      ::= , DUMP
                  | <RIEN>
```

Sémantique

La valeur entière de l'instruction LEVEL exprime un niveau de sévérité. Elle est mise dans les renseignements généraux. Si le LEVEL n'est pas spécifié, la valeur zéro est prise. Quant à l'exécution du programme une erreur à niveau de sévérité supérieur au niveau déclaré dans l'instruction LEVEL se produit, le programme se termine anormalement (l'équivalent en OS est ABEND). A la demande, une image post mortem est produite. La variable de sévérité peut être redéfinie dans un programme, mais ce n'est que la dernière définition qui sera prise en compte.

EXEMPLES :

```
LEVEL   8;                  (1)
LEVEL   8, DUMP;            (2)
```

(1) ne produira pas d'image mémoire en cas de terminaison anormale.

3.2.2.2.3.2. Instruction DEFAULT.

Syntaxe

```
                    DEFAULT <CHOIXDEF> <LCHOIXDEF>;
<CHOIXDEF>      ::= SPACE := (<MESURE> <LSPACE1>)
                  | LRECL := <ENTIER>
                  | BLKSIZE := <ENTIER>
<LCHOIXDEF>     ::= , <CHOIXDEF> <LCHOIXDEF>
                  | <RIEN>
```

Sémantique

Les valeurs spécifiées pour les SPACE, LRECL ou BLKSIZE sont des valeurs initiales attribuées à des variables de ce type et utilisées par la suite comme valeurs par défaut pour ces entités. Si rien n'est spécifié pour SPACE on prend les valeurs 800,100,50,2, pour LRECL 80 et pour BLKSIZE 800. Les valeurs par défaut peuvent être redéfinies dans un programme-réseau. Comme ce sont des valeurs attribuées au moment de la compilation, au fur et à mesure des définitions ou des redéfinitions le compilateur les prend en compte.

EXEMPLES :

```
DEFAULT  LRECL:=1600;                    (1)
DEFAULT  BLKSIZE:=7200;                  (2)
DEFAULT  SPACE:=(800,(200,50,1));        (3)
DEFAULT  SPACE:=(TRK,100);               (4)
```

Dans le bloc SPACE créé à l'initialisation du programme, le compilateur remplace les valeurs par 800, 200, 50, 1 dans l'exemple (3). Dans l'exemple (4), les valeurs après la compilation de l'instruction seront TRK, 100, 50, 2.

3.2.2.2.3.3. Instruction QUALIFY

Syntaxe

QUALIFY <ID> **BY** <ID> ;

Ordinateur — Qualificateur

Sémantique

Au fur et à mesure que des variables CPU sont déclarées, le compilateur construit une liste dans la zone de travail, en ajoutant un élément à la liste à chaque déclaration. Chaque élément de la liste contient une chaîne de blancs au départ. Par une instruction QUALIFY, on vient remplir l'élément de la liste correspondant à la variable CPU mentionnée, par la valeur chaîne

donnée. La chaîne ne peut commencer par un point, ni se terminer par un point, ni contenir des points qui se suivent. Un contrôle est effectué. (La longueur de la chaîne d'une variable DSN qui indique le nom physique d'un fichier de données étant d'au maximum 44 caractères pour les fichiers disque -et 17 caractères pour les fichiers bandes- les préfixes déclarés par une instruction QUALIFY ne peuvent dépasser 42 caractères). Au moment où le compilateur rencontre une instruction QUALIFY qui correspond à une variable CPU non encore déclarée, il fera d'abord la génération du bloc CPU approprié.

EXEMPLE :

```
QUALIFY  IMAG.67  BY  S5161.PO580.DECALUWE ;
```

3.2.2.3. *Les instructions exécutables.*

On suppose que les déclarations suivantes sont faites une fois pour toutes, pour la discussion des instructions exécutables en détail.

```
        CPU C1:=IMAG.67/2314/V1:=IMAG.80/D1:=DPGM;
        CPU C2:=SAC.91/2314/V4:=DISK4/D4:=SPGM;
        CPU C1/U1:=2400/V2:=000238, C1/U1/V3:=000239;
        VOLUME V2/D2:=BPGM2, V3/D3:=BPGM3;
        DSN D4/M4:=MBR1, D1/M5:=(MBR2);
        LRECL  L1:=80, L2:=120, LV1;
        BLKSIZE B1:=1600, B2:=800, BV1;
        SPACE S1:=(800,(200,20,0)),
              S2:=(400,(600,300,0)),
              SV1;
        DSCB1, DSCB1:=(S1,(FB,L1,B1),PS);
INP     INP1:='DELIMITEUR'(1,3);
//TRAVAIL JOB(...
//     DD ...
//     EXEC ...
/*
//
DEL
```

Dans la représentation des schémas suivants, les flèches représentant des pointeurs sont remplacées par des petites flèches verticales suivies du nom correspondant à la variable que l'on désigne.

3.2.2.3.1. L'instruction MOUNT (monter un volume)

Syntaxe

```
                    MOUNT <ID> <SPECBANDE> ;
<SPECBANDE>    ::= ( <OPTRING> , <OPTLABEL> )
                 | <RIEN>
<OPTRING>      ::= RING
                 | NORING
                 | <RIEN>
<OPTLABEL>     ::= SL
                 | NL
                 | BLP
                 | <RIEN>
```

On notera que cette syntaxe impose un ordre aux paramètres

```
MOUNT  VOL1 (RING,SL);
```

alors que :

```
MOUNT  VOL1 (SL,RING);
```

serait strictement équivalent car les paramètres sont des symboles terminaux.

Sémantique

Par cette instruction on ordonne le montage d'un volume (disque, bande). Dans le cas d'une bande on donne les spécifications nécessaires pour le montage, si on le veut en lecture-écriture (RING), en protection

d'écriture (NORING) -,et les spécifications du mode d'utilisation de la bande- sans étiquette (NL), avec étiquette standard (SL) ou en ignorant l'étiquette (BLP). Ces spécifications viennent prendre place dans la variable VOLUME mentionnée, puisqu'elles sont en fait plutôt associées au volume qu'au montage. Si on demande le montage d'une bande en indiquant les spécifications par défaut, NORING et SL sont pris en compte. Si par contre, on demande le montage d'une bande en omettant les spécifications, ce seront celles du montage précédent qui sont prises en compte (ou celles de l'initialisation s'il n'y a pas eu de montage antérieurement).

EXEMPLES :

On suppose les déclarations préalables *(1)* et *(2)* pour le disque IMAG 80 et les bandes 000238 et 000239 sur l'ordinateur IMAG.67.

```
CPU C1:=IMAG.67/2314/V1:=IMAG.80;                    (1)
CPU C1/U1:=2400/V2:=000238,C1/V1/V3:=000239;         (2)
        MOUNT  V1;                                   (3)
        MOUNT  V2(RING, SL);                         (4)
        MOUNT  V3(     ,BLP);                        (5)
        MOUNT  V3;                                   (6)
```

Dans *(3)* il n'y a pas de spécifications, puisqu'il s'agit d'un disque. Dans *(4)* on demande expressement le montage de la bande avec anneau et en mode d'utilisation étiquette standard. Dans *(5)* l'omission du premier paramètre exprime que l'on désire le montage en option standard, sans anneau; par le deuxième paramètre on exprime qu'on traitera la bande en ignorant l'étiquette. Dans *(6)* les options qui sont prises en compte sont NORING et BLP conformément au montage précédent.

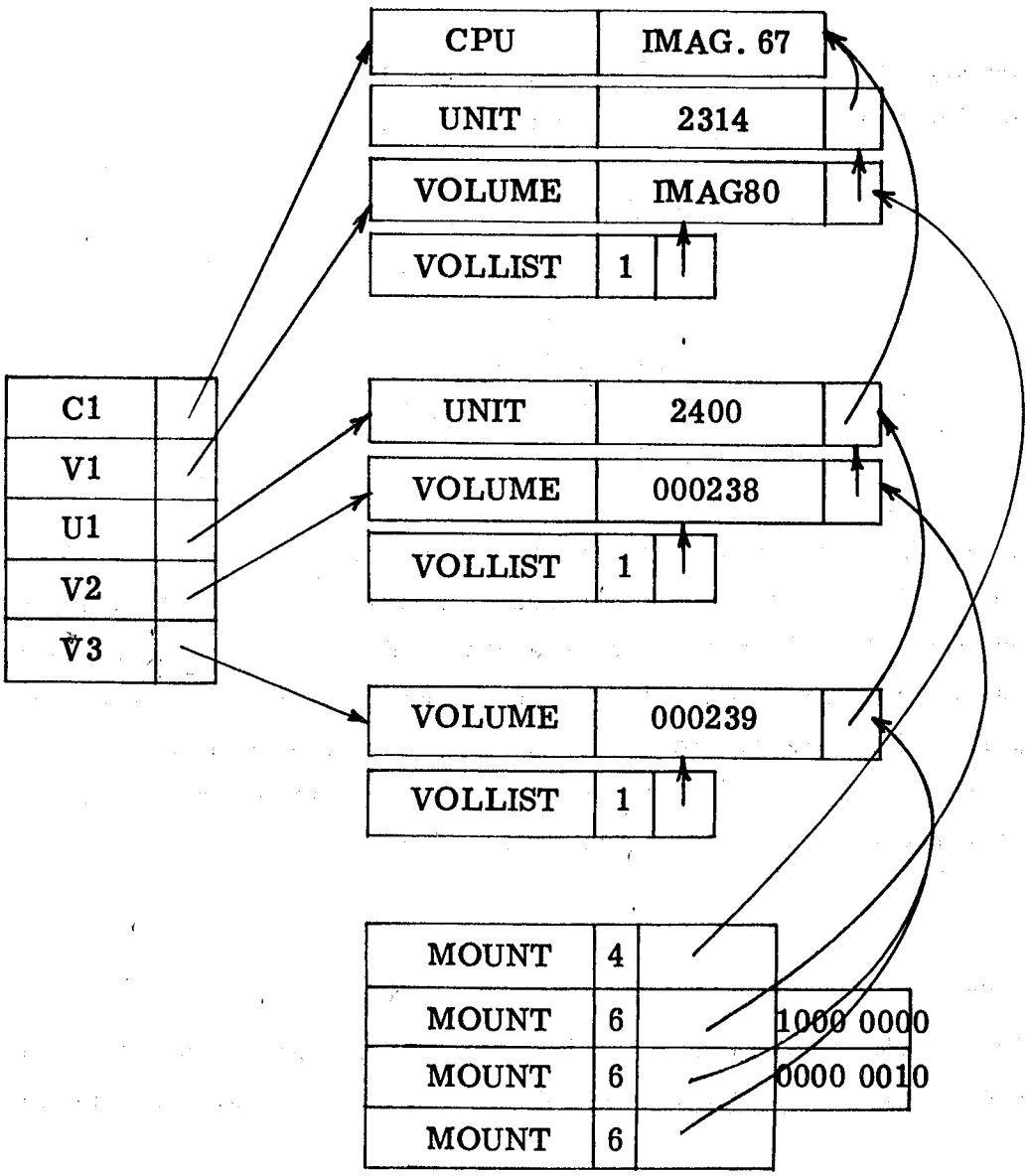
Code généré

Selon qu'il y ait des spécifications de montage ou non, le compilateur génère une I-instruction MOUNT de 6 ou de 4 octets, qui contient un pointeur

vers la zone associée au VOLUME mentionné et éventuellement exprimer les spécifications

Bit 0	0	pour NORING
	1	pour RING
Bits 6-7	00	pour SL
	01	pour NL
	10	pour BLP

Schéma pour les exemples :



3.2.2.3.2.L'instruction DISMOUNT (démonter un volume)

Syntaxe

DISMOUNT <ID> ;

Sémantique

Par cette instruction on ordonne le démontage du volume indiqué. On ne fait plus de distinction entre disques et bandes dans l'écriture de l'instruction.

EXEMPLES (en supposant les déclarations *(1)*, *(2)* de

```
DISMOUNT    V1;          (1)
DISMOUNT    V2;          (2)
```

Code généré

Le compilateur génère une I-instruction DISMOUNT (de 4 octets) qui contient un pointeur vers le bloc VOLUME mentionné.

Schéma pour les exemples :

DISMOUNT	**4**	**↑V1**
DISMOUNT	**4**	**↑V2**

3.2.2.3.3. L'instruction CATALG (Cataloguer un fichier)

Syntaxe

CATALG <ID> <OPTCATALG> ;

<OPTCATALG> ::= ON <ID> (Fichier / Volume)
| <RIEN>

Sémantique

Par cette instruction on ordonne le cataloguage d'un fichier de données sur le volume de contrôle indiqué, ou à défaut d'indication, sur le volume de contrôle standard sur l'ordinateur.

EXEMPLES :

```
CATALG   D1   ON   V1;          (1)
CATALG   D2   ON   V1;          (2)
CATALG   D3;
```

Cataloguage d'un fichier disque ou bande sur le volume de contrôle V1 (dans les exemples (1) et (2)) et sur le volume de contrôle standard (exemple (3)).

Code généré

L'I-instruction CTLG qui correspond à l'E-instruction CATALG est de 6 ou de 4 octets selon qu'un volume de contrôle est indiqué ou non. Elle contient un pointeur vers la zone associée au DSN mentionné et éventuellement un pointeur vers la zone associée au volume indiqué.

Schéma pour les exemples

CTLG	6	↑D1	↑V1
CTLG	6	↑D2	↑V1
CTLG	4	↑D3	

3.2.2.3.4. L'instruction UNCTLG (décataloguer un fichier)

Syntaxe

UNCTLG <ID> <OPTCATALG> ;

Fichier

Sémantique

Cette instruction est complètement analogue à l'instruction CATALG, servant au décataloguage des fichiers de données.

EXEMPLES :

```
UNCTLG  D1  ON  V1;
UNCTLG  D2  ON  V1;
UNCTLG  D3;
```

Code généré

Il y a analogie complète avec la génération de l'instruction CATALG.
Une I-instruction UNCTLG est générée.

Schéma pour les exemples :

UNCTLG	6	↑D1	↑V1
UNCTLG	6	↑D2	↑V1
UNCTLG	4	↑D3	

3.2.2.3.5. L'instruction EMPTY (vider un fichier)

Syntaxe

EMPTY <ID> ;

Fichier

Sémantique

L'instruction EMPTY, inspirée sur une fonction analogue en MTS, a pour fonction de vider un fichier de son contenu, c'est-à-dire de substituer aux valeurs de l'espace utilisé des valeurs zéro, dans le cas du fichier disque. Pour une bande, cette fonction ne peut avoir de sens dans la mesure que le fichier mentionné est le dernier enregistré sur la bande, car sur une bande magnétique, il n'est pas possible d'écrire un enregistrement nouveau entre deux anciens. Bien entendu, les fichiers catalogués gardent leur entrée dans le catalogue.

EXEMPLES :

```
EMPTY  D1;
EMPTY  D2;
```

Code généré

Deux I-instructions sont générées :

- Une I-instruction MOUNT pour demander le montage du volume sur lequel se trouve le fichier indiqué et
- une I-instruction EMPTY, de longueur 4 octets, qui contient un pointeur vers la zone associée au DSN indiqué.

Schémas pour les exemples :

MOUNT	**4**	**↑V1**
EMPTY	**4**	**↑D1**

3.2.2.3.6. L'instruction DELETE (abandonner)

Syntaxe

DELETE <ID> ;

Fichier

Sémantique

L'instruction DELETE est de forme identique à l'instruction EMPTY. Sa fonction est de détruire le fichier de données indiqué et s'il est catalogué, de le décataloguer, par opposition à EMPTY qui conserve l'espace alloué et fait perdre le contenu. Ici l'espace alloué est restitué.

EXEMPLES

```
DELETE  D1;
DELETE  D2;
```

Code généré

Comme pour l'instruction EMPTY deux I-instructions sont générées, un MOUNT et un DELETE, analogue au EMPTY.
L'I-instruction DELETE est de longueur 4 octets et contient un pointeur vers un DSN.

Schéma pour les exemples :

MOUNT	4	↑V1
DELETE	4	↑D1

3.2.2.3.7. L'instruction CREATE

Syntaxe

CREATE <ID> (<ID>) ;
Fichier Spécifications

Sémantique

Cette instruction sert à créer un fichier de données avec les spécifications mentionnées. Les spécifications apparaissent dans l'instruction entre parenthèses, sous un nom de DSCB (fichier disque), de REC(fichier bande) ou de DSN (référence à fichier de même type que celui qu'on crée).
C'est dans l'instruction CREATE que peuvent donc vraiment servir les valeurs par défaut qu'on s'est défini. L'utilisation se fait de façon indirecte, par l'intermédiaire de l'une des variables DSCB, REC ou DSN. C'est le seul point d'attache entre les variables de localisation et les variables de spécification du programme-réseau. A l'instruction CREATE, le compilateur fait le nécessaire pour remplir ou faire remplir à l'exécution la partie spécifications de la zone associée à une variable DSN. Trois cas peuvent se produire :

1. Dans les zones de travail, le compilateur a toutes les valeurs qu'il faut pour remplir la partie spécifications de la variable DSN. A ce moment, il remplit, génère un I-MOUNT pour le volume sur lequel doit se trouver le fichier et un I-CREATE pour le fichier. L'I-instruction CREATE est une instruction à 4 octets qui contient un pointeur vers la zone associée à la variable DSN du fichier a créer.

2. Le compilateur n'a pas les spécifications elles-mêmes, mais des pointeurs dans la partie spécifications d'une variable DSN d'un fichier de référence existant (ou de plusieurs fichiers de référence; la génération est alors analogue). Deux possibilités :

a) Le fichier référencé est sur le même ordinateur que celui où doit être créé le fichier indiqué. Le compilateur génère un I-MOUNT pour le volume sur lequel se trouve le fichier de référence, ensuite une I-instruction FDSB.

L'I-instruction FDSB est une instruction à 4 octets qui contient un pointeur vers une zone associée à une variable DSN et qui sert à remplir la partie spécification de la zone associée à la variable DSN en question, à partir des données contenues dans le descripteur physique du fichier. Après avoir généré ces deux I-instructions, il génère un I-MOUNT pour le volume sur lequel on crèera le nouveau fichier, suivi d'un I-CREATE pour ce fichier.

b) Le fichier référencé est sur un autre ordinateur que celui où doit être créé le fichier indiqué.
Les fonctions à générer correspondent à celles générées dans le cas précédent, mais réparties sur deux séquences d'instructions, interprétées en parallèle sur les deux ordinateurs différents. Pour que l'échange d'informations se passe correctement, il faut que les deux ordinateurs concernés soient synchronisés au moment de la transmission.

Des I-instructions appropriées existent.
L'I-instruction DPC sert à synchroniser les deux ordinateurs. Elle est à 4 octets et contient un pointeur vers la zone associée à la variable CPU qui correspond à l'ordinateur avec lequel il faut synchroniser.
Les I-instructions SINFO et RINFO ('send information' et 'receive information') qui vont de pair, qui font le transfert de l'information d'une zone indiquée sur l'ordinateur à une zone indiquée sur l'autre ordinateur sont à 6 octets et contiennent, outre le pointeur vers la zone, la longueur en caractères de l'information à transmettre.
SINFO et RINFO utilisées pour l'E-instruction CREATE pointeront des zones dans le bloc associé aux variables déclaratives.

EXEMPLES :

CREATE D1(DSCB1) ;	*(1)*	disque			ordinateur différent
CREATE D2(DCB1) ;	*(2)*	bande	même ordinateur		
CREATE D3(D2) ;	*(3)*	bande			
CREATE D4(D1) ;	*(4)*	disque			

(1)

CREATE	4	↑ D1

(2)

CREATE	4	↑ D2

(3)

MOUNT	4	↑ V2
FDSB	4	↑ D2
MOUNT	4	↑ V3
CREATE	4	↑ D3

(4)

MOUNT	4	↑ V1	
FDSB	4	↑ D1	
DPC	4	↑ C2	⟺
SINFO	6	info	longueur info

DPC	4	↑ C1	
RINFO	6	info	longueur info
MOUNT	4	↑ V4	
CREATE	4	↑ D4	

3.2.2.3.8. L'instruction ADD.

Syntaxe

ADD <PARID> TO <ID>;
Emetteur Récepteur

<PARID> ::= (<ID> <LID1>)
| <ID>

<LID1> ::= , <ID> <LID1>
| <RIEN>

Sémantique

Cette instruction sert à concaténer un fichier, un membre de fichier ou une liste de membres, à un fichier, un membre de fichier, une liste de membres de fichier (Toutes les combinaisons ne sont pas permises). Une vérification doit être faite par le compilateur selon le type de la E-variable et selon le type d'organisation du fichier.
A titre d'exemple, les compatibilités sont représentées dans le tableau suivant, pour les fichiers PS et PO, ADD B TO A;

Emetteur \ Récepteur	PS	PO	Membre	Liste de Membres	Spool
PS	Possible	Impossible	Séq→Membre	Impossible	Possible
PO	Impossible	Possible	Impossible	Impossible	Impossible
Membre	Possible	Membre → Membre de Même nom	Possible	Impossible	Possible
Liste de membres	Séquentiel = membres concaténés	Membres de même nom	Membre = membres concaténés	Corresp. membre à membre	Impossible
Spool	Possible	Impossible	Possible	Impossible	Possible

EXEMPLES :

```
ADD  D2  TO  D1  ;        (1)    fichier à fichier
ADD  M4  TO  D1  ;        (2)    membre à fichier
ADD  M5  TO  D1  ;        (3)    liste de membres à
                                 fichier
```

Code généré

1. Cas d'un seul fichier origine

Deux séquences d'I-instructions sont générées, une destinée à l'ordinateur sur lequel se trouve le fichier de référence, une destinée à l'ordinateur sur lequel se trouve le fichier récepteur. Outre les ordres de transmission, de synchronisation et de montage, de nouvelles I-instructions interviennent ici. L'I-instruction FVOLL ('fill volume list') à 4 octets,

qui contient un pointeur vers une zone associée à un DSN, sert à faire vérifier que le volume et l'unité sur lesquels se trouve le fichier indiqué par le pointeur, sont connus. Dans le cas d'un fichier catalogué où ces blocs sont vides, cette instruction provoque une consultation du catalogue et remplit les zones appropriées des zones associées au VOLUME et UNIT à partir des renseignements trouvés dans le catalogue. Les vérifications se font à l'exécution du programme (puisque le catalogue évolue dans le temps). Un fichier non-trouvé dans le catalogue donnera une erreur.

Les I-instructions RESDSR et RESDSW qui sont complémentaires et formellement pareilles ('réserve data set read' et 'reserve data set write') sont des instructions à 4 octets, contenant un pointeur vers une zone associée à une variable DSN. Elles servent à réserver le fichier indiqué par le pointeur en lecture ou en écriture.

Les fichiers doivent avoir été créés auparavant.

L'I-instruction COMPDSB ('compare dsb') est une instruction à 8 octets, contenant deux pointeurs vers des zones associées à des variables DSN et un pointeur vers la zone RESULT dans les renseignements généraux. Elle sert à comparer les deux zones associées aux DSN fournis en paramètre. Le résultat de la comparaison est stocké dans la zone RESULT. RESULT est composé de R1 et de R2. R1 est destiné à contenir le code condition retour après la comparaison et R2 à contenir un paramètre pour les fonctions de transmission (pour plus d'explications se reporter plus loin, aux I-instructions TRMS et TRMR). La comparaison des deux DSN se fait sur un seul ordinateur (le cas échéant, des instructions complémentaires sont générées pour amener ces informations sur le même ordinateur). Les tests, mentionnés plus haut, portent sur la taille des espaces donnés et occupés et sur les types d'organisation.

L'I-instruction CVTVCC ('Convert value to condition code') est une instruction à 4 octets contenant un pointeur vers une zone qui contient une valeur à convertir en code condition. L'instruction retourne le code condition égal à la valeur fournie en paramètre. (Ceci est le moyen pour transférer des codes condition d'un ordinateur à un autre).

Remarquons que ce traitement des codes condition est spécifique à OS.

Les I-instructions TRMS et TRMR ('transfer send' et 'transfer receive'), complémentaires et analogues, sont des instructions à 6 octets. Elles

contiennent un pointeur vers une zone associée à une variable DSN et un pointeur vers la zone R2. Le DSN correspond au fichier que l'on veut transmettre ou recevoir et dans R2 des renseignements relatif au transfert sont gardés.

Dans le cas de ADD d'un fichier à un autre fichier, la séquence générée pour l'ordinateur où se trouve le fichier origine comprend :

- Une I-instruction FVOLL, pointant sur le fichier origine
- Une " " MOUNT, pointant sur le VOLUME associé à ce fichier
- Une " " FDSB , " " DSN
- " " " RESDSR " " DSN
- Une I-instruction DPC, pointant sur la zone associée à la variable CPU, indiquant l'ordinateur où se trouve le fichier récepteur.
- Une I-instruction SINFO, pointant sur le fichier origine de longueur de la zone DSN
- Une I-instruction RINFO, pointant sur RESULT (donc à longueur 4 en deuxième paramètre)
- Une I-instruction CVTVCC, pointant sur R1 et
- " " TRMS, pointant sur le fichier origine et R2

La séquence générée pour l'ordinateur récepteur comprend :

- une I-instruction FVOLL, pointant le fichier récepteur
- une I-instruction MOUNT, pointant le VOLUME associé au fichier
- une I-instruction FDSB, pointant la zone DSN du fichier
- une I-instruction RESDSW pointant " " " "
- une I-instruction DPC pointant la zone CPU associée à l'ordinateur émetteur,
- une I-instruction RINFO, pointant le fichier origine et indiquant la longueur du transfert.
- une I-instruction COMPDSB, pointant les DSN des fichiers origine et récepteur
- une I-instruction SINFO, pointant sur RESULT (4 en deuxième paramètre)
- une I-instruction CVTVCC, pointant sur R1
- une I-instruction TRMR, pointant sur la zone associée au DSN du fichier récepteur et sur R2.

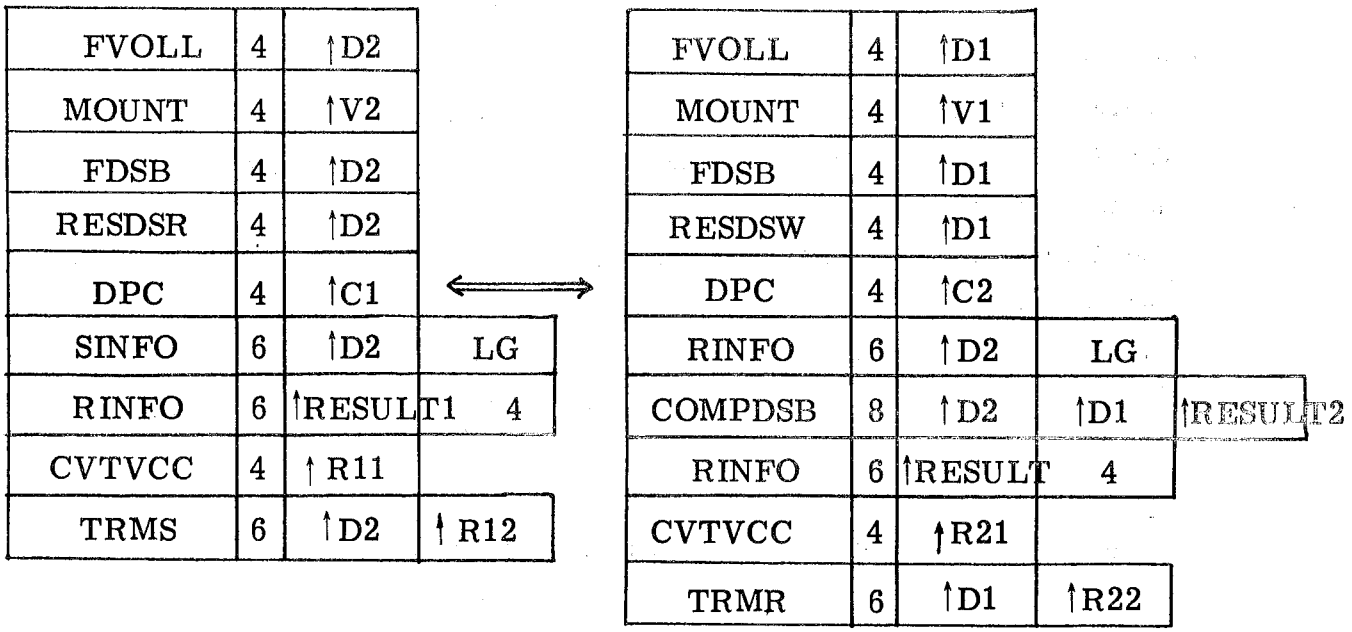
2. *Cas de plusieurs fichiers origine*

   L'opération se compose de plusieurs opérations telles que décrites sous *1.*

3. *S'il s'agit de membres ou de listes de membres au lieu de fichiers, la génération est similaire.*

   Des zones associées aux membres sont pointées au lieu des zones associées aux DSN.

Schéma des séquences générées pour l'exemple (1)



LG ≡ Longueur du fichier

3.2.2.3.9. L'instruction COPY (Copier)

Syntaxe

COPY <PARID> TO <ID> ;

Emetteur Récepteur

Sémantique

Par cette instruction on ordonne de copier le contenu de ou des premiers fichiers, membres ou listes de membres, dans le second fichier membre ou liste de membres, en écrasant éventuellement le contenu de ce second.

EXEMPLE :

```
COPY  D2  TO  D1 ;
```

Code généré

Une copie correspond à un vidage du fichier (membre, liste de membres) suivi d'un ou de plusieurs ADD selon le cas.

Le schéma des séquences générées dans l'exemple est exactement le même que pour l'exemple élaboré sous ADD, où l'on insère une I-instruction EMPTY dans la deuxième séquence après le FDSB.

3.2.2.3.10. L'instruction LIST (Lister)

Syntaxe

LIST <**PARID**> **AT** <**ID**> ;

Fichiers ... Ordinateur

Sémantique

Ordre de listage de ou des fichiers (membres, listes de membres) indiqués par le premier paramètre sur l'imprimante de l'ordinateur indiqué par le deuxième paramètre.

EXEMPLES :

```
LIST   D1   AT   IMAG.67;          (1)
LIST   D1   AT   C1;               (2)
```

(1) et *(2)* sont équivalents.

Code généré

Si le fichier de données destiné à être imprimé n'est pas sur l'ordinateur indiqué, le compilateur génère pour lui la séquence d'instructions LI correspondant à une E-instruction COPY sur un fichier 'spool'.

3.2.2.3.11.L'instruction RUN (exécuter)

Syntaxe

RUN <**ID**> **AT** <**ID**> <**OPTRUN**>;

Travail inséré Ordinateur

<**OPTRUN**> ::= **LIST AT** <**ID**>
| <**RIEN**> Ordinateur

Sémantique

Cette instruction ordonne l'exécution d'un travail enregistré dans le fichier indiqué par le premier paramètre sur l'ordinateur indiqué par le deuxième paramètre; optionnellement on peut demander une liste sur l'imprimante d'un ordinateur indiqué par un troisième paramètre. Si le troisième paramètre est omis, la liste est faite sur l'ordinateur d'origine. Le plus souvent, cette instruction va de pair avec une déclaration INP, le travail que l'on veut faire exécuter constituant un 'fichier spool' décrit par une zone associée DIS, mais en fait n'importe quel fichier séquentiel contenant un travail convient à l'instruction RUN.

EXEMPLE :

```
RUN   INP1   AT   C1   LIST   AT   C2 ;
```

Code généré

Essentiellement il y a trois phases distinctes dans l'exécution de l'instruction RUN, sauf bien entendu dans le cas de soumission sur l'ordinateur.

D'abord, le travail inséré dans le flot d'entrée, qui se trouve éventuellement depuis la déclaration correspondante INP dans un fichier 'spool', est transféré s'il y a lieu à l'ordinateur où il doit être exécuté.

Puis, l'I-instruction TRISS ('transmit input stream send') et TRISR ('transmit input stream receive') avec en opérande un pointeur vers le DIS du fichier 'spool' concerné servent au transfert.
La soumission au système local se fait par l'I-instruction INSERT, à longueur fixe de 6 octets et à deux opérandes; le premier opérande est un pointeur vers le fichier 'spool' local contenant le travail, le second opérande est un pointeur vers l'ordinateur de la sortie des résultats et des messages.

Enfin, pour envoyer et ramener les listes -résultats d'une exécution à distance, ainsi que les messages des systèmes SOC locaux, les I-instructions COPEX et RECCOPEX ont été inventées. COPEX contient en opérande un pointeur vers l'ordinateur vers lequel le transfert se fait. Elle est à 4 octets.

Les sorties habituelles du système d'exploitation local, obtenues par tous les RUN du travail-réseau concerné qui se sont déroulés sur l'ordinateur-émetteur, plus les messages, sont récupérées ensemble. A l'ordinateur-récepteur une instruction RECCOPEX ('receive copex') est issue, qui va de pair avec l'instruction COPEX. RECCOPEX n'a pas d'opérandes et est à deux octets. De part et d'autre,les instructions TRISS, TRISR, COPEX et RECCOPEX sont entourées par des instructions DPC et SPC ('Stop Processus Correspondant'). L'I-instruction SPC de désynchronisation entre deux ordinateurs est analogue à l'instruction DPC de synchronisation. SPC n'a pas d'opérandes et a donc une longueur fixe de 2 octets.

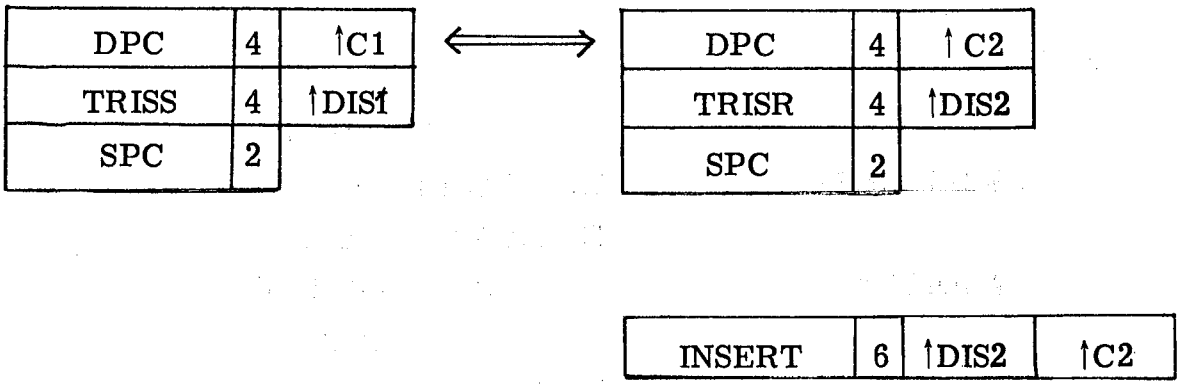
Schéma pour l'exemple.

Nous rappelons qu'il y a trois phases distinctes dans l'exécution de l'E-instruction de l'exemple qui correspondent à trois séquences d'I-instructions et où le problème de la synchronisation intervient.

La première phase est le transfert du fichier 'spool' qui contient le travail inséré. Pour assurer ce transfert il faut que les deux ordinateurs impliqués soient synchronisés. Une demande de synchronisation sera donc faite de part et d'autre par les ordinateurs impliqués, le transfert ordonné et l'arrêt de la synchronisation demandé.

La deuxième phase consiste à faire insérer le travail dans la file d'attente du système d'exploitation local et à le faire exécuter.

La troisième phase qui vise la recopie des résultats et des messages n'est en fait générée qu'une fois pour toutes à la fin du programme-réseau, pour le groupe de tous les travaux insérés et les messages. Comme, là encore, il faut que les deux ordinateurs impliqués soient synchronisés, des ordres de synchronisation et de désynchronisation entourent l'ordre de recopie.


DPC 4 ↑C1
TRISS 4 ↑DIS1
SPC 2
DPC 4 ↑C2
TRISR 4 ↑DIS2
SPC 2
INSERT 6 ↑DIS2 ↑C2
DPC 4 ↑C1
RECCOPEX 2
SPC 2
DPC 4 ↑C2
COPEX 4 ↑C2
SPC 2

3.2.2.3.12. Expressions arithmétiques

Syntaxe

```
                        <ID> := <EXPRARITH>
<EXPRARITH>     ::= <ID> <ADDITION>
                  | <ENTIER> <ADDITION>
<ADDITION>      ::= + <MULT> <ADDITION>
                  | - <MULT> <ADDITION>
                  | <RIEN>
<MULT>          ::= <ENTIER> * <ID>
                  | <ID>
                  | <RIEN>
```

Sémantique

Quelques possibilités d'arithmétique élémentaire ont été introduites dans LE.1 pour faciliter les opérations utilitaires sur les fichiers. Par exemple allouer à un fichier un espace, somme de deux autres. On exprime l'élaboration d'une somme algébrique de termes,chaque terme étant soit une valeur désignée par un nom, soit un entier que multiplie une valeur désignée par un nom.

Les noms peuvent être des noms de zones associées à des variables SPACE, LRECL ou BLKSIZE. Ce n'est que sur les variables LRECL, BLKSIZE et SPACE que des opérations arithmétiques sont prévues et définies, essentiellement dans le but de pouvoir définir des espaces se basant sur des espaces de fichiers qu'on veut copier ou concaténer.

Dans le cas des variables LRECL et BLKSIZE les opérations arithmétiques ont le sens habituel. Dans le cas de la variable SPACE, qui contient 4 éléments, une opération arithmétique sur deux SPACE n'est définie que dans le cas où les premiers éléments sont égaux (même unité de mesure. Si la mesure est exprimée en taille moyenne de blocs d'enregistrement, la plus grande taille sera gardée).

L'addition ou la soustraction correspondent à 3 additions ou soustractions au sens habituel avec comme opérandes les éléments correspondants des 3 derniers éléments des SPACE concernés.

La multiplication par un entier correspond à 3 multiplications au sens habituel avec comme opérandes l'entier et successivement chacun des 3 derniers éléments du SPACE.

EXEMPLES :

```
LV1:=L1+L2;
BV1:=800+B1-2*B2;
SV1:=2*S1+S2;
```

Code généré

Les I-instructions ADD, SUB et MUL (ADD4, SUB4 et MUL4 pour les SPACE) à 8 octets sont générées. Ces instructions ont un format de notation usuelle pour les instructions à trois opérandes

C := A + B ; étant écrit

+	A	B	C

Elles contiennent donc 3 pointeurs (vers deux zones d'opérandes et une zone résultat) ou valeurs (si les opérandes ou le résultat sont de type entier, la valeur elle-même est stockée dans l'I-instruction). Puisqu'il n'y a pas de parenthésage dans les expressions arithmétiques on n'a pas besoin de faire construire une pile pour elles par le compilateur. Une mémoire-tampon pour des résultats intermédiaires est nécessaire; elle est à un endroit fixe dans la zone de travail du compilateur.
En effet, la plus compliquée des expressions arithmétiques définissables en LE.1 se réduit à

$$m_1 * x_1 + m_2 * x_2$$

Comme on peut le voir dans l'exemple :

$$\underbrace{a * x + b * y}_{X} + c * z \qquad (1)$$

$$X + c * z$$

$$a * x + b * y \qquad (2)$$

Comme on évalue les produits dans l'ordre où on les rencontre, il faut pouvoir stocker leur résultat en attendant d'y ajouter ou d'y soustraire le terme suivant, mais il n'y a jamais plus d'un seul résultat intermédiaire à garder car il n'y a pas de parenthésage autorisé.

Schéma de génération pour les exemples :

(1)

ADD	8	↑L1	↑L2	↑LV1

(2)

ADD	8	800	↑B1	↑BV1
MUL	8	2	↑B2	↑BUFA
SUB	8	↑BV1	↑BUFA	↑BV1

(3)

MUL	8	2	↑S1	↑SV1
ADD	8	↑ SV1	↑S2	↑SV1

3.2.2.4. *Les ordres de contrôle et l'exécution d'un programme-réseau.*

3.2.2.4.1. L'instruction NETIN (entrée dans le réseau).

Syntaxe

NETIN (<ENTIER> , <ENTIER> , <ENTIER>) ,
(<ID> , <ID>) ;

Sémantique

L'instruction contient des informations comptables et utiles à la sortie des listes.Il y a :
- le n° de service ou numéro de compte
- le n° de programmeur
- le n° de programme
- le nom du programmeur
- une chaîne que l'on veut voir imprimée sur les listes de sortie

A l'instruction NETIN le lecteur-compilateur reconnaît un travail-réseau. Le programme-réseau qui suit est compilé et généré dans la zone de travail du compilateur.

EXEMPLE :

```
NETIN(5161,580,1),(DECALUWE,TRAV1 );
```

3.2.2.4.2. L'instruction NETOUT (sortie du réseau).

Syntaxe

NETOUT

Sémantique

A l'encontre du mot-clé NETOUT, le lecteur-compilateur reconnait la fin du travail-réseau et le processus-système INITIATEUR est appelé. Un processus

utilisateur est créé avec les résultats des travaux du compilateur. En cas d'erreurs à la compilation, il n'y a que le listage du travail-réseau à faire par l'intermédiaire du système-réseau. Si tout s'est bien passé, les séquences d'instructions LI sont transférées aux ordinateurs concernés et ces travaux sont insérés dans les files d'attente des systèmes d'exploitation. En plus, il y a également la liste du travail-réseau lui-même à sortir. En fait le compilateur LE.1 accepte en entrée un flot d'entrée constitué d'une suite de travaux réseau. Il doit à cet effet traiter avec soin les cas où les délimiteurs NETIN et NETOUT sont omis ou erronés.

3.2.2.4.3. Mise en oeuvre d'un programme généré en LI.

Bien que ne rentrant pas totalement dans la description du langage LE, ces familles d'instructions correspondent aux solutions plus générales adoptées dans le projet SOC pour :

- la synchronisation de processus s'exécutant sur des ordinateurs différents
- les problèmes de fiabilité devant permettre la reprise des travaux en cas d'incident.

Sans vouloir traiter en détail le problème de la synchronisation de processus telle qu'il est abordé dans SOC, on peut schématiser grossièrement le procédé en disant que chaque ordinateur dispose d'une file d'attente de travaux. Ces travaux sont sélectionnés pour exécution en fonction de la disponibilité des ressources locales. Une fois lancée, un processus A peut avoir à se synchroniser avec un processus B de même type sur un autre ordinateur. Soit B est actif et la synchronisation peut s'établir, soit il est encore en file d'attente, A s'endort et sera réactivé par le lancement de B.
Dans ce qui précède, les problèmes de synchronisation sont toujours traités par paire d'ordinateurs, même si le programme-réseau dans son ensemble fait intervenir plus de deux ordinateurs.

Un programme-réseau est par ailleurs décomposé en 'étapes' dans un double but :

- fiabilité
- exécution distribuée dans les temps en fonction de la disponibilité des ressources.

Outre les I-instructions que nous avons décrites à l'occasion de la génération des diverses instructions LE.1, il existe d'autres I-instructions qui ont pour fonction d'assurer la synchronisation de chaque paire d'ordinateurs en communication, et de régler l'exécution du travail-réseau pas à pas.

L'I-instruction DJC ('demande de job correspondant') est générée sur l'ordinateur de soumission en tête de la séquence LI, pour chaque ordinateur à qui le programme LE.1 fait appel. Elle contient en opérande, un pointeur vers l'ordinateur requis et est de 4 octets. Les I-instructions de transfert sont plusieurs. Nous avons déjà vu TRMR, TRMS, TRISS et TRISR et TRISR à l'occasion de l'instruction LE.1 RUN. On a encore l'I-instruction TRLIS ('transmit LI send'), de 2 octets sans opérandes, qui sert à ordonner, au moment de la mise en oeuvre du programme-réseau, le transfert du LI à l'ordinateur intéressé, avec lequel l'ordinateur d'émission est synchronisé.
Elle est prise entre une I-instruction DPC et une I-instruction SPC.

Quand deux ordinateurs sont synchronisés et que l'un d'eux a besoin de communiquer avec un troisième, la synchronisation est demandée par l'I-instruction DR('demande de réveil'). L'instruction contient en opérande un pointeur vers l'ordinateur avec lequel la synchronisation est demandée. La longueur de l'instruction est de 4 octets.

De l'autre côté, l'ordinateur délaissé pour le nouveau venu est désynchronisé par l' I-instruction JSUSP ('job suspend'). L'instruction ne contient pas d'opérandes et est à 2 octets. Pour assurer l'exécution pas à pas et permettre la reprise par pas, il y a l' I-instruction ES ('end of step'), sans opérandes, à 2 octets. La fin du travail-réseau est marquée pour chaque ordinateur intervenant, par l' I-instruction JEND ('end of step'), sans opérandes .

Les fichiers de données auxquels on a accès après une I-instruction RDSR ou

RDSW sont libérés par l'I-instruction FRDS ('free data set'), avec pointeur vers le fichier, et à 4 octets. L'emplacement d'un fichier 'spool' utilisé est libéré par l'I-instruction DELSP ('delete spool'), à 4 octets, avec pointeur vers le DIS du fichier 'spool'.

3.2.2.4.4. Exemple

Un travail-réseau est soumis à l'ordinateur SAC.91. Il contient un travail inséré dans le flot d'entrée qui doit être exécuté par l'ordinateur IMAG.67 et pour lequel la liste des résultats et des messages doit être produite par l'ordinateur CIRCE.75.
En langage externe, le travail-réseau s'écrit :

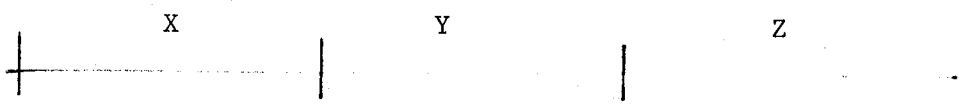
```
NETIN  (9999, 001, 1) , (DECALUWE,  TIFE );
INP    ALPHA :=  'DEL' (1,3) ;
//JOB  JOB  ...
:
//
DEL
RUN  ALPHA  AT  IMAG.67  LIST  AT  CIRCE.75 ;
NETOUT;
```

En langage interne trois séquences sont générées, une pour chaque ordinateur intervenant, comme le montre le schéma joint.
A SAC.91 un premier pas consiste à créer les travaux pour les différents ordinateurs et à y expédier le code. Après, le travail inséré dans le flot d'entrée est traité. On assure le transfert du contenu du fichier 'spool' à IMAG.67, l'insertion du travail dans le système OS local et le renvoi des sorties du RUN sur CIRCE.75. Ensuite le renvoi des messages sur l'ordinateur-origine est assuré à partir de IMAG.67 et CIRCE.75 et le travail des différents ordinateurs proprement terminé.

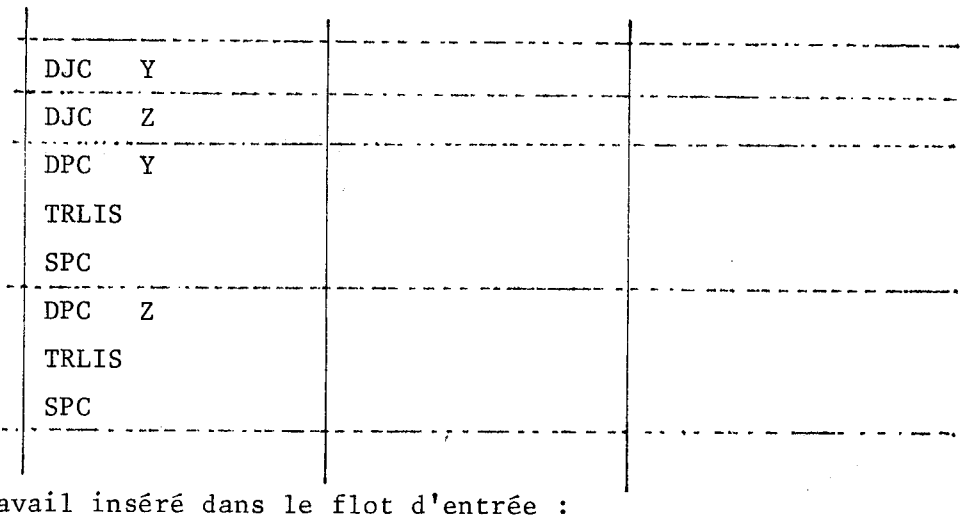
Schéma du code généré pour les instructions exécutables.

Nous notons :

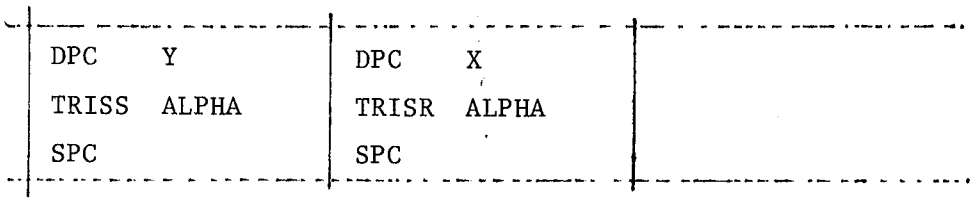
X pour l'ordinateur SAC.91
Y pour l'ordinateur IMAG.67
et Z pour l'ordinateur CIRCE.75



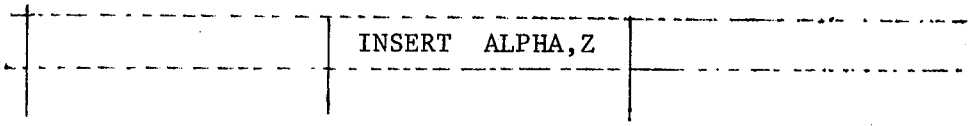
Création du travail-réseau et transfert du LI aux ordinateurs impliqués :
L'instruction DJC provoque le lancement sur l'ordinateur indiqué, du processus 'lecteur-interne' qui va alimenter la file d'attente de cet ordinateur à partir de l'ordinateur qui l'émet; le transfert se passe comme si l'ordinateur-récepteur du LI exécutait un ordre TRLIR ('Transmit LI Receive') qui travaille en parallèle avec TRLIS; le travail lu internement a un numéro connu des deux ordinateur- impliqués.



Transfert du travail inséré dans le flot d'entrée :
En traitant l'ordre de synchronisation DPC X, l'interpréteur du travail se charge de transférer le numéro de ce travail (implicitement)



Insertion dans le système local OS de l'IMAG.67 :



X	Y	Z

Renvoi des sorties du RUN de IMAG.67 sur CIRCE.75 :

	DPC Z COPEX Z SPC	DPC Y RECCOPEX SPC

Renvoi des messages de SOC vers l'origine à partir des autres ordinateurs impliqués :

DPC Y RECCOPEX SPC	DPC X COPEX X SPC	
DPC Z RECCOPEX SPC		DPC X COPEX X SPC

Terminaison :

ES DELSP ALPHA JEND	ES DELSP ALPHA JEND	ES DELSP ALPHA JEND

3.3. Le langage de programmation ALGOLW comme langage d'écriture d'un prototype de compilateur.

Pour réaliser un prototype du compilateur LE.1, nous avons utilisé le langage AlgolW.

Nous avons apprécié particulièrement certaines de ses qualités, mais quelques aspects du langage et de son implémentation nous ont mis quelquefois en difficulté pour arriver à notre but.

Le compilateur qui était disponible au moment où nous avons commencé la programmation est la version 2 d'août 1969 qui ne permet pas l'assemblage de procédures précompilées ou l'écriture de procédures en code. Assez tardivement, vis-à-vis de l'évolution de notre travail, la version 3 d'octobre 1971 du compilateur est arrivée. Nous avons continué la programmation en cours, sans pouvoir recourir à la précompilation de certaines procédures -ce qui nous aurait tant convenu!- à cause d'un problème de passage de références en paramètre, pas résolu dans la nouvelle version. Une version privée en a été cependant fabriquée, dans laquelle les limitations du compilateur ont été changées.

Aspects appréciables.

Le langage est facile à écrire, y compris la manipulation de listes.
Le compilateur est rapide et le code généré s'exécute vite.
Un détail appréciable est la longueur permise des identificateurs qui est de 256 (avec des blancs significatifs possibles).

Aspects contraignants.

Trois grandes critiques sont à faire :
Le compilateur est trop limitatif pour traiter des programmes de grande envergure.
Le nombre de classes d'enregistrements par exemple est réduit à 15 (14 par erreur dans la version fournie). Le compilateur modifié qui a été utilisé en permet 63.

D'autres limitations peuvent être contraignantes, mais sont plus ou moins faciles à pallier par programmation, par exemple le nombre de cas possibles dans une instruction "CASE", le niveau d'imbrication des blocs (statique, parce qu'il est lié à un nombre de registres), le nombre de déclarations de procédure permises.

- Le passage de références en paramètre n'est pas au point (pour les procédures précompilées).
  D'une façon générale, le langage n'est pas commode pour exprimer des notions qui sont près de la construction physique de la machine (bits, octets, adresses), sinon à grands frais.

- Les possibilités d'entrée-sortie ne sont pas assez soignées: très limitatives ou uniquement accessibles par l'intermédiaire de procédures en assembleur, PL360, Fortran.

Bilan.

Il nous parait que le langage est très commode et le compilateur très efficace pour traiter des programmes de petite et de moyenne taille (plutôt limités en nombre de déclarations qu'en nombre d'instructions), pour fixer les idées jusqu'à 500 instructions, dont un 1/10e de déclarations.
En ce qui concerne les grands programmes, qui se mesurent en milliers d'instructions, même la dernière version peut poser des problèmes, parce que le compilateur n'offre pas assez de facilités à cause de son caractère limitatif, pour passer des variables en paramètre et pour les possibilités d'entrée-sortie qui sont trop réduites.

CONCLUSION

Nous espérons que notre travail puisse être considéré comme une contribution concrète dans le domaine de l'étude systématique des réseaux d'ordinateurs. Nous voyons son importance dans le fait qu'il a servi à saisir les difficultés de la définition du langage de contrôle et de commande pour le réseau qui est une remise en question répétée pour trouver un compromis entre les possibilités que l'on veut mettre dans le langage et les problèmes d'implémentation qui y sont afférents.

En gros, nous pouvons dire que le langage défini se rapproche des langages de commande des systèmes existants où les facilités des programmes utilitaires ont été incorporées. La définition syntaxique est celle d'un langage de haut niveau qui se prête à être analysée par des méthodes classiques.

La génération de code en parallèle pour plusieurs ordinateurs est une particularité remarquable du compilateur.

B I B L I O G R A P H I E

[AHO] A.V.AHO, P.J.DENNING, J.D.ULLMAN
"Weak and mixed strategy precedence parsing"
JACM 19,2,Avril 1972.

[ASP] System Programmer's Manual (GH20-0323-8)
Application Programmer's Manual(GH20-0322-7)
System Description (GH20-0466-6)

[BOUS] J.C. BOUSSARD
"Etude et réalisation d'un compilateur Algol 60 sur calculateur électronique du type IBM 7090/94 et 7040/44".
Thèse, IMAG, Juin 1964.

[BROO] R. BROOKER, I.MACCALLUM, D.MORRIS, J.ROHL
The Compiler's Compiler.
Third Annual Review of Automatic Programming.
Pergamon Press, 1963.

[CARR] S.CARR et AL
"Host-Host Communication Protocol in the ARPA Network"
Proc.SJCC 1970, pp.589-597.

[CHEA] T.E.CHEATHAM Jr., KIRK SATTLEY,
"Syntax directed compiling"
Proc.EJCC, AFIPS, Vol.25, pp.31-57, 1964

[COL1] A.COLMERAUER
"Total precedence analysis"
JACM 17,1, Janvier 1970.

[COL2] A.COLMERAUER
"Precedence, analyse syntaxique et langages de programmation"
Thèse d'Etat, Grenoble, Septembre 1967.

[CP67] Control Program-67 (Cambridge Monitor System)
"User's Guide"
1970

[CROC] S.D. CROCKER, J.F.HEAFNER, R.M.METCALFE, J.B.POSTEL
"Function-oriented protocols for the ARPA Computer Network"
AFIPS Vol 40 SJCC 1972.

[DECA] R.de CALUWE
"Etude d'un langage de commande pour un système d'ordinateurs connectés".
Rapport de DEA, IMAG 1971.

[DUMA] J.du MASLE
"Etude d'un compilateur Algol à l'aide d'un système autocodeur"
Congrès AFIRO, Avril 1964.

[ELI1] M.ELIE
"Le réseau d'ordinateurs de l'ARPA"
CII

[ELI2] M.ELIE
"General Purpose Network of Computers"
Masters Thesis Dept.of Comp.Science
Univ.of California, Los Angeles (UCLA) 1970

[FLAN] L.K. FLANIGAN
Introduction to the Computing Center and to MTS (Michigan Terminal System)
Computing Center and Department of Computer and Communication Sciences, University of Michigan, September 1969.

[FLOY] R.W.FLOYD
"Syntactic analysis and operator precedence"
JACM 10,3, Juillet 1963.

[GOTT] L.R.GOTTSCHALL
"Cybernet-General Design for Shadow"
54073-1XX-1- March 1970.

[GRAH] M.L.GRAHAM
"Illiac Control Language"
Univ.of Illinois, October 1970.

[GRIF] M.GRIFFITHS
"Analyse déterministe et compilateurs"
Thèse d'Etat, IMAG, octobre 1969.

[ICL] Serie ICL 1900
Manuel descriptif
1970.

[ICOM] Intercom2 Reference Manual
CDC 6000 Series Computer Systems

[JCLR] IBM System/360 Operating System
Job Control Language Reference
1970.

[MORR] D.MORRIS, I.R.WILSON
A system Program Generator
Computer Sciences Dept.Manchester University, 3.10.67.

[OSUL] T.O'SULLIVAN et AL
Telnet protocol
Arpa Network Working Group Request for Comments
(RFC) 158
ARPA Network Information Center (NIC)
6768 May 1971.

[RAND] B. RANDELL and L.J. RUSSELL
"Algol 60 Implementation"
Academic Press, 1964.

[SAMM] J.SAMMETT
"Programming Languages-History and Fundamentals"
Pergamon 1969.

[SHAD] SHADOW/SHADE
Communications Software System.
"Terminal User Guide"
CDC 1970.

[SOMI] M.SOMIA
"Le projet SOC"
Congrès AFCET, Grenoble, novembre 1972.

[TSSC] TSS/360 SYSTEMS REFERENCE LIBRARY IBM
"Command System User's Guide" Form C28-2001.

[WIR1] N.WIRTH, H.WEBER
"Euler: a generalization of Algol and its formal definition P1"
CACP 9,1, Janvier 1966.

[WIR2] N.WIRTH
A programming Language for the 360 Computers
Stanford University.

ANNEXES

ANNEXE I

LISTE DES MOTS-CLE AVEC LEUR SIGNIFICATION USUELLE

Mot clé	Anglais	Français	Contexte
* RECFM	Record Format	Format d'enregist.	
* F	Fixed	Fixe	Longueur d'enregis.
* U	Undefined	Indéfini	" "
* V	Variable	Variable	
* FB	Fixed blocked	Fixe bloqué	Blocage d'enregis.
* VB	Variable blocked	Variable bloquée	" "
* VS	Variable spanned	Variable étendue	" "
* VBS	Variable blocked spanned	Variable bloquée étendue	
* PS	Physical Sequential	Séquentiel	Organisation de fichiers
* DA	Direct access	Accès direct	" "
* IS	Indexed sequential	Indexé séquentiel	" "
* PO	Partionned Organisation	Partitionné	"
* DSORG	Data set organisation	Organisation de fichier	"
* SL	Standard Label	Etiquette standard	Etiquette volume
* NL	No Label	Sans Etiquette	"
* BLP	By pass label processing	Ne pas traiter étiquette	"
CPU	Central Processor Unit	Unité Centrale	Localisation
* UNIT	Unit	Unité	"
* VOLUME	Volume	Volume	"
* DSN	Data set name	Nom d'ensemble de données	"
MBR	Member	Membre	"
**RING	Ring	Avec anneau	de protection écriture
**NORING	No Ring	Sans anneau	de protection écriture

* Repris de l'OS/360; ** Repris d'ASP

* SPACE	Space	Espace	Allocation d'espace par fichier
* TRK	Track	Piste	" "
* CYL	Cylinder	Cylindre	" "
ALLOC	Allocated	Alloué	" "
* LRECL	Logical Record Length	Longueur d'enregis.logique	
* BLKSIZE	Block size	Taille de Bloc	
COPY	Copy	Copier	
RUN	Run	Exécuter	
ADD	Add	Ajouter	
CREATE	Create	Créer	
DELETE	Delete	Détruire	
EMPTY	Empty	Vider	
CATALG	Catalog	Cataloguer	
UNCTLG	Uncatalog	Décataloguer	
MOUNT	Mount	Monter	
DISMOUNT	Dismount	Démonter	
QUALIFY	Qualify	Qualifier	
DEFAULT	Default	Défaut	

ANNEXE II

GRAMMAIRE DE REPRESENTATION

```
<NETJOB>        ::=  NETIN ( <ENTIER> , <ENTIER> , <ENTIER> , <ENTIER> ) ,
                     ( <ID> , <ID> ) ; <NETPROGRAM> NETOUT
<NETPROGRAM>    ::=  CPU <DCLCPU> <STDCLCPU> <STPROG>
                   | UNIT <DCLUNIT> <STDCLUNIT> <STPROG>
                   | VOLUME <DCLVOL> <STDCLVOL> <STPROG>
                   | VOLLIST <DCLVLL> <STDCLVLL> <STPROG>
                   | DSN <DCLDSN> <STDCLDSN> <STPROG>
                   | MBR <DCLMBR> <STDCLMBR> <STPROG>
                   | LMBR <DCLLMBR> <STDCLLMBR> <STPROG>
                   | QUALIFY <ID> BY <ID> <STPROG>
                   | DEFAULT <CHOIXDEF> <LCHOIXDEF> <STPROG>
                   | LEVEL <ENTIER> <OPTLEVEL> <STPROG>
                   | RECFM <DCLRECFM> <STDCLRECFM> <STPROG>
                   | LRECL <DCLLRECL> <STDCLLRECL> <STPROG>
                   | BLKSIZE <DCLBLKSIZE> <STDCLBLKSIZE> <STPROG>
                   | RECFM <DCLRECFM> <STDCLRECFM> <STPROG>
                   | DSORG <DCLDSORG> <STDCLDSORG> <STPROG>
                   | SPACE <DCLSPACE> <STDCLSPACE> <STPROG>
                   | REC <DCLREC> <STDCLREC> <STPROG>
                   | DSCB <DCLDSCB> <STDCLDSCB> <STPROG>
                   | INP <ID> := ' <TEXT> '
                     <INP1> ; INPUTSTREAM ' <TEXT> '
                   | MSG <ID> ' <TEXT> ' <STPROG>
                   | MOUNT     <ID> <SPECBANDE> <STPROG>
                   | DISMOUNT     <ID> <STPROG>
                   | EMPTY <ID> <STPROG>
                   | DELETE <ID> <STPROG>
                   | CATALG <ID> <OPTCATALG> <STPROG>
                   | UNCTLG <ID> <OPTCATALG> <STPROG>
                   | COPY <PARID> TO <ID> <STPROG>
                   | ADD <PARID> TO <ID> <STPROG>
                   | LIST <PARID> AT <ID> <STPROG>
                   | RUN <ID> AT <ID> <OPTRUN> <STPROG>
                   | CREATE <ID> ( <ID> ) <STPROG>
                   | <ID> := <EXPRARITH> <STPROG>
                   | COMMENT ' <TEXT> ' <STPROG>
                   | <RIEN>
<STPROG>        ::=  ; <NETPROGRAM>
<DCLCPU>        ::=  <ID> <IDCL> <SEPUNIT>
                   | <RIEN>
<IDCL>          ::=  := <ID> <IDCL>
                   | <RIEN>
<STDCLCPU>      ::=  , <DCLCPU> <STDCLCPU>
                   | <RIEN>
<SEPUNIT>       ::=  / <SECUNIT> <SEPVOL>
                   | <RIEN>
<SECUNIT>       ::=  <ID> <PARTUNIT>
                   | <ENTIER>
                   | <RIEN>
<DCLUNIT>       ::=  <ID> <PARTUNIT> <SEPVOL>
<PARTUNIT>      ::=  := <VALUNIT>
                   | <RIEN>
<VALUNIT>       ::=  <ID> <PARTUNIT>
                   | <ENTIER>
<SEPVOL>        ::=  / <SECVOL> <SEPDSN>
                   | <RIEN>
```

```
<SECVOL>          ::=  <ID> <PARTVOL>
                     |  <ENTIER>
                     |  (<LIDENT>)
                     |  <RIEN>
<STDCLUNIT>       ::=  <ID> <PARTVOL> <SEPDSN>
<PARTVOL>         ::=  := <VALVOL>
                     |  <RIEN>
<VALVOL>          ::=  <ID> <PARTVOL>
                     |  <ENTIER>
                     |  (<LIDENT>)
<LIDENT>          ::=  <ID> <LID>
                     |  <ENTIER> <LID>
<LID>             ::=  , <ENTID> <LID>
                     |  <RIEN>
<STDCLVOL>        ::=  , <DCLVOL> <STDCLVOL>
                     |  <RIEN>
<SEPDSN>          ::=  / <ID> <IDCL> <SEPMBR>
                     |  <RIEN>
<DCLMBR>          ::=  <ID> <PARTMBR>
<PARTMBR>         ::=  := <VALMBR>
<VALMBR>          ::=  <ID> <PARTMBR>
                     |  ( <ID> <STID>)
<STID>            ::=  , <ID> <STID>
                     |  <RIEN>
<STDCLMBR>        ::=  , <DCLMBR> <STDCLMBR>
                     |  <RIEN>
<DCLVLL>          ::=  <ID> <IDCL> <SEPDSN>
<STDCLVLL>        ::=  , <DCLVLL> <STDCLVLL>
                     |  <RIEN>
<STID>            ::=  , <ID> <STID>
                     |  <RIEN>
<DCLLMBR>         ::=  <ID> := <ID> <IDCL>
<STDCLLMBR>       ::=  , <DCLMBR> <STDCLLMBR>
                     |  <RIEN>
<SEPMBR>          ::=  / <VALMBR>
<DCLRECFM>        ::=  <ID> <PARTRECFM>
<PARTRECFM>       ::=  := <IDCLRM>
<IDCLRM>          ::=  <ID> <PARTRECFM>
                     |  F
                     |  FB
                     |  V
                     |  VB
                     |  VS
                     |  VBS
                     |  U
<STDCLRECFM>      ::=  , <DCLRECFM> <STDCLRECFM>
                     |  <RIEN>
<DCLDSORG>        ::=  <ID> <PARTDSORG>
<PARTDSORG>       ::=  := <IDCLDS>
<IDCLDS>          ::=  <ID> <PARTDSORG>
                     |  PS
                     |  DA
                     |  IS
                     |  PO
<STDCLDSORG>      ::=  , <DCLDSORG> <STDCLDSORG>
                     |  <RIEN>
```

```
<DCLLRECL>      ::=  <ID> <PARTUNIT>
<STDCLLRECL>    ::=  , <DCLLRECL> <STDCLLRECL>
                   | <RIEN>
<DCLBLKSIZE>    ::=  <ID> <PARTUNIT>
<STDCLBLKSIZE   ::=  , <DCLBLKSIZE> <STDCLBLKSIZE>
                   | <RIEN>
<DCLSPACE>      ::=  <ID> <PARTSPACE>
<PARTSPACE>     ::=  := <LSPACE>
                   | <RIEN>
<LSPACE>        ::=  <ID> <PARTSPACE>
                   | ( <MESURE> <LSPACE1> )
                   | ALLOC <ID>
<LSPACE1>       ::=  , <LSPACE2>
                   | <RIEN>
<LSPACE2>       ::=  <ID>
                   | ( <VENTID> <LSPACE3> )
<LSPACE3>       ::=  , <VENTID> <LSPACE4>
                   | <RIEN>
<LSPACE4>       ::=  , <VENTID>
                   | <RIEN>
<VENTID>        ::=  <ID>
                   | <ENTIER>
                   | <RIEN>
<MESURE>        ::=  CYL
                   | TRK
                   | <ENTIER>
<STDCLSPACE>    ::=  , <DCLSPACE> <STDCLSPACE>
                   | <RIEN>
<DCLREC>        ::=  <ID> <PARTREC>
<PARTREC>       ::=  := <VALREC>
<VALREC>        ::=  <ID> <PARTREC>
                   | ( <VIDCLRM> <LREC1> )
<LREC1>         ::=  , <VIDCLV> <LREC2>
                   | <RIEN>
<LREC2>         ::=  , <VIDCLV>
                   | <RIEN>
<STDCLREC>      ::=  , <DCLREC> <STDCLREC>
                   | <RIEN>
<VIDCLRM>       ::=  <ID> <PARTRECFM>
                   | <RIEN>
<VIDCLV>        ::=  <ID> <PARTUNIT>
                   | <RIEN>
<DCLDSCB>       ::=  <ID> <PARTDSCB>
<PARTDSCB>      ::=  := <VALDSCB>
                   | <RIEN>
<VALDSCB>       ::=  <ID> <PARTDSCB>
                   | ( <VIDCLSP> <LDSCB1> )
<LDSCB1>        ::=  , <VIDCLREC> <LDSCB2>
                   | <RIEN>
<LDSCB2>        ::=  , <VIDCLDS>
                   | <RIEN>
<STDCLDSCB>     ::=  , <DCLDSCB> <STDCLDSCB>
                   | <RIEN>
<VIDCLSP>       ::=  <ID> <PARTSPACE>
                   | <RIEN>
<VIDCLREC>      ::=  <ID> <PARTREC>
```

```
                    |   <RIEN>
<VIDCLDS>       ::=   <ID> <PARTDSORG>
                    |   <RIEN>
<CHOIXDEF>      ::=   SPACE := ( <MESURE> <LSPACE1> )
                    |   LRECL := <ENTIER>
                    |   BLKSIZE := <ENTIER>
<CHOIXDEF>      ::=   , <CHOIXDEF> <LCHOIXDEF>
                    |   <RIEN>
<OPTLEVEL>          ::=   , DUMP
                    |   <RIEN>
<LIN1>          ::=   ( <ENTIER> <INP2> )
                    |   <RIEN>
<INP2>          ::=   , <ENTIER>
                    |   <RIEN>
<ENTID>         ::=   <ENTIER>
                    |   <ID>
<SPECBAND2>     ::=   ( <OPTSTRING> , <OPTLABEL> )
                    |   <RIEN>
<OPTSTRING>     ::=   RING
                    |   NORING
                    |   <RIEN>
<OPTLABEL>      ::=   SL
                    |   NL
                    |   BLP
                    |   <RIEN>
<OPTCATALG>     ::=   ON <ID>
                    |   <RIEN>
<PARID>         ::=   ( <ID> <LID1> )
                    |   <ID>
<LID1>          ::=   , <ID> <LID1>
                    |   <RIEN>
<OPTION>        ::=   LIST AT <ID>
                    |   <RIEN>
<EXPARITH>      ::=   <ID> <ADDITION>
                    |   <ENTIER> <ADDITION>
<ADDITION>      ::=   + <MULT> <ADDITION>
                    |   - <MULT> <ADDITION>
                    |   <RIEN>
<MULT>          ::=   <ENTIER> * <ID>
                    |   <ID>
                    |   <RIEN>
<ID>            ::=   <DIGIT> <ALPHAMER>
                    |   <LETTER> <ALPHAMER>
                    |   . <ALPHAMER>
<ENTIER>        ::=   <DIGIT> <STENTIER>
<TEXT>          ::=   <DIGIT> <TEXT>
                    |   <LETTER> <TEXT>
                    |   , <TEXT>
                    |   <SPECIALTOKEN> <TEXT>
<ALPHAMER>      ::=   <ALPHAMER>
                    |   <LETTER>
<ALPHAMER>      ::=   <DIGIT>
                    |   <LETTER>
                    |   .
<STENTIER>      ::=   <DIGIT> <STENTIER>
                    |   <RIEN>
```

```
<DIGIT>          ::=  0|1|2|3|4|5|6|7|8|9
<LETTER>         ::=  A|B|C|D|E|F|G|H|I|
                      J|K|L|M|N|O|P|Q|R|
                        S|T|U|V|W|X|Y|Z
<SPECIALTOKEN>::=  +|-|*|/|;|,|'|(|)|:=
```

ANNEXE III

GRAMMAIRE D'IMPLEMENTATION

```
<DECIDEPROG>   ::=  NETIN ( <ENTIER> , <ENTIER> , <ENTIER> ) ,
                    ( <ID> , <ID> ) ; <STENONCES> NETOUT
                 |  EOF
<STENONCES>    ::=  CPU <INSTRDCL> <FSINSTR>
                 |  UNIT <INSTRDCL> <FSINSTR>
                 |  VOLUME <INSTRDCL> <FSINSTR>
                 |  VOLLIST <INSTRDCL> <FSINSTR>
                 |  DSN <INSTRDCL> <FSINSTR>
                 |  MBR <INSTRDCL> <FSINSTR>
                 |  LMBR <INSTDCL> <FSINSTR>
                 |  QUALIFY <ID> BY <ID> <FSINSTR>
                 |  DEFAULT <CHOIXDEF> <LCHOIXDEF> <FSINSTR>
                 |  LEVEL <ENTIER> <OPTLEVEL> <FSINSTR>
                 |  SPECL <SPECDSN1> <FSINSTR>
                 |  BLKSIZE <SPECDSN1> <FSINSTR>
                 |  RECFM <SPECDSN1> <FSINSTR>
                 |  DSORG <SPECDSN1> <FSINSTR>
                 |  SPACE <SPECDSN3> <FSINSTR>
                 |  REC <SPECDSN4> <FSINSTR>
                 |  DSCB <SPECDSN5> <FSINSTR>
                 |  INP <ID> :=        <TEXTE> <INP1> <FSINSTR>
                 |  MSG <ID>    <TEXTE>    <FSINSTR>
                 |  MOUNT      <ID> <SPECBANDE> <FSINSTR>
                 |   DISMOUNT      <ID> <FSINSTR>
                 |  EMPTY <ID> <FSINSTR>
                 |  DELETE <ID> <FSINSTR>
                 |  CATALG <ID>  ON  <ID> <FSINSTR>
                 |  UNCTLG <ID>  ON  <ID> <FSINSTR>
                 |  COPY <PARID> TO <ID> <FSINSTR>
                 |  ADD <PARID> TO <ID> <FSINSTR>
                 |  LIST <PARID> AT <ID> <FSINSTR>
                 |  RUN <ID> AT <ID> <OPTRUN> <FSINSTR>
                 |  CREATE <ID> ( <ID> ) <FSINSTR>
                 |  <ID> := <EXPRARITH> <FSINSTR>
                 |  ; <STENONCES>
                 |  COMMENT <STENONCES>
                 |  <RIEN>
<INP1>         ::=  ( <ENTIER> <INP2>
                 |  <RIEN>
<INP2>         ::=  )
                 |  , <ENTIER> )
<FSINSTR>      ::=  ; <STENONCES>
<LID>          ::=  , <ENTID> <LID>
                 |  <RIEN>
<PARID>        ::=  ( <ID> <LID> )
                 |  <ID>
<LIDENT>       ::=  <ID> <LID>
                 |  <ENTIER> <LID>
<LENTID>       ::=  <ENTIER> <LID>
                 |  <ID>
<RENTID>       ::=  <ENTIER>
                 |  <ID>
                 |  <RIEN>
<LRENTID>      ::=  , <RENTID> <LRENTID>
                 |  <RIEN>
<INSTRDCL>     ::=  ( <LIDENT> ) <INSTRDCL1> <LINSTRDCL>
```

```
                    |  <ID> <IDCL1> <INSTRDCL1> <LINSTRDCL>
                    |  <ENTIER> <INSTRDCL1> <LINSTRDCL>
<IDCL1>        ::=  := <IDCL2>
                    |  <RIEN>
<IDCL2>        ::=  ( <LIDENT> )
                    |  <ENTIER>
                    |  <ID> <IDCL1>
<INSTRDCL1>    ::=  / <REPETSLASH> <INSTRDCL> <INSTRDCL1>
                    |  <RIEN>
<REPETSLASH>   ::=  / <REPETSLASH>
                    |  <RIEN>
<LINSTRDCL>    ::=  , <INSTRDCL> <LINSTRDCL>
                    |  <RIEN>
<OPTLEVEL>     ::=  , NUM
                    |  <RIEN>
<CHOIXDEF>     ::=  SPACE := <SPECDSN6>
                    |  REC := <SPECDSN5>
                    |  DSCRC := <SPECDSN2>
<LCHOIXDEF>    ::=  , <CHOIXDEF> <LCHOIXDEF>
                    |  <RIEN>
<SPECDSN1>     ::=  <ID> <SPECDSN11>
<SPECDSN11>    ::=  := <SPECDSN2> <LSPECDSN1>
                    |  <RIEN>
<LSPECDSN1>    ::=  , <SPECDSN1> <LSPECDSN1>
                    |  <RIEN>
<SPECDSN12>    ::=  PS
                    |  SA
                    |  IS
                    |  PG
                    |  F
                    |  FB
                    |  J
                    |  VB
                    |  U
                    |  VS
                    |  VBS
                    |  <ENTIER>
                    |  <ID> <SPECDSN12>
<SPECDSN12>    ::=  := <SPECDSN2>
                    |  <RIEN>
<SPECDSN3>     ::=  <ID> <SPECDSN31>
<SPECDSN31>    ::=  := <SPECDSN6> <LSPECDSN3>
                    |  <RIEN>
<LSPECDSN3>    ::=  , <SPECDSN3> <LSPECDSN3>
                    |  <RIEN>
<SPECDSN4>     ::=  ALLOC <ID>
                    |  ( <TYPEREDEFV> , <SPECDSN7> )
                    |  <ID> <SPECDSN42>
<SPECDSN42>    ::=  := <SPECDSN6>
                    |  <RIEN>
<SPECDSN7>     ::=  ( <REMTID> <LREMTID> )
                    |  <ENTIER>
                    |  <RIEN>
<SPECDSN4>     ::=  <ID> := <SPECDSN6> <LSPECDSN4>
<LSPECDSN4>    ::=  , <SPECDSN4> <LSPECDSN4>
                    |  <RIEN>
```

```
<SPECDSN8>      ::=  ( <SPECDSN2> , <SPECDSN1> , <SPECDSN1> )
                  |  <ID> <SPECDSN42>
<SPECDSN42>     ::=  := <SPECDSN8>
                  |  <RIEN>
<TYPERESERV>    ::=  TRK
                  |  CYL
                  |  <ENTIER>
<SPECDSN5>      ::=  <ID> := <SPECDSN10> <LSPECDSN5>
<LSPECDSN5>     ::=  , <SPECDSN5> <LSPECDSN5>
                  |  <RIEN>
<SPECDSN10>     ::=  ALLOC <ID>
                  |  ( <SPECDSN3> , <SPECDSN4> , <SPECDSN1> )
                  |  <ID> <SPECDSN52>
<SPECDSN52>     ::=  := <SPECDSN10>
                  |  <RIEN>
<SPECBANDE>     ::=  ( <OPTRING> , <OPTLABEL> )
                  |  <RIEN>
<OPTRING>       ::=  RING
                  |  NOPING
                  |  <RIEN>
<OPTLABEL>      ::=  SL
                  |  NL
                  |  BLP
                  |  <RIEN>
<OPTRUN>        ::=  LIST AT <ID>
                  |  <RIEN>
<EXPRARITH>     ::=  <ID> <EXPR1> <EXPR3>
                  |  <ENTIER> <EXPR2> <EXPR3>
<EXPR1>         ::=  * <ENTIER>
                  |  <RIEN>
<EXPR2>         ::=  * <ID>
                  |  <RIEN>
<EXPR3>         ::=  + <EXPRARITH>
                  |  - <EXPRARITH>
                  |  <RIEN>
```

ANNEXE IV

INSTRUCTIONS LI

Nom	n° en hexa	Longueur	Type	Utilisé dans	Associé en parallèle à	Associé en Séquentiel à
LVL	00	4	F, PI	LEVEL		
SINFO	01	6	F, PI	CREATE-ADD RUN	RINFO	
RINFO	02	6	F, PI	CREATE-ADD RUN	SINFO	
CVTVCC	03	4	F, P	ADD-RUN		
ADD	04	8	F, PPP	Expr.Arith.		
MUL	05	8	F, PPP	Expr.Arith		
SUB	06	8	F, PPP	Expr.Arith		
ADD4	07	8	F, PPP	Expr.Arith		
MUL4	08	8	F, PPP	Expr.Arith		
SUB4	09	8	F, PPP	Expr.Arith		
DJC	0A	4	F, P	Initialisation		
DPC	0B	4	F, P	Synchron.	DPC	SPC
DR	0C	4	F, P	Synchron.		
ES	0D	2	F	Terminaison		
SPC	0E	2	F	Synchron.	SPC	DPC
JSUSP	0F	2	F	Terminaison		
JEND	10	2	F	Libération		
INSERT	11	6	F, PP	RUN		
MOUNT	12	4 ou 6	V, PI	MOUNT-EMPTY CREATE-ADD RUN		DISMOUNT
DISMOUNT	13	4	F, P	MOUNT-EMPTY CREATE-ADD RUN		MOUNT
CTLG	14	4 ou 6	V, PP	CATALG		UNCTLG
UNCTLG	15	4 ou 6	V, PP	UNCTLG		CTLG
EMPTY	16	4	F, P	EMPTY-COPY		

Nom	n° en hexa	Longueur	Type	Utilisé dans	Associé en parallèle à	Associé en Séquentiel à
DELETE	17	4	F, P	DELETE		
CR	18	4	F, P	CREATE		
FDSB	19	4	F, P	CREATE-ADD RUN		
FVOLL	1A	4	F, P	ADD-RUN		FRDS
RESDSR	1B	4	F, P	ADD-RUN		FRDS
RESDSW	1C	4	F, P	ADD-RUN		
TRMS	1D	6	F, PP	ADD-RUN	TRMR	
TRMR	1E	6	F, PP	ADD-RUN	TRMS	
COMPDSB	1F	8	F, PPP	ADD-RUN		
FRDS	20	4	F, P	Libération		RESDSR RESDSW
DELSP	21	4	F, P	Libération		
TRISS	22	4	F, P	RUN	TRISR	
TRISR	23	4	F, P	RUN	TRISS	
TRLIS	24	2	F	Mise en oeuvre		
COPEX	25	4	F, P	RUN	RECCOPEX	
RECCOPEX	26	2	F	RUN	COPEX	

Longueur des instructions

F fixe

V variable

Opérandes

I immédiat

P pointeur