Etude comparative des principaux langages de programmation

Olivier Lecarme

N° d'ordre

T H E S E

présentée à la Faculté des Sciences
de l'Université de GRENOBLE

pour obtenir

le titre de Docteur de Troisème Cycle

" Mathématiques Appliquées "

par

Olivier LECARME

ETUDE COMPARATIVE DES PRINCIPAUX LANGAGES DE PROGRAMMATION

-:-:-:-:-:-:-:-:-

Thèse soutenue le 28 Juin 1965 devant la Comission d'Examen :

MM. KUNTZMANN Président

VAUQUOIS

GASTINEL Examinateurs

BOLLIET

FACULTE DES SCIENCES

L I S T E D E S P R O F E S S E U R S

DOYENS HONORAIRES — M. FORTRAT P.

M. MORET L.

DOYEN — M. WEIL L.

PROFESSEURS TITULAIRES

MM.	NEEL L.	MAGNETISME ET PHYSIQUE DU SOLIDE
	DORIER A.	ZOOLOGIE
	HEILMANN R.	CHIMIE ORGANIQUE
	KRAVTCHENKO J.	MECANIQUE RATIONNELLE
	CHABAUTY C.	CALCUL DIFFERENTIEL ET INTEGRAL
	PARDE M.	POTAMOLOGIE
	BENOIT J.	RADIOELECTRICITE
	CHENE M.	CHIMIE PAPETIERE
	BESSON J.	ELECTROCHIMIE
	WEIL L.	THERMODYNAMIQUE
	FELICI N.	ELECTROSTATIQUE
	KUNTZMANN J.	MATHEMATIQUES APPLIQUEES
	BARBIER R.	GEOLOGIE APPLIQUEE
	SANTON L.	MECANIQUE DES FLUIDES
	OZENDA P.	BOTANIQUE
	FALLOT M.	PHYSIQUE INDUSTRIELLE
	GALVANI O.	MATHEMATIQUES
	MOUSSA A.	CHIMIE NUCLEAIRE
	TRAYNARD P.	CHIMIE
	SOUTIF M.	PHYSIQUE
	CRAYA A.	HYDRODYNAMIQUE
	REULOS R.	THEORIE DES CHAMPS
	AYANT Y.	PHYSIQUE APPROFONDIE
	GALISSOT F.	MATHEMATIQUES APPLIQUEES
Melle	LUTZ E.	MATHEMATIQUES
MM.	BLAMBERT M.	MATHEMATIQUES
	BOUCHEZ R.	PHYSIQUE NUCLEAIRE
	ILLIBOUTRY L.	GEOPHYSIQUE
	MICHEL R.	GEOLOGIE ET MINERALOGIE
	BONNIER E.	ELECTROCHIMIE
	DESSAUX	PHYSIQUE ANIMALE
	PILLET E.	ELECTROCHIMIE
	DEBELMAS J.	GEOLOGIE
	GERBER R.	MATHEMATIQUES
	PAUTHENET R.	ELECTROTECHNIQUE
	VAUQUOIS B.	MATHEMATIQUES APPLIQUEES
	SILBER R.	MECANIQUE DES FLUIDES
	MOUSSIEGT J.	ELECTRONIQUE
	BARBIER J.C.	PHYSIQUE
	KPSZUL J.L.	MATHEMATIQUES
	BUYLE-BODIN M.	ELECTRONIQUE

PROFESSEURS SANS CHAIRE

M.	LACASE A.	THERMODYNAMIQUE
Mme	KOFLER L.	BOTANIQUE
MM.	DREYFUS B.	THERMODYNAMIQUE
	VAILLANT F.	ZOOLOGIE ET HYDROBIOLOGIE
	GIRAUD P.	GEOLOGIE
	GIDON P.	GEOLOGIE ET MINERALOGIE
	ARNAUD P.	CHIMIE
	PERRET R.	SERVOMECANISMES
Mme	LUMER L.	MATHEMATIQUES
Mme	BARBIER M.J.	ELECTROCHIMIE
Mme	SOUTIF J.	PHYSIQUE
MM.	BRISSONNEAU P.	PHYSIQUE
	COHEN J.	ELECTROCHIMIE
	DEPASSEL R.	MECANIQUE
	GASTINEL N.	MATHEMATIQUES APPLIQUEES

PROFESSEURS ASSOCIES

MM.	LUMER G.	MATHEMATIQUES
	HIGUCHI	BIOSYNTHESE DE LA CELLULOSE
	WAGNER	BOTANIQUE

MAITRES DE CONFERENCES

MM.	ROBERT A.	CHIMIE PAPETIERE
	ANGLES D'AURIAC	MECANIQUE DES FLUIDES
	BIAREZ J.P.	MECANIQUE PHYSIQUE
	COUMES A.	ELECTRONIQUE
	DODU J.	MECANIQUE DES FLUIDES
	DUCROS PL	MINERALOGIE ET CRISTALLOGRAPHIE
	CLENAT P.	CHIMIE
	HACQUES G.	CALCUL NUMERIQUE
	LANCIA R.	PHYSIQUE AUTOMATIQUE
	PEBAY-PEROULA	PHYSIQUE
	KAHANE	PHYSIQUE GENERALE
	DOLIQUE	ELECTRONIQUE
Mme	KAHANE J.	PHYSIQUE
MM.	DEGRANGE C.	ZOOLOGIE
	GAGNAIRE D.	CHIMIE PAPETIERE
	RASSAT A.	CHIMIE SYSTEMATIQUE
	KLEIN J.	MATHEMATIQUES
	POULOUJADOFF M.	ELECTROTECHNIQUE
	DEPOMMIER P.	PHYSIQUE NUCLEAIRE
	DEPORTES C.	CHIMIE
	BARRA J.	MATHEMATIQUES APPLIQUEES
Mme	BOUCHE L.	MATHEMATIQUES
MM.	PERRIAUX J.	GEOLOGIE
	SARROT-REYNAULD	GEOLOGIE
	CAUQUIS G.	CHIMIE GENERALE
	LABRE A.	BOTANIQUE
	BONNET G.	PHYSIQUE GENERALE
	BARNOUD F.	BIOSYNTHESE DE LA CELLULOSE
Mme	BONNIER M.J.	CHIMIE
	CAUBET	MATHEMATIQUES APPLIQUEES
	BERTRANDIAS	MATHEMATIQUES APPLIQUEES

MAITRES DE CONFERENCES ASSOCIES

MM.	ISHIKAWA Y.	MAGNETISME
	QUATTROPANI	THERMODYNAMIQUE

Je tiens à exprimer ma profonde reconnaissance à

Monsieur le Professeur KUNTZMANN, directeur de l'Institut de Mathématiques Appliquées de Grenoble, qui a bien voulu me faire l'honneur de présider le jury;

Messieurs les Professeurs VAUQUOIS et GASTINEL, qui ont accepté de faire partie de ce jury;

Monsieur BOLLIET, Ingénieur au C.N.R.S., qui a donné l'idée de cette thèse et a su l'orienter vers les résultats les plus intéressants;

à tous ceux qui m'ont aidé en m'offrant des conseils en programmation, en particulier AUROUX, CLEEMANN, Du MASLE, Madame GAREYTE et Madame GUELTON, LAGARDE, TREHEL, et j'en passe;

à tous ceux qui m'on aidé en me demandant des conseils en programmation, en particulier BARDOULAT, CLEEMANN, Du MASLE, DUNG, LAGARDE, Monsieur J. LABORDE, SABONNADIERE, TREHEL, WOLF, les étudiants du D.E.A. de mathématiques appliquées et du certificat de T.M.P., et j'en passe encore plus;

à ceux qui ont été mes voisins de bureau pendant mon travail, c'est-à-dire TREHEL, Du MASLE, CLEEMANN, Madame GAREYTE et COLMERAUER, LAGARDE, et DUNG;

à HYACINTHE, qui m'a donné l'occasion de faire dans un cas précis une fructueuse comparaison entre Algol et Fortran;

à toutes les personnes du service de perforation qui ont travaillé pour mes programmes innombrables et trop souvent mal écrits;

à tous les opérateurs de l'I.B.M. 7044, à qui ces mêmes programmes ont causé des ennuis sans nombre;

à Mademoiselle LAFOSSE, Messieurs MOUNET, VIDAL et DEBARGUES, qui ont rendu possible la réalisation matérielle de cet ouvrage;

enfin, à tous ceux du Laboratoire de calcul que je ne cite pas ici et qui, faisant de la programmation, se sont intéressé peu ou prou à ce travail.

-:-:-:-:-:-:-:-:-

I N T R O D U C T I O N

Première partie : Caractéristiques principales des langages étudiés.

A Langages orientés vers l'analyse numérique.

1 - Fortran

2 - Algol

3 - Jovial

4 - Neliac

5 - PL/I

B Langages orientés vers des traitements spécialisés.

1 - Cobol

2 - Apt

3 - IPL

4 - Lisp

5 - Slip

6 - Comit

7 - Snobol

Deuxième partie : Etude comparative des langages.

A Syntaxe

1 - Structure

2 - Rigueur

3 - Restrictions

4 - Ambiguités

5 - Souplesses

B Structure

1 - Déclarations

2 - Type des données

3 - Arrangements des données

4 - Affectations et conversions

C Utilisation

1 - Opérations arithmétiques

2 - Opérations logiques

3 - Manipulations d'ensembles

4 - Déroulement du contrôle

5 - Sous-programmes

6 - Facilités diverses

Conclusions

Troisième partie : Comparaison entre quelques programmes

1 - Calcul différentiel

2 - Calcul matriciel

3 - Traitement de fichiers

4 - Traitement de chaînes

5 - Récursivité

6 - Programmes types de chaque langage

ANNEXES -

Table et graphiques de comparaison rapide

Bibliographie

Liste non exhaustive des compilateurs

Cartes syntaxiques de Fortran IV, Neliac, Lisp et Comit.

-:-:-:-:-:-:-:-

ETUDE COMPARATIVE DES PRINCIPAUX LANGAGES DE PROGRAMMATION

"......aussi la nomma-t-on Babel, car c'est là
que Yahvé confondit le langage de tous les
habitants de la terre pour qu'ils ne s'entendent
plus les uns les autres ....."

La programmation sur les premiers ordinateurs se faisait directement dans leur langage numérique interne, ce qui était d'une difficulté désespérante. La traduction des instructions "en clair" permit d'éclaircir un peu le problème, et surtout l'introduction des langages d'assemblage aboutit à des programmes d'écriture moins ésotérique et surtout plus générale. Leurs inconvénients principaux sont d'une part l'écriture lourde et sans rapports apparents avec les conventions mathématiques ou logiques habituelles, d'autre part le fait que le passage d'une machine à l'autre impose la reprise complète de la conception et de la réalisation de tous les programmes.

Les langages de programmation à haut niveau ont été créés à partir de 1956 afin d'obtenir une écriture des programmes proche du langage parlé tout en restant rigoureuse, et assez indépendante des machines utilisées. Depuis cette date, ils se sont multipliés de façon très rapide, d'une part pour leur appliquer des perfectionnements et des généralisations, d'autre part pour les adapter plus ou moins à de nouvelles possibilités des machines, enfin pour leur permettre de traiter de nouveaux problèmes. Chaque année depuis cette date, une vingtaine de langages en moyenne ont vu naissance; certains ont disparu rapidement, mais la quantité de langages qui existent et sont utilisés à l'heure actuelle reste très impressionnante.

Pour choisir les douze langages que j'étudie dans ce travail, il m'a donc fallu déterminer certains critères. Ceux-ci sont forcément discutables, et l'on pourra s'étonner de voir figurer dans cette étude certains langages et non pas certains autres. J'ai essayé de choisir, parmi les plus utilisés des langages, ceux qui présentaient le plus de points intéressants et originaux, et ceux qui étaient

les plus représentatifs de leur famille. Je n'étudie qu'un langage spécifiquement commercial, parce qu'il constitue l'aboutissement de tous les autres. J'étudie au contraire trois langages qui traitent des listes et deux langages qui traitent des chaînes parce qu'ils sont extrêmement différents. Parmi ces douze langages figurent deux cas particuliers: PL/I, qui n'est encore utilisé effectivement et couramment par personne, et Slip qui ne constitue pas en lui-même un véritable langage. Parmi les manquants, CPL est peut-être celui dont l'absence se fait le plus sentir, mais son état définitif et la documentation complète ne datent que de mai 1965.

-:-:-:-:-:-:-:-

P R E M I E R E P A R T I E

CARACTERISTIQUES PRINCIPALES DES LANGAGES ETUDIES.

CARACTERISTIQUES PRINCIPALES DES LANGAGES ETUDIES

CHAPITRE A - LANGAGES ORIENTES VERS L'ANALYSE NUMERIQUE.

1 - Fortran.

1°) Fortran est le premier en date des langages de programmation à haut niveau, puisqu'il date de 1956. Il a derrière lui une histoire déjà assez longue, il a beaucoup évolué depuis son état initial, et c'est actuellement le plus utilisé, et de très loin, de tous les langages de programmation; c'est dire que son âge d'une part et sa puissance d'autre part lui donnent une place très particulière parmi les langages que j'étudierai. Il existe plusieurs Fortran : Fortran II quoique ancien, est encore très utilisé car les habitudes sont longues à perdre et les changements très coûteux; plusieurs constructeurs de machines ont défini leur Fortran à eux, pour l'adapter à des calculateurs particuliers ou pour combler de la façon qu'ils entendaient certaines lacunes particulièrement graves; quoique ce ne soit pas le Fortran le plus évolué, j'étudierai ici Fortran IV, d'une part parce que c'est celui dans lequel j'ai pu le plus programmer, d'autre part, parce que c'est celui qui a le plus d'avenir (Fortran II est utilisé sur 704, 709, 7090 et 7094, Fortran IV sur 7040, 7044 et Système 360).

2°) Le but de Fortran est avant tout, et presque uniquement, l'analyse numérique, quoique faute de mieux on ait tenté de l'utiliser pour à peu près n'importe quoi. Il peut donc traiter tous les calculs arithmétiques, mais il rencontre des difficultés avec le type des variables utilisées, car il ne permet les expressions mixtes que de façon très limitée. Il peut faire aussi du calcul vectoriel ou matriciel, mais se caractérise dans ce domaine par des restrictions très nombreuses et souvent étroites. Tout dans la définition du langage

est orienté vers la génération d'un programme en langage machine qui soit très optimisé, et c'est ce qui explique les innombrables restrictions de définition.

3°) Un programme Fortran est une suite d'instructions, écrites sur cartes perforées, avec des étiquettes numériques placées dans une zone spéciale. Les séparateurs sont très peu nombreux, souvent absents, le même séparateur peut jouer un grand nombre de rôles différents, en particulier les parenthèses et la virgule. La structure du programme est complètement linéaire, et généralement assez peu lisible. Les déclarations sont presque toujours implicites; il n'y a aucune dynamique dans l'allocation de mémoires ou dans la structure du programme, toute récursivité quelle qu'elle soit est impossible sauf programmation explicite.

4°) Fortran peut traiter des variables simples et des tableaux de une à trois dimensions, la borne inférieure d'indice étant toujours 1; ces variables peuvent être de type entier, réel, complexe, logique et double précision. Il reconnait cinq opérateurs arithmétiques (+, -, ×, /, **), mais le mélange des types dans une expression est presque toujours interdit, de même que dans les comparaisons au moyen des opérateurs de relation. Seuls trois opérateurs logiques peuvent être utilisés (.NØT., .AND., .ØR.); l'instruction d'affectation est toujours simple. Le transfert de contrôle peut être inconditionnel, calculé par recherche dans une table, ou imposé antérieurement. L'essai d'une condition ne peut commander qu'une instruction. Les boucles itératives ont une structure très rigide et des restrictions souvent inexplicables si l'on oublie les procédés de compilation. Il est possible de définir des fonctions "ouvertes", insérées dans le programme à la place de chaque appel, ou des sous-programmes ou fonctions extérieurs compilés séparément, sans aucune imbrication et sans aucun contrôle du type des paramètres. Des sous-programmes peuvent communiquer entre eux au

moyen d'une zone de mémoires commune, certains peuvent être écrits en langage machine. Les opérations d'entrée-sortie se font d'une seule façon, au moyen d'un modèle dirigeant la transmission, sans aucun contrôle de la simultanéité et sans aucun accès aux fichiers eux-mêmes.

2 - Algol.

1°) Algol est issu des travaux de plusieurs conférences et surtout d'un comité international en grande partie européen. Sa première définition date de 1958, son état actuel de 1962. Il est utilisé surtout en Europe pour la programmation, mais dans le monde entier pour la communication d'algorithmes,et aucun langage ne peut pour l'instant lutter avec lui sur ce dernier plan. La version du langage que j'étudierai est celle qui est définie par le rapport "Algol 60" (référence 1.4). Aux termes mêmes de ce rapport, Algol est capable d'exprimer "une classe très large de processus numériques"; c'est dire que son but est l'analyse numérique, même si là encore on l'a utilisé et on l'utilise pour tout faire. Les deux idées maîtresses qui semblent avoir guidé sa conception sont la généralité et la rigueur; nous verrons par la suite ce qui en résulte, disons tout de suite qu'un programme Algol est généralement facile à écrire, mais que le programme objet est très difficile à optimiser.

2°) Un programme Algol est constitué d'un bloc, c'est-à-dire d'un ensemble de déclarations puis d'instructions placées entre les symboles début et fin. Sa structure est indépendante du support sur lequel il est écrit, et très fortement "parenthésée", le mot "parenthèse" étant pris au sens le plus large. Il utilise un grand nombre de séparateurs très différenciés; tout identificateur utilisé doit être déclaré; la structure de blocs autorise d'une part, l'écriture de plusieurs parties du programme de manière indépendante (au point de vue programmation), d'autre part, l'optimisa-

tion automatique de l'utilisation des mémoires par l'allocation dynamique (au point de vue exécution); la récursivité dans l'emploi des procédures ou sous-programmes est explicitement prévue.

3°) Algol peut traiter des variables simples ou des tableaux à nombre quelconque de dimensions et bornes d'indices également quelconques; ces variables peuvent être de type entier, réel ou booléen; les expressions arithmétiques utilisent six opérateurs ($+$, $-$, $\times$, $/$, $\div$, $\uparrow$), et peuvent mêler sans restriction les types entier et réel. Les expressions booléennes utilisent cinq opérateurs ($\neg$, $\wedge$, $\vee$, $\supset$, $\equiv$,); l'instruction d'affectation peut être multiple. Le transfert de contrôle peut être inconditionnel ou commandé par un aiguillage, l'instruction conditionnelle est de forme très générale et peut commander n'importe quoi par l'intermédiaire de l'instruction composée. Les boucles itératives sont de structure très souple et peuvent être imbriquées de façon quelconque. On peut définir des fonctions ou des sous-programmes par le moyen de déclarations de procédures, qui ont les propriétés de déclarations normales au point de vue de la portée. Elles ne peuvent être extérieures au programme, et n'ont qu'un point d'entrée et un point de sortie. Les procédés d'entrée-sortie ne sont pas prévus dans la définition du langage, ce qui est une de ses plus grosses lacunes, et chaque compilateur a défini ses propres fonctions de manière totalement anarchique. Ce n'est que très tardivement (Mars 1964) que le groupe Algol a défini quelques procédures d'entrée-sortie.

3 - Jovial.

1°) Le langage Jovial a été construit à partir de 1959 par la "System Development Corporation", en s'inspirant fortement des travaux du comité Algol. Il est utilisé systématiquement par la Marine américaine, mais assez peu par d'autres utilisateurs, son but principal est le traitement des problèmes de commande et de contrôle. Tout en utilisant la plupart des acquisitions d'Algol, il développe donc particulièrement les points qui concernent plus étroitement la

définition et le traitement de variables de types très divers.

2°) Un programme Jovial est formé d'une série de déclarations puis d'ordres étiquetables; sa structure est indépendante du support, très parenthésée, avec l'utilisation des blocs et des instructions composées. Il n'y a cependant pas d'allocation dynamique des mémoires. Les séparateurs sont nombreux mais non différenciés des mots normaux, et l'espace joue le rôle de séparateur. Toute variable utilisée doit être déclarée, sauf les indices.

3°) Un programme Jovial peut traiter des variables simples, des tableaux à nombre de dimensions quelconques mais à borne inférieure d'indices toujours égale à zéro, des variables duales, des structures ou des tables qui peuvent se recopier ou se superposer. Il est possible de traiter des variables entières, réelles de longueur variable avec virgule fixe ou flottante, booléenne complexes, octales, alphanumériques, littérales et symboliques ("variables d'état"). Cinq opérateurs arithmétiques (+, -, * , /, **) et trois opérateurs booléens (NØT, AND, ØR) sont reconnus; les opérateurs de relation peuvent porter sur des variables numériques, avec mélange quelconque des types, littérales ou symboliques, plusieurs relations pouvant être placées à la suite (A < B < C). Il existe une instruction d'affectation simple et une instruction d'échange. Le transfert du contrôle se fait de façon inconditionnelle ou par aiguillage. La boucle itérative est de structure très souple et générale; une instruction spéciale permet de la reprendre avant sa fin logique. Des modificateurs fonctionnels facilitent les recherches dans les tables, les manipulations de caractères et de chiffres binaires, le passage des nombres flottants aux nombres fixes ou entiers. L'instruction conditionnelle est du type le plus général, elle commande une instruction composée. Les procédures ont toutes les propriétés de celles d'Algol, avec en plus la possibilité de plusieurs points de sortie; des sous-programmes "fermés" peuvent être placés en n'importe quel point d'un programme; dans l'appel de ces procédures et sous-programmes on doit spécifier quels sont les paramètres de sortie. Les opéra-

tions d'entrée-sortie se font par traitement de fichiers et par ordres de lecture ou écriture dirigés par les données.

4 - Neliac -

1°) Le langage Neliac est un dialecte d'Algol, c'est-à-dire qu'il a été conçu comme une version simplifiée de ce langage; la première définition date de 1958, au Laboratoire électronique de la Marine américaine. Depuis, un grand nombre de Neliacs plus ou moins évolués ont vu naissance dans divers laboratoires et pour diverses machines, mais il n'en existe pas actuellement une version normalisée et reconnue. J'essaierai d'étudier ici un Neliac fictif qui ne soit pas trop simpliste mais pas trop proche d'Algol. Neliac a été conçu comme un langage orienté vers l'analyse numérique, mais en tenant compte des problèmes de commande et de contrôle, et surtout des problèmes de compilation; en effet, tous les compilateurs Neliac ont été écrits en Neliac, et en fait plus des trois quarts des programmes écrits dans ce langage sont des compilateurs. Ceci indique que la compilation se fera très rapidement, les modifications à la définition du langage également, mais que les caractéristiques du programme résultant n'ont été étudiées que d'assez loin.

2°) Un programme Neliac a la structure déjà vue pour Algol et Jovial de déclarations suivies d'instructions. Les délimiteurs sont très nombreux, et toujours des "caractères spéciaux"; l'allocation dynamique de mémoires, la structure de blocs et la récursivité ne sont pas prévues. Tout symbole utilisé doit être déclaré, sauf les indices. Les variables peuvent être simples ou en tableaux, à dimensions et bornes d'indices limitées, de types entier, réel ou logique. Les expressions arithmétiques interdisent le mélange des types et utilisent les cinq opérateurs habituels. Les intructions conditionnelles et les boucles itératives ont une syntaxe simple mais limitée; les procédures sont définies et utilisées comme en Algol. Les problèmes d'entrée-sortie sont abordés de façon différente par chacune des définitions du langage.

5 - PL/I.

1°) Seul de tous les langages que j'étudierai, PL/I n'est pas encore utilisé de façon normale (la Boeing Corporation l'utilise à titre expérimental depuis quelques mois). Il est à présent à peu près complètement défini, mais aucun compilateur d'exploitation normale n'est encore terminé. Il s'agit d'un langage conçu spécifiquement pour les nouvelles machines IBM du système 360, par un groupe de personnes de la compagnie IBM et de l'organisation Share. Le but explicite est de remplacer tous les langages de programmation existants, y compris le langage machine. Un programme PL/I doit donc en principe pouvoir faire tout ce qu'il est possible de faire avec une machine, étant admis qu'il est cependant plus spécialement orienté vers les problèmes d'analyse numérique et de gestion, mais en tenant compte des problèmes de multi-programmation et de temps réel.

2°) Un programme est une suite d'instructions séparées par des points-virgules, certaines instructions devant générer un fragment du programme résultant, d'autres devant être traitées au moment même de la compilation, c'est-à-dire destinées exclusivement au compilateur. La structure de blocs et d'instructions composées est utilisée très largement, mais avec quelques limitations pourtant. Les séparateurs sont peu nombreux et très peu différenciés, les mots de contrôle sont très nombreux mais ne se distinguent des mots normaux que par l'endroit où ils apparaissent; le blanc est un séparateur. Les déclarations, l'allocation dynamique des mémoires, la récursivité peuvent toujours se faire soit implicitement, soit explicitement. Toute construction ayant un sens est interprétée par le compilateur, c'st-à-dire considérée comme syntaxiquement correcte.

3°) PL/I peut traiter des variables simples, des tableaux de la façon la plus générale, et des structures. Ces variables peuvent être arithmétiques (avec le type réel ou complexe, le mode fixe ou flottant, la représentation binaire ou décimale), logiques, ou être des chaînes ou des étiquettes; leur longueur est toujours variable.

Les expressions arithmétiques utilisent les cinq opérateurs normaux, et peuvent mêler indistinctement tous les types quels qu'ils soient. Il existe pour les chaînes un opérateur de concaténation; l instruction d'affectation peut être multiple. Le transfert de contrôle peut être inconditionnel, calculé, gouverné par aiguillage ou par tableau d'étiquettes. L'instruction conditionnelle et la boucle itérative sont de forme très générale, toutes les constructions et utilisations sont permises. Les procédures ou sous-programmes peuvent être compilés séparément, avoir des points d'entrée ou de sortie multiples, avoir des listes de paramètres modifiables; l'appel récursif est permis. Les opérations d'entrée-sortie peuvent se faire de trois façons différentes, suivant que le traitement est dirigé par un modèle, par les données ou par l'ordre lui-même. Les problèmes de multi-programmation et de temps réel sont explicitement prévus; l'allocation des mémoires peut être statique comme en Fortran, automatique comme en Algol, ou contrôlée par le programmeur. Il est possible, parmi les instructions destinées au compilateur, de définir des macros-procédures. Presque tout ce qui n'est pas spécifié est considéré par le compilateur comme implicitement défini, ce qui est à la fois très commode et très dangereux.

CHAPITRE B - LANGAGES ORIENTES VERS DES TRAITEMENTS SPECIALISES.

1 - Cobol.

1°) Le langage Cobol est né en 1959 des travaux du comité CODASYL composé de constructeurs de machine et d'utilisateurs à l'initiative du "Department of Defense" américain. Sa définition a été améliorée en 1960; le langage est à présent mis en place systématiquement sur toute machine destinée à être utilisée par des organismes gouvernementaux. Grâce au gros appuis dont il a ainsi disposé, c'est maintenant de très loin le plus utilisé des langages de programmation orientés vers le traitement des problèmes de gestion. Sa conception a été fortement guidée par l'idée d'obtenir des programmes d'une lecture très facile, et c'est de là que viennent la plupart de ses caractéristiques.

2°) Un programme Cobol comprend quatre grandes divisions subdivisées en sections et en paragraphes : identification, environnement, données et procédures. Il ressemble plus ou moins à la langue anglaise, ce qui pose des problèmes étant donné qu'il s'agit d'une langue fondamentalement ambiguë; en conséquence, les mots réservés sont extrêment nombreux, le blanc est significatif, les ambiguités et les interdictions arbitraires sont légion. Tout ce qui est utilisé doit être déclaré et complètement décrit dans la division données.

3°) Un programme Cobol peut traiter des variables simples, des tableaux à trois dimensions au maximum, des structures hiérarchisées jusqu'à cinquante niveaux et qui peuvent elles-mêmes contenir des tableaux. Ces variables sont de types assez mal définis car prévus uniquement pour des machines à caractères. Les formules arithmétiques utilisent les cinq opérateurs normaux, d'une façon extrêmement lourde; les trois opérateurs booléens NØT, AND et ØR sont d'une manipulation plus souple, grâce aux mises en facteur implicites, mais d'une syntaxe très arbitraire, ainsi que les relations. L'instruction d'affectation peut être multiple; le transfert peut être inconditionnel, ou contrôlé par des aiguillages manoeuvrés à distance. L'instruc-

tion conditionnelle est de forme très générale, et assez souple grâce à l'utilisation des noms-conditions. La boucle itérative est également d'une utilisation très souple en ce qui concerne le test d'arrêt. Il n'existe pas de procédures à proprement parler, mais des sous-programmes sans paramètres, dont la syntaxe et la sémantique sont très mal définies. L'accent a été mis plus particulièrement sur les opérations d'entrée-sortie, dont le travail est dirigé par les données; le contrôle des fichiers est donc très complet. La possibilité d'appel de sous-programmes extérieurs est explicitement prévue, mais réalisée de façons très variables, ainsi que l'insertion de séquences de programmes écrites dans un autre langage. Les données alphanumériques peuvent être traitées exactement de la même manière que les données numériques.

2 - Apt.

1°) Le langage Apt est né en 1955 au M.I.T. , mais sa véritable définition et son utilisation datent de 1958. Il continue actuellement d'évoluer vers une plus grande généralité, sous le contrôle de l'IIT (Illinois Institute of Technology). C'est actuellement, et de très loin, le plus utilisé des langages de son type, quoiqu'il ne fasse que commencer de s'implanter en Europe. Sa destination le range complètement à part parmi les langages que j'étudierai, puisqu'il est destiné à la commande numérique des machines-outils. L'effet d'un programme Apt sera donc de donner une table de coordonnées et d'ordres destinés à cette commande. Les instructions du programme sont de deux catégories complètement différentes : les instructions arithmétiques, dont l'effet ne joue que pendant l'éxécution du programme, et les instructions géométriques, qui auront un effet au moment de l'application des résultats du programme. De même l'exécution du programme se fait en deux phases, la première calculant des positions géométriques, et la seconde utilisant ces positions pour calculer les ordres à donner à la machine-outil: cette seconde phase est évidemment complètement dépendante de la machine à laquelle on applique le travail.

2°) Un programme Apt est une suite d'instructions, écrites sur cartes perforées, avec des étiquettes placées dans une zone spéciale. Ces instructions peuvent être des définitions géométriques, des ordres arithmétiques ou des ordres géométriques. Les séparateurs sont peu nombreux et d'une utilisation assez arbitraire, en particulier la barre de fraction (/) et la virgule. La structure du programme est complètement linéaire, les mots réservés sont extrêmement nombreux mais le plus souvent différenciés par leur place. Les conventions d'écriture sont presque toujours celles de Fortran, avec les inconvénients inhérents.

3°) Les variables que traite un programme Apt sont divisées dans les deux catégories arithmétique et géométrique. Les variables arithmétiques sont toujours réelles et de structure fixe, un programme les utilise d'ailleurs assez peu normalement. Les variables géométriques sont des symboles qui peuvent désigner des points, des courbes ou des surfaces; ils sont définis avant leur utilisation par des pseudos-instructions d'affectation qui peuvent avoir un grand nombre de formes diverses, et qui peuvent d'ailleurs utiliser des variables géométriques définies antérieurement. Les ordres arithmétiques utilisent les variables du même type pour des calculs du même type que ceux de Fortran. Les ordres géométriques sont des ordres de mise en place de l'outil puis de déplacement, les conditions de ce déplacement étant très souples: le long d'une courbe, suivant l'intersection de deux surfaces, jusqu'à un point, jusqu'à la rencontre d'une courbe ou d'une surface, etc..... Les transferts peuvent être des transferts de contrôle (arithmétiques) ou des transferts d'outils (géométrique), ils sont inconditionnels ou soumis à des conditions extrêmement simples. Il n'est évidemment pas question de calcul logique, d'entrées-sorties, mais le manque de boucle itérative est assez grave étant donnée la nature des problèmes traités.

3 - IPL.

1°) Le langage IPL a été créé en 1956 à la Rand Corporation par Newell, Shaw et Simon. La première version effective, IPL-3, date de 1957, cependant que la version actuelle, IPL-5, considérable-

ment augmentée par rapport aux précédentes, date de 1958. Les buts primitifs du langage IPL sont les études sur la programmation heuristique; malgré son âge, c'est aujourd'hui le plus utilisé des langages qui utilisent les listes, à cause de ses qualités propres et de la facilité d'écriture du système interpréteur. Le langage IPL est le langage d'une machine fictive dont les mémoires seraient toutes des piles, et qui utiliserait abondamment les micro-programmes, avec un assembleur peu perfectionné, qui n'utilise qu'un symbolisme extrêmement réduit, et qui constitue le langage IPL lui-même.

2°) La mémoire de la machine IPL est divisée en trois parties: la mémoire centrale et les mémoires auxiliaires, rapide et lente. Un mot machine comprend quatre parties: P, le code opération, Q, le désignateur, sont des chiffres octaux; le symbole et le lien sont de la longueur voulue pour représenter une adresse. Q permet un adressage indirect à zéro, un ou deux niveaux, on l'appel de listes ou de sous-programmes rangés dans des mémoires auxiliaires. P représente les huit opérations de base de la machine IPL : exécuter, en empilant l'adresse de retour dans le registre d'adresse; empiler un symbole dans le registre de communication; faire l'opération inverse; restaurer un registre, c'est-à-dire remonter la pile correspondante d'un niveau; préserver un registre (opération inverse); copier un symbole dans le registre de communication; faire l'opération inverse; tester un registre spécial.

3°) Les opérations sur les listes et les opérations d'entrée-sortie sont réalises par des micro ou macro-programmes, ainsi que les opérations arithmétiques. Les types existant sont l'entier, le flottant et l'octal, les opérations de mode mixte sont permises, mais toutes se font en rangeant les paramètres de la fonction et en retrouvant les résultats dans le registre de communication, ce qui équivaut à une notation polonaise post-fixée. Les entrées-sorties sont assez

complètes, puisqu'elles comprennent les traitements de caractères, de termes numériques et alphanumériques, de symboles et de listes dans la notation IPL. Les ordres conditionnels se font par appel à un registre de test, ce qui est souvent notoirement insuffisant. Programmes et données ont exactement même structure et même traitement, ce qui permet très simplement l'auto-modification d'un programme.

4 - Lisp.

1°) Le langage Lisp a été créé en 1958 au M.I.T., par une équipe dirigée par J. Mac Carthy. La version actuelle, nommée Lisp 1.5, date de 1960, et ajoute à la version initiale de très nombreux perfectionnements tendant à en faire un véritable langage de programmation d'utilisation commode. Il a été conçu dans le but d'étudier les problèmes de programmation heuristique, de logique et de compilation. Sa caractéristique la plus marquante est que le langage, en lui-même, constitue un système formel simple et complet.

2°) Le langage Lisp est un ensemble de fonctions récursives d'expressions symboliques, construit à partir d'un très petit nombre de fonctions de base. Une de ces fonctions sera un compilateur, une autre un assembleur, une autre l'interpréteur d'un langage proche d'Algol, une autre enfin, la plus importante, servira à évaluer de façon complètement générale toutes ces fonctions. Le système tout entier est organisé de façon complètement dynamique et récursive; la notation utilisée est toujours la notation fonctionnelle, c'est-à-dire une notation polonaise préfixée. Sauf cas très particuliers, les identificateurs utilisés ne sont pas déclarés.

3°) Les données traitées par un programme sont des expressions symboliques organisées sous forme de listes, et qui représentent des chaînes, des structures, des fonctions, des variables, des tableaux, etc....Les opérations arithmétiques, les instructions d'affectations, le transfert de contrôle, l'instruction conditionnelle, le traitement des listes et des chaînes, les opérations d'entrée sortie, se font toujours par le moyen de fonctions ou de pseudos-

fonctions. Les opérations d'entrée-sortie se font suivant des modèles standards, ou par listes complètes, ou caractère par caractère. La mémoire de la machine est divisée en huit parties: l'interprète, le chargeur, le compilateur et l'assembleur constituent le système proprement dit; la liste de l'espace libre, la liste de récursivité, l'espace des programmes binaires et l'espace des mots pleins se partagent le reste. De même qu'en IPL, le programme et ce qu'il traite sont de même forme, et même souvent impossibles à distinguer, d'où la possibilité d'auto-modification ou d'auto-construction d'un programme.

5 - Slip.

1°) Inspiré des travaux de Gelertner sur FLPL et de Perlis sur les "Threaded lists", Joseph Weizenbaum a défini en 1963 un certain nombre de fonctions et de sous-programmes destinés à être placés dans un langage de programmation déjà existant pour lui permettre de traiter des listes, qui ont la particularité d'être symétriques et non pas à sens unique comme celles de Lisp et d'IPL. Slip n'est donc pas un langage, mais un ensemble de procédures plus ou moins complexes, construites à partir d'une dizaine de fonctions de base qui, à la différence des autres, ne sont pas écrites dans le langage hôte mais dans le langage d'une machine particulière. Ces procédures sont donc conçues pour être indépendantes à la fois de la machine et du langage hôte. Si le premier point est vérifié, il n'existe pas à ma connaissance de tentative de placer les procédures du système Slip dans un autre langage que Fortran.

2°) Slip utilise des listes d'un type original, dont l'unité de base est une paire de mots : le premier contient deux liens et un marqueur, le second contient une donnée, qui peut être à son tour constituée de deux liens. A l'intérieur d'une liste, une paire pointe donc à la fois sur la paire précédente et sur la suivante, l'en-tête

d'une liste ayant pour "précédent" le dernier mot de la liste, et celui-ci ayant pour "suivant" l'en-tête. Une sous-liste a son nom dans les deux liens du deuxième mot d'une paire de la liste principale. L'en-tête d'une liste contient le nom d'une liste descriptive de style IPL, et le nombre de listes dont cette liste est une sous-liste, ce qui permet le "ramassage d'ordures" automatique. Une entité spéciale, le lecteur, permet par le biais d'empilages et de dépilages d'évoluer le long d'une structure pour la consulter. Slip comprend la plupart des fonctions habituelles sur les listes, plus celles qui utilisent le lecteur et celles qui tiennent compte de la symétrie. La récursivité s'obtient par l'intermédiaire de deux fonctions qui empilent ou dépilent l'adresse de retour et les paramètres. De plus, on peut évidemment utiliser toutes les possibilités du langage hôte, en tenant compte de ses restrictions.

6 - Comit.

1°) Le langage Comit a été créé au M.I.T. en 1961, sou direction de V. Yngve, dans le but d'étudier certains problèmes de traduction automatique par le traitement des chaînes. Il a été et reste très largement utilisé au M.I.T., mais peu en dehors, en particulier à cause de la rareté des compilateurs, et aussi parceque son âge commence à se faire sentir. Une nouvelle version améliorée, Comit II, est annoncée depuis un certain temps. Le langage Comit est très étroitement lié au traitement des chaînes alphabétiques et aux problèmes de recherche dans des dictionnaires; une de ses caractéristiques les plus intéressantes est qu'il est très facile à apprendre, même par un non-mathématicien.

2°) Un programme Comit est formé d'une suite de règles de grammaire, qui sont des instructions portant sur des chaînes, et de règles de liste, qui font des recherches dans des dictionnaires. Ces règles travaillent sur des chaînes de caractères, munies d'une liste descriptive. Les seuls symboles sont les noms, qui ont une utilisation restreinte, ne se réfèrent pas à des chaînes et ne sont pas

considérés comme des chaînes, du moins au niveau de l'exécution du programme.

3°) Un programme dispose de 128 chaînes de constituants. L'une est l'espace de travail, accessible en détail, les autres sont les rayons, accessibles en bloc ou en séquence, et traités comme des piles ou comme des files d'attente; ils peuvent à tout moment échanger leur rôle avec l'espace de travail. Une règle de grammaire recherche une structure dans l'espace de travail, pour la modifier, effectuer des compressions, des opérations d'entrée-sortie, des interversions avec les rayons. Elle traite comme unité d'information le constituant, son indice numérique, ses indices logiques et leur valeur; on peut insérer un constituant, le modifier, le remplacer, le déplacer, le supprimer, travailler sur ses divers indices, en liaison avec un aiguilleur qui permet des choix contrôlés ou aléatoires. Le transfert de contrôle est lié à la reconnaissance de structure; il peut se faire directement, ou par un adressage indirect particulier, ou encore par aiguillage, celui-ci pouvant être aléatoire mais non directement pondéré. Il existe très peu de possibilités de traitement autres que celui des chaînes; l'arithmétique est très lourde et très pauvre, elle ne connaît que les entiers positifs; les opérations d'entrée-sortie sont peut nombreuses mais bien adaptées au langage.

7 - Snobol.

1°) Le langage Snobol a été créé aux laboratoires de la Bell Telephone par Farber, Griswald et Polensky en 1963. Il est destiné et restreint à la manipulation des chaînes; il s'inspire de Comit et de SCL, mais avec beaucoup d'idées neuves, et avec l'intention arrêtée de créer un langage donc l'écriture soit simple et ne réclame que peu d'explications. Il n'est connu pour le moment que dans un état provisoire, car de nombreuses adjonctions sont prévues.

2°) Snobol présente de grandes ressemblances avec Comit, et ne s'en cache d'ailleurs pas. Un programme Snobol est constitué d'une

suite de règles qui recherchent des structures, les modifient et leur donnent des noms. Ce dernier point constitue la grande nouveauté par rapport à Comit et le caractère le plus intéressant du langage. Cependant le langage est encore loin d'être complet, et beaucoup reste à faire.

3°) Le seul objet que traite Snobol est la chaîne de caractères. Si une chaîne a un nom, celui-ci est une autre chaîne; ceci permet à une chaîne de contenir le nom d'une autre chaîne, ce qui constitue une sorte d'adressage indirect, de longueur indéfinie. Les opérations élémentaires sont la création de chaînes, la concaténation, l'insertion, la suppression, la modification, l'extraction ou la simple recherche; elles traitent les chaînes soit explicitement soit par leur nom. Le transfert de contrôle peut être lié au succès ou à l'échec de la recherche de structure. L'arithmétique est, au moins pour le moment, extrêment pauvre; elle a pour opérandes encore une fois des chaînes ou des fragments de chaînes. Les entrées-sorties sont à peu près nulles dans l'état actuel. On n'a aucun accès à la représentation effective du programme ou des données dans la machine.

D E U X I E M E P A R T I E

ETUDE COMPARATIVE DES LANGAGES

ETUDE COMPARATIVE DES LANGAGES

CHAPITRE A - SYNTAXE (1).

1- Structure.

Sous ce terme un peu vague, j'entends parler des caractéristiques syntaxiques dominantes des langages. Je diviserai pour cela les langages en deux grandes catégories : les langages à structure linéaire, dont le type est Fortran, et qui comprennent aussi Cobol, Apt, Comit, IPL et Snobol; les langages à structure parenthésée ou récursive, représentés par Algol, et où sont rangés Neliac, Jovial, PL/I et Lisp.

Les caractéristiques de la première catégorie sont les suivantes :

1°) La structure du langage est liée au support, en l'occurence la carte perforée, le changement de carte en particulier servant de séparateur d'instructions. Ce dernier point ne vaut pas pour Cobol.

2°) L'instruction est coulée dans un moule assez rigide, bien que des parties puissent en être absentes; ce moule est très peu apparent en Cobol, un peu plus en Fortran, beaucoup plus en Apt, Snobol et Comit, complètement rigide en IPL.

3°) La structure du programme est complètement linéaire, c'est-à-dire que toutes les instructions sont au même niveau, sans aucune hiérarchie: en fait, le programme est vraiment une suite d'instructions. Seul Cobol ajoute une très légère hiérarchie au moyen des procédures, mais d'une façon timide et peu rigoureuse. La description par carte syntaxique est généralement difficile, et s'étale en largeur plutôt qu'en hauteur.

(1) dans tout ce chapitre, j'ignorerai systèmatiquement Slip, dont la syntaxe est identique à celle du langage qui l'héberge, en l'occurence Fortran.

Les caractéristiques correspondantes dans la seconde catégorie sont les suivantes :

1°) La structure du langage est indépendante du support, il existe un séparateur d'instructions explicite, pour autant que le concept d'instructions soit encore valable (ce qui n'est plus vrai en Lisp).

2°) L'instruction ou ce qui en tient lieu est construite au moyen de véritables règles de syntaxe, par enchaînement d'articles et regroupement pour constituer une unité de niveau supérieur, qui peut très souvent, après mise entre parenthèses (le mot "parenthèse" étant pris au sens le plus large), redevenir un article de niveau inférieur. PL/I est un peu en marge à ce point de vue, et n'effectue pas de travail de façon systématique.

3°) La structure du programme est complètement parenthésée, c'st-à-dire que des "parenthèses" au sens large (par exemple { et } en Neliac, début et fin en Algol, DØ; et END; en PL/I) introduisent une hiérarchie de profondeur théoriquement indéfinie dans les instructions. La description par carte syntaxique est généralement assez facile (au moins si l'on néglige toutes les complications et cas particuliers de Jovial et PL/I) à cause de la récursivité des définitions, des définitions, et elle s'étale en hauteur plutôt qu'en largeur.

Dans ces deux catégories, Cobol et PL/I se placent assez mal, Cobol ayant gagné quelques caractéristiques de la seconde catégorie, et PL/I ayant conservé certaines des limitations de la première.

2 - Rigueur.

Ce mot n'implique aucun jugement de valeur. J'entends parler ici d'une part de la souplesse relative de la syntaxe, d'autre part de la précision de la définition de cette syntaxe.

1°) La souplesse de la syntaxe n'est pas forcément liée à la structure linéaire ou parenthésée, et d'autre part elle ne doit pas être confondue avec un relâchement de cette syntaxe qui n'en est qu'une dégradation. La syntaxe d'un langage n'est pas souple si l'on peut mettre à peu près n'importe quoi n' importe où, pourvu que cela puisse

avoir approximativement un sens, comme risque de l'admettre quelquefois PL/I, elle ne l'est pas non plus si elle permet des constructions théoriquement parfaites, mais tout aussi parfaitement incompréhensibles, comme celles qui ont pu être obtenues en faisant en Algol de l'art pour l'art. Nous pourrons ainsi dire de langages très différents, comme Comit, Neliac ou Cobol, que leur syntaxe est suffisamment souple, et d'autres tels que Lisp, Fortran ou surtout IPL, que leur syntaxe est rigide. Ceci veut dire que j'appelerai souple la structure d'un langage qui permet en fait d'écrire la même chose de plusieurs façons différentes, toutes également légales mais foncièrement différentes du point de vue syntaxique. De ce point de vue, il est facile de classer les langages, de façon approximative, du plus rigide au plus souple. IPL vient en tête évidemment, en tant que langage machine; Fortran, très proche d'un langage machine malgré ses dernières adjonctions, suit d'assez près; Comit et Snobol sont à peu près équivalents de ce point de vue; Apt, dans le domaine très limité qui est le sien, permet pourtant plus de liberté; Lisp vient ensuite, à une place surprenante étant donné sont haut niveau, mais qui provient du nombre extrêment limité de constructions possibles, bien que celles-ci permettent de représenter des faits très divers; pour un langage commercial, Cobol est très souple, et les limitations que l'on rencontre proviennent plus souvent des divers compilateurs, ce qui indique peut-être une déficience fondamentale du langage; Neliac, Jovial et Algol, membres de la même famille, se suivent de près mais dans cet ordre, et vient enfin PL/I, à qui il arrive comme je l'ai déjà dit, de tomber de la souplesse dans le relâchement.

2°) Depuis le rapport Algol, et malgré les malentendus qu'il a occasionnés, on s'accorde à reconnaître l'utilité, sinon la nécessité, d'une définition précise et rigoureuse de la syntaxe des langages de programmation. Tous ne sont pas également favorisés de ce point de vue, surtout à cause du manque d'une notation permettant une définition vraiment complète. La notation de Backus, utilisée dans le

rapport Algol, n'est pas complète, laisse certains points dans l'ombre, et est trop souvent d'une lourdeur énorme. C'est ce qui explique qu'on ne l'ait jamais utilisée telle quelle pour la description d'autres langages, et qu'en fait on ait chaque fois redéfini un méta-langage de définition, même s'il comprend à la base la notation de Backus, pour tenir compte des particularités du langage décrit. Ainsi pour Jovial ou Comit il faut faire intervenir la signification spéciale du blanc comme séparateur dans certains cas; pour Fortran, il faut un moyen de décrire la répétition limitée de certains éléments, et le fait que la syntaxe de la fin d'une "phrase" puisse dépendre du début : Rabinowitz (référence 6.6.) introduit ainsi des fonctions de variables méta-syntaxiques qui sont une extension particlièrement intéressante du langage de Backus; pour Neliac, Apt ou Lisp (où la description est un record de brièveté), la notation utilisée est à peu de chose près celle de Backus. La définition syntaxique rigoureuse et complète manque pour Fortran IV, Cobol ou PL/I, qui sont les langages pour lesquels elle serait le plus difficile; il existe pour Cobol une description sous forme normale de Backus de la division traitement (référence 3.15), qui est déjà assez monstrueuse. Pour IPL, une description rigoureuse ne semble pas très nécessaire.

3 - Restrictions.

Il s'agit de limitations apportées à la syntaxe, en imposant une particularité à la place de la généralité. Elles peuvent provenir de trois raisons différentes: ce sont les restrictions liées à la machine (par exemple les limitations sur la taille des nombres ou des identificateurs), les restrictions liées à l'optimisation, (par exemple l'ensemble des restrictions sur les indices de tableau en Fortran), et les restrictions arbitraires (comme semble en être une le fait qu'en Algol, la partie valeur doive précéder la partie spécification dans une en-tête de procédure). Le premier type de restrictions ne devraient pas normalement figurer dans la définition d'un langage se

voulant général. En fait, Fortran, Lisp, Comit, PL/I ont été crées d'abord pour une machine particulière (respectivement les IBM 650, 704, 709 et 360), la généralité n'apparaissant que comme sous-produit. Algol au contraire, se voulant complètement indépendant de la machine, l'a été beaucoup trop, jusqu'à oublier l'existence des moyens de communication de la machine avec l'extérieur. En général, les restrictions de ce type se rapportent à la taille maximum des nombres ou des identificateurs, au nombre d'indexages possibles simultanément dans l'utilisation des tableaux ou des boucles "pour", au type et à la représentation des variables, aux caractères mêmes utilisés dans le langage. Les restrictions du second type prédominent en Fortran, manquent nettement au contraire en Algol, sont absentes dans des langages comme Lisp, Comit ou IPL qui se préoccupent peu du temps d'éxécution effectif. On en trouve quelques-unes en Jovial ou en PL/I, dans ce dernier ces restrictions peuvent être indiquées dans le programme lui-même. Les restrictions du dernier type ne devraient figurer dans aucun langage sérieux, mais en fait on en trouve dans tous les langages, peut être pas toujours complètement arbitraires, mais liées à des considérations si bizarres et si compliquées que cela revient au même. Ce sont par exemple, en Fortran l'obligation pour les indices de tableau d'avoir une borne inférieure égale à 1, en PL/I l'obligation du point virgule après DØ, BEGIN ou END, en Algol l'interdiction de l'écriture "2". pour un nombre réel ou l'interdiction des chiffres dans les commentaires d'en-tête de procédure, en Cobol la pléiade des restrictions imposées pour des raisons obscures par chaque compilateur et différant pour chacun, en Lisp, les bizarreries sur la nature et la portée des déclarations, etc... La liste est malheureusement bien longue, et pour l'ensemble de toutes ces restrictions les champions sont sans conteste Fortran et Cobol.

4 - Ambiguités.

Outre les ambiguités réelles de définition du langage, qui ne sont résolues qu'arbitrairement et par chaque compilateur, il ne

faut pas oublier les ambiguités apparentes, c'est-à-dire les constructions très difficiles à interpréter par l'utilisateur, bien que la construction ne soit pas réellement ambiguë ; les ambiguités du premier type sont extrêment rares, heureusement, car leur recherche systématique est très difficile, mais il en existe pourtant, quoiqu'on ne puissent dire effectivement qu'elles existent que dans le cas d'un langage défini de façon explicite et rigoureuse. Les premières définitions d'Algol comprenaient des ambiguités qui ont été levées lors de la révision de 1962. Il est fort probable que l'on en trouvera en PL/I, puisque toute construction approximativement sensée doit être permise. Le fait qu'un "END" suivi d'un nom de bloc puisse fermer plusieurs blocs à la fois, et que des blocs différents imbriqués puissent porter le même nom, semble en particulier pouvoir conduire à des ambiguités caractérisées. En Comit, en supprimant progressivement mais légalement presque tous les éléments d'une règle de grammaire, on arrive à la construction A B C , om il est impossible de déterminer si B est une structure à recherche ou un nom de sous-règle.

5 - Souplesses.

1°) Programmes et instructions.

Je vais retrouver la distinction entre structures linéaires et structures parenthésées au moment de l'inventaire des possiblités des langages dans ce domaine. L'absence ou la présence de la structure de bloc est des l'abord le fait le plus marquant. On la trouve en Algol, en Neliac, en Jovial, en PL/I et en Lisp, avec des représentations et des significations diverses. Dans les quatre premiers, elle est indiquée par un encadrement entre les symboles début et fin (ou leur équivalent), pour Lisp il ne s'agit que d'une paire de parenthèses parmi les autres. En PL/I se produit une particularité bizarre : BEGIN et END ne sont pas des délimiteurs mais des instructions destinées au compilateur, et c'est pourquoi ils doivent obligatoirement être compris tous les deux entre deux points-virgules. De plus l'utilité principale de la structure de blocs, qui est de rendre dynamique l'allocation de mémoires et la désignation des variables, est fortement

bouleversée et compliquée par le fait que "tout marche", et que le compilateur ne doit jamais considérer un identificateur comme non déclaré. D'où les nombreux moyens qui servent à spécifier la portée des identificateurs, leur rémanence ou leur évolution dynamique. Dans les autres langages cependant, on a souvent cherché à supprimer de façon facultative l'automatisme de la structure de bloc : c'est le rôle, d'ailleurs très mal défini, du rémanent d'Algol, c'est celui des spécifications SPECIAL et COMMON de Lisp. Neliac et Jovial, quant à eux, ignorent le problème dans leurs versions officielles. Assez bizarrement, un langage à aussi bas niveau qu'IPL introduit une sorte de structure de blocs au moyen de ses symboles locaux; ceci ne remplit pas cependant le rôle de l'allocation dynamique des mémoires, mais sert au contraire dans le domaine très particulier de l'effacement des structures de liste. La possibilité de l'instruction composée est plus limitée que celle de la structure de bloc, mais elle y est fortement apparentée, puisqu'en fait un bloc est toujours une instruction composée avec quelque chose de plus. La possibilité de "mise entre parenthèses" des instructions au même titre que les expressions, est un outil extrêmement puissant des langages de programmation, mais que l'on ne trouve évidemment que dans les langages à structure parenthésée. Elle manque donc cruellement à Fortran, Apt, Comit ou Snobol. Cobol l'introduit d'une façon très souple mais un peu dangereuse, par l'absence de délimiteur entre les instructions : l'unité normale est l'instruction composée, terminée par un point. En IPL, l'instruction composée n'existe pas vraiment, il en serait difficilement autrement, mais la facilité d'appel des sous-programmes ou procédures est telle qu'elle remplace complètement cette possibilité.

Le nombre et la variété des délimiteurs joue un grand rôle, d'une part dans la rigueur des écritures (donc la facilité de compilation), d'autre part dans la facilité de lecture. La gamme des possibilités s'étend depuis IPL, où les délimiteurs sont complètement absents, puisque les différentes parties d'une instruction doivent figurer à un

endroit précis sur une carte, jusqu'à Algol, où sauf quelques exeptions (le caractère "deux-points" en particulier) on trouve un délimiteur différent pour chaque cas. C'est un très gros ennui en Fortran ou en PL/I que de ne disposer que de fort peu de délimiteurs, car la virgule et les parenthèses sont utilisés sans arrêt et de façon totalement arbitraire; c'est ainsi que l'on doit écrire GØ TØ (4, 5),I mais READ (4,5) I, ou bien DØ 2 I = 1,10,1 mais READ (4,5) (A (I), I = 1,10,1). En Cobol le gros ennui est le manque total de rigueur (et de logique) dans l'utilisation des délimiteurs, puisqu'ils sont presque toujours facultatifs.

D'un autre point de vue, en excluant IPL, on peut diviser les langages en deux catégories, ceux pour lesquels l'espace est sans signification, et ceux pour lesquels il joue le rôle de délimiteur. La première catégorie comprend Fortran, Algol, Neliac et Apt. Lisp est un peu à cheval sur les deux catégories puisque l'on peut, en utilisant systématiquement la virgule comme séparateur, ignorer complètement les espaces à condition de ne jamais couper un symbole. Jovial constitue également un cas un peu hybride: en tant que dialecte d'Algol, il utilise un grand nombre de délimiteurs bien différenciés, mais les adjonctions qu'il fait sur le type et l'utilisation des variables, en faisant placer après chaque déclaration un certain nombre d'attibuts, introduisent à des fins de simplification l'espace comme délimiteur, d'une façon d'ailleurs très souvent arbitraire. C'est ainsi que Cobol réclame, on ne sait trop pourquoi, un espace avant la parenthèse droite d'une variable indicée, ou qu'en PL/I les deux écritures 45.E+02 et 45.E + 02 ont deux significations différentes. En Comit et Snobol les problèmes sont assez similaires, mais cela semble plus normal étant donné leur bas niveau et leur structure linéaire.

Pour Fortran, Apt, Comit et Snobol, le passage à la ligne a de plus la signification de fin d'instruction, assez logiquement si [illegible] la carte perforée, mais avec cependant des inconvé-

nients dans le cas où une instruction est plus longue qu'une ligne. De plus en Fortran et en Cobol la place dans la carte joue un certain rôle, ce qui lie encore plus le langage au support sur lequel il est écrit.

2°) Identificateurs.

Ils peuvent être limités de plusieurs façons différentes, et tout d'abord dans leur longueur. IPL est le plus limité, puisqu'il s'arrête à cinq caractères . Fortran, Apt et Snobol se limitent en général à six caractères (8 pour Fortran sur certaines machines), Comit s'arrête à 12, Cobol à 25, Lisp à 30. Algol, Neliac, Jovial et PL/I n'imposent aucune limitation, étant entendu en général que seule une certaine partie au début de l'identificateur est significative, et dépend au point de vue longueur du compilateur particulier utilisé. Quoique non rigoureuse, cette méthode est sans doute la meilleure.Une autre variante consiste à rendre significatifs par exemple les six premiers et les six derniers caractères d'un identificateur, pour distinguer ainsi RACINE IMAGINAIRE 1 et RACINE IMAGINAIRE 2.

Une autre limitation concerne les caractères autorisés dans la construction d'un identificateur. IPL est là encore le plus limité puisqu'un symbole doit avoir une structure fixe qui indique quelle sera son utilisation, mais il est le seul également à admettre tous les caractères : 15 ou + 17 sont des symboles IPL légaux. Dans tous les autres langages un symbole doit commencer par une lettre et peut contenir des lettres ou des chiffres. Il peut souvent de plus contenir un caractère spécial : l'espace en Algol ou en Neliac, le tiret en Cobol, l'apostrophe en Jovial, le caractère "__" en PL/I, l'astérisque en Comit, avec dans ce cas une signification particulière.

Une notion importante ayant trait aux identificateurs est celle des mots-clés. Dans un bon nombre de langages, les mots tels que DEBUT, FAIRE, SI, ne sont pas considérés comme des symboles de base mais comme des identificateurs placés au même niveau que ceux qui

sont définis par le programme. Plusieurs conséquences sont possibles, en particulier l'interdiction d'utiliser ces symboles à d'autres fins. Le tableau ci-dessous indique les conventions spéciales à chaque langage, y compris le traitement des identificateurs de fonctions standards, qui ressort au même problème.

Algol : pas de mots-clés; les fonctions standards peuvent être redéfinies, car elles sont considérées comme des procédures déclarées dans un bloc encadrant tout le programme; les symboles de base se distinguent par leur écriture.

Cobol : très nombreux mots-clés qu'il est interdit de redéfinir, pas de fonctions standards à proprement parler mais certains des mots-clés sont à ranger dans cette catégorie (AT END, ØVERFLØW, DIVIDE-CHECK, etc....).

Fortran : certains seulement des mots-clés sont interdits dans seulement certains cas mal définis; les fonctions standards peuvent normalement être redéfinies si elles sont fermées, mais ceci n'est pas vrai sur tous les compilateurs.

Apt : très nombreux mots-clés jouant souvent le rôle de fonctions standards, et qu'il est interdit de redéfinir.

Neliac : comme Algol en général, avec la différence qu'en Neliac pur il n'y a pas de symboles de base à cause de la surabondance de caractères spéciaux.

PL/I : il n'y a pas de mots-clés à proprement parler; les mots reconnus par le compilateur sont très nombreux, et comprennent aussi bien les identificateurs de fonctions standards que des pseudos-délimiteurs (c'est-à-dire que BEGIN ou SIN sont traités de la même façon), mais leur utilisation avec une autre signification est toujours permise, la valeur d'un mot dépendant du contexte: ;CALL BEGIN; sera l'appel à une fonction , ;BEGIN; indique le début d'un bloc.

Lisp : pas de mots-clés réels (QUØTE ou NIL sont en fait des pseudos-fonctions), toutes les fonctions standards peuvent être redéfinies, aux risques et périls de l'utilisateur.

IPL : les régions H, J et W, déjà définies dans le système, peuvent être considérées comme des hybrides de mots-clés et de fonctions standards, qu'il est toujours possible de redéfinir.

Comit : pas de mots-clés, pas de fonctions.

Snobol : pas de mots-clés, les deux seules fonctions .READ et .PRINT se distinguent par leur écriture même.

Les étiquettes constituent des identificateurs d'un type particulier, pour lesquelles les conventions sont généralement différentes. Il faut d'abord un moyen de les distinguer des autres identificateurs, et d'autre part leur écriture peut présenter certaines particularités. En Algol, les étiquettes sont soit des identifictateurs, soit des nombres entiers (ce dernier point en théorie seulement à cause des ambiguités possibles au niveau des parmètres). En Neliac, Jovial et PL/I ce sont des identificateurs normaux. Dans ces quatre langages, l'étiquette se reconnaît au signe ":" qui la suit, et plusieurs étiquettes peuvent nommer la même instruction; en PL/I cependant cette notation est ambiguë puisqu'elle est aussi utilisée pour indiquer le nom d'une procédure lors de sa déclaration à moins que l'on ne considère une procédure comme une instruction avec un nom. En Cobol, Apt, Comit et Snobol, l'étiquette se reconnaît à sa place dans la ligne de texte, tout au début de la ligne et souvent dans une zone spéciale (pour Cobol, Fortran et Apt), toujours séparée du reste de la ligne ou de l'instruction par au moins un blanc (et un point en Cobol). En Fortran et Apt, les étiquettes sont uniquement numériques; en Cobol ou Snobol ce sont des nombres ou des identificateurs; en Comit l'étiquette est toujours nécessaire, mais peut-être remplacée par le symbole passe-partout "*". En Lisp, l'étiquette n'est utilisée qu'exeptionnellement, comme paramètre des fonctions IAP et PRØG, et se reconnaît à ce qu'il s'agit d'un symbole atomique placé au niveau supérieur de la liste de paramètres. En IPL, le nom d'une instruction joue en particulier le rôle d'étiquette, c'est un symbole IPL normal.

CHAPITRE B. STRUCTURE.

1 - Déclarations.

1°) Groupage ou mise en facteur des attributs.

Le rôle des déclarations est de préciser la nature et les caractéristiques des variables utilisées, et dans ce domaine règne le plus grand désordre. Du fait de la grande variété possible des types de variables et du mode de déclaration, les solutions adoptées par chaque langage sont très diverses et inégales et il est impossible de dire que l'une est meilleure que l'autre à tous points de vue. Je n'envisagerai pour l'instant que les déclarations elles-mêmes, le paragraphe suivant étudiant les divers types de variables qui peuvent être déclarés. En règle générale, une déclaration associe au nom d'une variable une ou plusieurs caratéristiques. Etant donné qu'une même variable peut avoir plusieurs caractéristiques, ceci peut se faire de plusieurs façons. Certains langages donnent la liste des variables satisfaisant à une propriété : ce sera le cas d'Algol, Fortran ou Neliac; d'autres au contraire donnent pour chaque variable la liste de ses propriétés, et c'est le cas de Cobol, Jovial ou Apt. Etant donnée la multiplicité des caractéristiques qui peuvent être associées à chaque variable, PL/I utilise un moyen mixte, en donnant la liste des variables rangées dans une même grande catégorie, et en associant dans cette liste aux variables qui le nécessitent, avec possibilité de "mise en facteur" sur plusieurs variables, des caractéristiques plus particulières. C'est, sous une forme beaucoup plus étendue, la solution qu'utilisent Algol, Fortran ou Neliac pour les déclarations de dimensions de tableaux, la mise en facteur n'existant d'ailleurs pas en Fortran.

2°) Déclarations implicites.

Mais les déclarations constituent très rapidement, on s'en aperçoit en Cobol, quelque chose de lourd et de verbeux si l'on utilise beaucoup de variables et que peuvent leur être associées de

nombreuses caractéristiques. C'est ainsi que les déclarations implicites jouent un rôle plus ou moins important dans tous les langages. Deux possiblités existent d'ailleurs dans ce domaine.

D'une part, la ou les caratéristiques d'un certain symbole peuvent être déterminées par son écriture ou par sa place. Le premier cas existe systématiquement en Fortran, assez largement en PL/I, de façon beaucoup plus timide en Jovial et en Neliac, pas du tout en Algol, Cobol ou Apt. Il n'est utilisé d'ailleurs que pour déterminer le type d'une variable simple (ou d'un identificateur de fonction) entière ou réelle, d'après la première lettre de l'identificateur. En Jovial et Neliac, les lettres I à N représentent des indices, qui servent comme indices de tableaux et comme variables contrôlées de boucles de pour, et peuvent être utilisés sans avoir été déclarés. Pour PL/I, Jovial ou Neliac, langages qui utilisent la structure de blocs, ceci présente cependant le très gros inconvénient d'interdire ou de supprimer l'effet de cette structure sur la portée des déclarations, ce qui semble être payer bien cher une facilité relativement restreinte. En PL/I, il est possible de définir ou de modifier les corventions de détermination implicite des caractéristiques d'un symbole d'après sont écriture.

La détermination des caractéristiques d'un symbole d'après sa place dans un programme ou une instruction se rencontre sous des formes très diverses. Le cas des étiquettes a déjà été étudié : dans tous les langages, les étiquettes se déclarent elles-mêmes par leur apparition sous la forme normale. En Fortran, les fonctions arithmétiquement définies représentent un cas particulier de déclaration implicite liée à la place dans le programme. On retrouve quelque chose d'analogue en Apt, mais de façon beaucoup plus générale puisque l'on "déclare" ainsi des courbes, des surfaces, des volumes, au moyen d'équations explicites ou paramétriques.

La seconde possiblité de déclaration implicite affecte des caratéristiques "par défaut" dans le cas où on n'en a pas indiqué

explicitement. En d'autres termes, il est possible de spécifier toutes les propriétés d'une variable, mais si l'on ne mentionne pas certaines de ces propriétés on considère qu'il s'agit de celle qui est la plus courante. Ce procédé à l'avantage de la facilité (au moins apparente), aux dépens de la rigueur et de la clarté : pour tout dire, c'est une solution de paresse pour le programmeur. Algol ne l'utilise pratiquement pas, à part un tout petit cas particulier ("tableau" signifie "réel tableau"). Neliac et Fortran l'utilisent plus largement, mais le nombre de caractéristiques qui peuvent être associées à une même variable est très réduit. Il est plus important en Jovial, Apt ou Cobol, mais ici seules certaines des caractéristiques peuvent être considérées comme indiquées implicitement au moyen d'attributs "par défaut", les principales devant toujours être explicites. PL/I enfin, où tout doit marcher, pousse le procédé à son paroxysme, certainement aux dépens de la clarté immédiate des programmes: en fait, chaque fois que l'on omet ou que l'on oublie quelque chose, le langage à toute prête une solution de remplacement, qu'il s'agisse d'une option standard ou d'une option par défaut, ce qui ne va pas sans danger.

3°) Langages sans déclarations véritables.

J'ai omis jusqu'à présent de parler de nos quatre langages non numériques. Dans le domaine des déclarations, ils constituent en effet un cas très particulier puisqu'ils n'en possèdent en général pas quoique l'on trouve parfois des choses qui, considérées sous un certain angle, peuvent paraître en tenir lieu. Plus précisément, IPL contient certaines particularités qui rappellent les déclarations : les cartes de type 2 spécifiant le nom et les dimensions des régions utilisées, le premier caractère d'un symbole indiquant son mode de traitement (régional, c'est-à-dire protégé, local ou interne), le code P des termes donnés indiquant le type de la donnée, etc..... En Lisp, les propriétés d'un symbole sont rangées dans une liste qui lui est attachée et qui peut être modifiée à tout moment, ce qui constitue un cas particulièrement intéressant de déclaration complètement dynamique.

En Comit, les variables utilisées sont déclarées complètement par leur apparition et leur utilisation (si tant est que l'on puisse parler de déclaration), ce qui constitue un nouveau cas de déclaration implicite liée à la forme. Quant à Snobol, toutes les variables traitées y sont des chaînes, et aucune déclaration n'a besoin de leur être associée pour indiquer des propriétés qui n'existent pas.

2 - Types des données.

1°) Variables simples arithmétiques.

Dans la diversité des types de variables simples intervient beaucoup la machine ou au moins le type de machine pour laquelle un certain langage est prévu. Il est indiscutable que les types définis en Fortran sont adaptés à une machine à mots, que ceux de Cobol sont destinés à une machine à caractères, et ceux de PL/I à une machine du type IBM 360.

En Algol et Neliac, les seuls types reconnus sont le type entier et le type réel. Etant donné qu'on ne veut pas dépendre d'une machine, il n'existe aucun moyen de spécifier la précision ou la longueur des variables, ce qui est très pauvre. En Cobol, il est supposé implictement que les nombres sont utilisés suivant les conventions des machines à caractères, et ceci explique pourquoi on peut spécifier la longueur d'un nombre, la place du signe ou du point décimal, la justification des chiffres significatifs de la zone réservée, etc..... Au contraire, la distinction entre représentation fixe et représentation flottante est très mal établie, et l'on n'est jamais très sûr de la façon dont sera traitée une variable. Apt ne connaît que les nombres réels, supposés traités dans une machine à mots, donc sans question de longueur et de précision, et n'utilise les entiers que pour le cas très particulier des indices, mais ceci semble normal étant donnée l'utilisation très spéciale à laquelle le langage est adopté. En Fortran, quatre types sont reconnus, les types entier, réel, double précision et complexe; ceci semble suffisant dans la plupart des cas, quoique la représentation exacte, longueur et précision, dépende de la machine et du compilateur et ne puisse jamais être spéci-

fiée. Jovial reconnaît deux types de plus que Fortran: au type réel flottant, qui correspond au type réel de Fortan, il ajoute le type réel fixe, et surtout le type dual, dont les applications semblent d'ailleurs assez limitées, quoique intéressantes pour le traitement des coordonnées sur une surface. De plus, on peut en Jovial spécifier exactement la longueur d'un nombre, la place du point décimal, le traitement de l'arrondi, etc.... La description d'une variable peut donc être très complète, ou être sous-entendue grâce à l'utilisation de l'attribut "par défaut", et elle ne paraît pas être liée à un type particulier de machine, ce qui peut-être risque de rendre l'application difficile aussi bien sur une machine à mots que sur une machine à caractères. PL/I étend les possibilités de Jovial pour les adapter aux caractéristiques particulières du système 360 : machines où les nombres peuvent être représentés et traités soit par caractères décimaux soit en binaire. Ceci a le gros inconvénient d'être très clairement et très étroitement lié à la machine. Par rapport à Jovial, PI/I supprime le type dual et la spécification du traitement de l'arrondi. IPL ne connaît que les types entier et réel, sans contrôle de la longueur. Il permet cependant la transcription des nombres entiers en écriture octale, ce qui suppose d'ailleurs implicitement l'utilisation d'une machine à mots. Lisp a les mêmes possibilités qu'IPL, sauf que chez lui le type octal est considéré comme logique plutôt qu'arithmétique. Comit est extrêmement pauvre puisqu'il n'est capable de traiter que le type entier, d'une façon très spéciale et limitée, et sans aucun contrôle possible sur la longueur ou la représentation. Snobol n'est pour le moment guère capable de faire mieux, les nombres qu'il traite sont des entiers ou des nombres en virgule fixe, traités de la même façon que des chaînes, c'est-à-dire de longueur indéfinie.

2°) Variables simples non arithmétiques.

Dans ce domaine les possibilités des langages sont très variables puisqu'elles vont de l'ignorance complète du problème jusqu'au traitement le plus général et exhaustif. Quatre types importants seront

à considérer dans ce domaine: les types booléen, logique (différent du booléen: je veux parler de variables qui sont formées d'un ensemble, fini ou non, de variables booléennes), chaîne et symbolique (c'est-à-dire le type d'une variable qui sera traitée pour son nom même et non pas pour le contenu du mot désigné par ce nom).

Algol ne connaît que les booléens et les chaînes. Les premiers sont utilisables d'une façon très générale, mais pour les chaînes aucune opération n'est prévue, et leur utilisation est limitée à l'apparition en paramètres effectifs de procédures, ce qui est une carence aussi absurde que grave. Cobol est un peu à part puisqu'il ne possède pas de véritables déclarations de type. En fait les variables qu'il traite peuvent le plus souvent être considérées comme des chaînes et traitées comme telles, avec une gamme étendue de manipulations possibles, mais ceci dépend très étroitement du compilateur utilisé. Fortran reconnaît le type booléen, qu'il traite de façon simple et commode. Il est possible par des moyens détournés de traiter des chaînes, mais le type n'est pas prévu, et la seule façon possible est d'utiliser des variables indicées de type quelconque, avec le risque de conversions intempestives. Apt n'a guère besoin des variables booléennes et logiques, les chaînes lui manquent peu, mais il utilise très largement les variables symboliques quoique d'une façon assez particulière, puisqu'une variable peut représenter un point, une courbe, une surface, une équation, une fonction, etc..... Il s'agit là d'un point de vue fort spécial mais fort intéressant, et qui mériterait sans doute d'être développé, même de façon limitée, dans d'autres langages. Neliac est le plus limité de tous puisqu'il ne connaît que le type booléen, et ignore totalement tous les autres : c'est assez maigre. Jovial au contraire est assez complet puisqu'il est capable de traiter des variables booléennes, des variables logiques au moyen du "modificateur"BIT, des chaînes au moyen du modificateur BYTE affectant des variables alphanumériques. L'inconvénient est que de véritables chaînes ne peuvent être traitées qu'au moyen de véritables indicées, comme en Fortran, car une variable aphanumérique est forcément

un mot de longueur fixe, limitée et constante. PL/I étend les possibilités de Jovial au traitement de véritables chaînes, de longueur quelconque mais spécifiée, il peut ainsi traiter des variables logiques ou des chaînes en complète banalisation.... en particulier parce qu'il les confond souvent. Lisp a des possibilités très spéciales. Toutes les variables sont en fait symboliques, et peuvent représenter absolument n'importe quoi, qu'il s'agisse d'un caractère, d'une adresse machine, d'une fonction ou du compilateur lui-même; en particulier les valeurs booléennes ne sont que deux valeurs possibles (*T* et NIL) parmi l'ensemble des valeurs symboliques. Le traitement des variables logiques se fait de façon très générale, de même que celui des chaînes, ces dernières étant de longueur indéfinie. En bref, Lisp est le langage le plus à l'aise avec les variables non arithmétiques, et le plus souple puisque le type d'une variable n'est en fait que la façon dont on l'utilise. IPL ignore totalement les types booléen ou logique, les décisions étant prises d'une autre façon. Il peut traiter des variables alphabétiques de longueur fixe et limitée, et des variables symboliques à condition qu'elles soient d'une forme IPL. Au total, cette simplicité spartiate ne s'explique que par sa nature de langage machine. Comit ne connaît pas à proprement parler le type booléen, mais plutôt le type logique, au moyen des diverses valeurs qui peuvent s'attacher à un indice et constituent un ensemble de variables booléennes de longueur limitée. Le traitement des chaînes est le travail principal de Comit, mais il n'y existe pas à proprement parler de variables qui soient des chaînes, puisqu'on n'a à sa disposition qu'un nombre limité de ces chaînes et que l'on ne travaille jamais que sur une seule à la fois. Les variables symboliques se retrouvent un peu dans les valeurs d'indices, mais avec une utilisation restreinte et bizarre, car Comit ne fait jamais rien comme les autres. Snobol ignore tout autant que Comit le type booléen, encore plus le type logique, il est finalement à ce point de vue aussi dépourvu de tout qu'IPL, ce qui est au moins anormal. Il traite les chaînes et les symboles d'une façon bien

à lui, puisqu'il les confond: les chaînes que traite le langage ont des noms symboliques, mais ceux-ci sont des chaînes comme les autres, qui peuvent à leur tour avoir des noms, et ainsi de suite. Ce point de vue original n'est pas sans intérêt, car il introduit enfin dans le traitement des chaînes une véritable généralité.

3°) Variables conditionnelles.

Quoique ce type de variable eût pu figurer dans le paragraphe précédent, je pense préférable de le considérer complètement à part. Je veux parler ici de ce que certains langages appellent "noms de conditions", d'autres "variables d'état", et qui sont à mi-chemin entre les variables logiques et les variables symboliques, avec des propriétés des deux. J'appelerai donc variable conditionnelle une variable qui peut prendre des valeurs symboliques représentant l'existence de certaines conditions. Ceci peut se faire de plusieurs façons, et tout d'abord être complètement ignoré: c'est la solution choisie (si l'on peut dire) par Algol, Fortran,Neliac et IPL.

Si l'on utilise les variables conditionnelles, on peut alors les considérer comme des variables normales, que le programmeur définit et utilise comme d'autres, cu comme des "pseudos-fonctions" spéciales définies dans le langage et sur lesquelles on n'a que peu de contrôle. La première solution se rencontre en Cobol, Jovial et Comit. En Cobol, ceci se fait cependant de façon très limitée, puisque l'utilisateur peut seulement attribuer un nom à certaines conditions puis regarder ensuite l'état de cette condition, mais sans jamais la modifier lui-même; au total, c'est une possibilité qui ne présente guère d'intérêt. Jovial au contraire reconnaît le type spécial "d'état", qui se manipule exactement comme un autre : il est le seul à avoir osé pousser cette idée jusqu'au bout, il offre ainsi une solution complètement générale mais presque trop luxeuse. Pour Comit, les indices logiques avec leurs valeurs, que j'ai déjà fait figurer parmi les variables logiques, sont aussi des variables conditionnelles puisque l'on donne ainsi un nom à la présence d'un certain chiffre binaire dans une variable logique. L'utilisation en est très différente de celle de Jovial, mais peut-être encore plus générale et pratique.

La seconde possibilité d'utilisation des variables conditionnelles est beaucoup moins ambitieuse, c'est pourquoi on la trouve dans plus de langages. Cobol, Jovial, Apt et PL/I le font de façons à peu près semblables, et dans le même ordre d'idées: fin de fichier, dépassement de capacité, division impossible, etc.....; PL/I ajoute les conditions spéciales liées à la multi-programmation, Apt ajoute celles qui sont liées à ses problèmes spéciaux, conditions sur la place de l'outil par exemple. De façon assez étonnante, Fortran IV représente à ce sujet un recul par rapport à Fortran II, qui reconnaissait certaines conditions; il est vrai que ce recul se fait au profit de la généralité puisque l'on remplace ceci par des appels à des fonctions. En Comit et en Snobol, on peut rattacher aux mêmes problèmes celui de la reconnaissance d'une certaine structure plus ou moins bien définie dans une certaine chaîne, mais ceci sera étudié plus en détail par la suite. Dans tout ce domaine, Lisp constitue un cas à part : il est assez bien servi d'ailleurs avec son utilisation complètement générale des fonctions-prédicats, qui grâce à la nature particulière des valeurs booléennes s'utilisent comme d'autres fonctions; certaines sont définies dans le langage, ce qui rappelle la deuxième des deux solutions précédentes, mais d'autres peuvent toujours être définies par l'utilisateur, ce qui ramène à la première solution.

4°) Types divers.

Sous ce titre je rangerai trois types inclassifiables: la variable étiquette (avec le cas particulier de l'aiguillage), les procédures et les fichiers. Ces trois types seront d'ailleurs étudiés plus en détail au moment de leur utilisation. Peu de langages considèrent les étiquettes comme des variables, mais presque tous ont ressenti la nécessité d'une solution de remplacement, qui est généralement l'aiguillage. La solution adoptée par Algol, Neliac et Jovial est celle d'un aiguillage considéré en quelque sorte comme un tableau d'étiquettes, mais rempli au moment de sa déclaration et par consé-

quent impossible à modifier.

Cette solution est boîteuse et incomplète, PL/I la généralise en permettant le remplissage d'un tel tableau (ou d'ailleurs d'une variable simple) au moyen d'une véritable instruction d'affectation d'étiquette. C'est un peu la solution qu'utilise Fortran avec son allerà imposé, mais l'utilisation en est lourde et délicate, de même que celle du allerà calculé, qui est un aiguillage redéclaré à chaque utilisation mais nor modifiable. Lisp utilise si peu l'étiquette qu'il ignore carrément le problème. IPL, Comit et Snobol remplacent l'aiguillage par l'adressage indirect dans l'instruction allerà, les symboles qui constituent des étiquettes étant traités exactement comme les autres. Cobol utilise une solution très lourde qui est en fait le allerà calculé de Fortran, cependant qu'Apt ignore le problème.

Les procédures constituent un cas très particulier de v ariables, en particulier parce qu'elles sont presque toujours constantes, sauf en PL/I et surtout en Lisp. Je n'examinerai pas ici leur utilisation mais ce qui se rattache à une déclaration dans leur définition. Tout d'abord, IPL, Snobol, Comit et Cobol ignorent totalement les prccédures, au sens de sous-programmes qui ont pour principal sinon seul effet d'affecter une valeur à un identificateur particulier, et pour principale facilité d'être appelés par la seule apparition de cet identificateur dans une expression ou une instruction. Algol, Fortran, Neliac et Jovial définissent les procédures de manières très semblables, le type de l'identificateur de procédure étant indiqué exactement comme pour un autre identificateur, explicitement ou non. Algol est le plus général par le fait que les déclarations de procédure ont une portée liée à la structure de blocs, exactement comme d'autres déclarations. Fortran et surtout Apt utilisent un procédé plus original, qui équivaut presque à une instruction d'affectation symbolique, pour la définition de fonctions au moyen d'une seule expression arithmétique. PL/I introduit de nombreuses nouveautés dans la

forme même de la déclaration : l'identificateur de procédure n'apparaît que comme une étiquette qualifiant un bloc; l'affectation de valeur à cet identificateur se fait par une instruction spéciale; enfin l'introduction des fonctions génératrices permet en fait de définir plusieurs procédures en une seule, le sous-programme qui sera effectivement appelé dépendant du type des paramètres effectifs; au total donc, PL/I n'introduit une plus grande généralité que par le fait qu'une procédure peut posséder plusieurs points d'entrée, il introduit surtout de nombreuses particularités et facilités supplémentaires. En Lisp, à peu près tous les symboles utilisés représentent des procédures; une procédure est définie quand elle apparaît en paramètre de la procédure DEFINE, elle peut être modifiée peu ou prou par d'autres procédures, un programme peut construire une procédure pour l'utiliser ensuite, ce qui est vraiment un cas extrême de déclaration dynamique.

Les fichiers sont des variables d'un type très spécial, qui représentent en même temps un grand nombre d'informations. Seuls trois langages les reconnaissent et les utilisent; il s'agit de Cobol, Jovial et PL/I, et les modes de déclarations sont très similaires, le premier langage ayant visiblement inspiré les deux autres. Le principal effet d'une déclaration de fichier est de réserver des zones de mémoires et des unités d'entrée-sortie, par un procédé et suivant des conventions très étroitement liés à la nature de la machine et de son système de programmation. Disons simplement que c'est Cobol qui en permet la description la plus complète, alors qu'en Jovial ou en PL/I beaucoup de paramètres sont fixés, rigoureusement ou dans des limites étroites, par le compilateur.

3 - Arrangements des données.

1°) Tableaux

Un tableau est essentiellement un ensemble ordonné d'éléments tous de même type, qui peuvent être à leur tour des tableaux , et auxquels les références se font par le moyen de coordonnées

numériques entières. Ce sont ces deux caractéristiques qui établissent la distinction avec les structures, que j'étudierai dans le paragraphe suivant. IPL, Comit et Snobol n'utilisent pas les tableaux; Apt n'est capable de traiter que des tableaux à une seule dimension, la borne inférieure étant toujours 1, la borne supérieure pouvant être une variable. Les autres langages définissent les tableaux de façons très semblables, quoique d'une généralité inégale. Etant donné qu'un tableau quel qu'il soit aura toujours les deux caractéristiques indiquées au début de ce paragraphe, les facilités de déclaration et d'utilisation peuvent être assez diverses. Pour des questions d'optimisation de la recherche dans un tableau, tous les langages sauf Algol et PL/I limitent à trois le nombre de dimensions. Pour des raisons semblables, tous les langages sauf ces deux-là imposent une borne inférieure d'indice fixe, 1 pour Cobol, Fortran et Lisp, 0 pour Neliac et Jovial. Enfin, seuls encore Algol et PL/I autorisent la déclaration de tableaux à bornes variables, eux seuls peuvent le faire puisque cette possibilité impose l'allocation dynamique de place à des tableaux ainsi déclarés. Pour Cobol, Fortran, Apt et PL/I, le ou les indices sont placés entre parenthèses, ce qui pose de graves problèmes de syntaxe quand il s'agit de faire la distinction avec les appels de fonctions, surtout pour Fortran et PL/I où les parenthèses sont utilisées à tort et à travers, alors qu'en Algol, Neliac et Jovial on utilise deux délimiteurs spéciaux. En PL/I en plus, pour ne pas dérouter les utilisateurs de Fortran, on autorise l'absence de spécification de la borne inférieure d'indice, elle est alors prise égale à 1. Lisp utilise les tableaux d'une manière bizarre: il est déjà assez étonnant que les bornes ne puissent être variables dans un langage où tout est dynamique; de plus la déclaration d'un tableau est en fait la déclaration d'une fonction de recherche d'éléments dans un ensemble, et l'occurence d'un élément sera un appel à cette fonction. En Cobol, la déclaration et l'utilisation d'un tableau ne sont que des cas particuliers dans la déclaration et l'utilisation des structures.

2°) Structures.

Une structure est un ensemble de données hiérarchisées de longueurs inégales: à partir d'éléments insécables du niveau le plus bas, on construit des ensembles de niveau de plus en plus élevé, jusqu'au niveau le plus haut qui est celui de la structure elle-même; un point très important est que l'on peut se référer à un sous-ensemble d'une structure à n'importe quel niveau. Le concept de structure est très adapté à la description d'enregistrements sur une bande magnétique par exemple, il est donc étroitement lié aux problèmes de gestions, et on ne le trouve que dans les langages qui prévoient de s'occuper de ces problèmes, c'est-à-dire Cobol, Jovial et PL/I. Là encore, Cobol a nettement inspiré les deux autres langages, si bien que les déclarations et les utilisations se font de façons très semblables, par numérotation des différents niveaux d'une structure. Cobol permet de plus, quant à un même niveau plusieurs éléments sont de même forme, de considérer leur ensemble comme un tableau; il est d'autre part possible d'utiliser des éléments de longueur variable. En Jovial, c'est la structure elle-même qui peut être de longueur variable, quand il s'agit d'une table qui est quelque chose d'intermédiaire entre un tableau et une structure (à peu de choses près, c'est un tableau de structures). Au total, les structures constituent un concept très intéressant mais mal défini, utilisé de façon anarchique et dans trop peu de langages. Ce pourrait être pourtant une excellente généralisation des tableaux.

3°) Listes.

Deux points capitaux caratérisent une liste: d'une part une liste est un ensemble d'éléments ordonnés (c'est-à-dire reliés par une relation d'ordre totale), d'autre part les éléments d'une liste sont à leur tour soit des listes, soit des éléments "atomiques", sans préjuger de ce que peuvent être ces atomes. La principale utilité d'une liste est que, étant donné un élément, on peut immédiatement trouver celui qui le suit (et en Slip, celui qui le précède). Ceci permet de posséder un ensemble de données qu'il est toujours possible de modifier dans sa longueur et dans sa structure, en particulier

pour y insérer des éléments. Quoique PL/I explicitement et Algol implicitement utilisent les structures de listes pour permettre la récursivité des procédures, que Comit et Snobol les utilisent dans l'organisation interne des données traitées, seuls IPL, Lisp et Slip en permettent l'utilisation effective et contrôlée. Deux points différents sont à étudier : d'une part le mode de représentation extérieure des listes, c'est-à-dire leur écriture symbolique, d'autre part leur représentation intérieure, c'est-à-dire le mode de rangement dans la mémoire d'une machine.

En IPL, une liste est constituée de mots qui contiennent deux zones principales : la zone "symbole" représente l'information contenue par le mot, la zone "lien"permet de trouver le mot suivant. L'écriture symbolique d'une liste se fait donc comme celle d'un programme, en écrivant simplement la suite des mots, chacun de ces mots pouvant avoir un nom c'est-à-dire une étiquette. En Lisp, deux représentations sont possibles; la première rappelle celle d'IPL, en ce sens qu'une liste est un ensemble de paires, chaque paire étant le premier ou le deuxième élément d'une paire qui la contient; la seconde représentation est moins clairement liée au mode de rangement mais beaucoup plus utilisable, elle consiste à sous-entendre dans chaque mot la partie qui pointe sur le suivant, les parenthèses servant alors à indiquer les hiérarchies. Les trois écritures suivantes, la première étant celle d'IPL, représentent la même structure de liste :

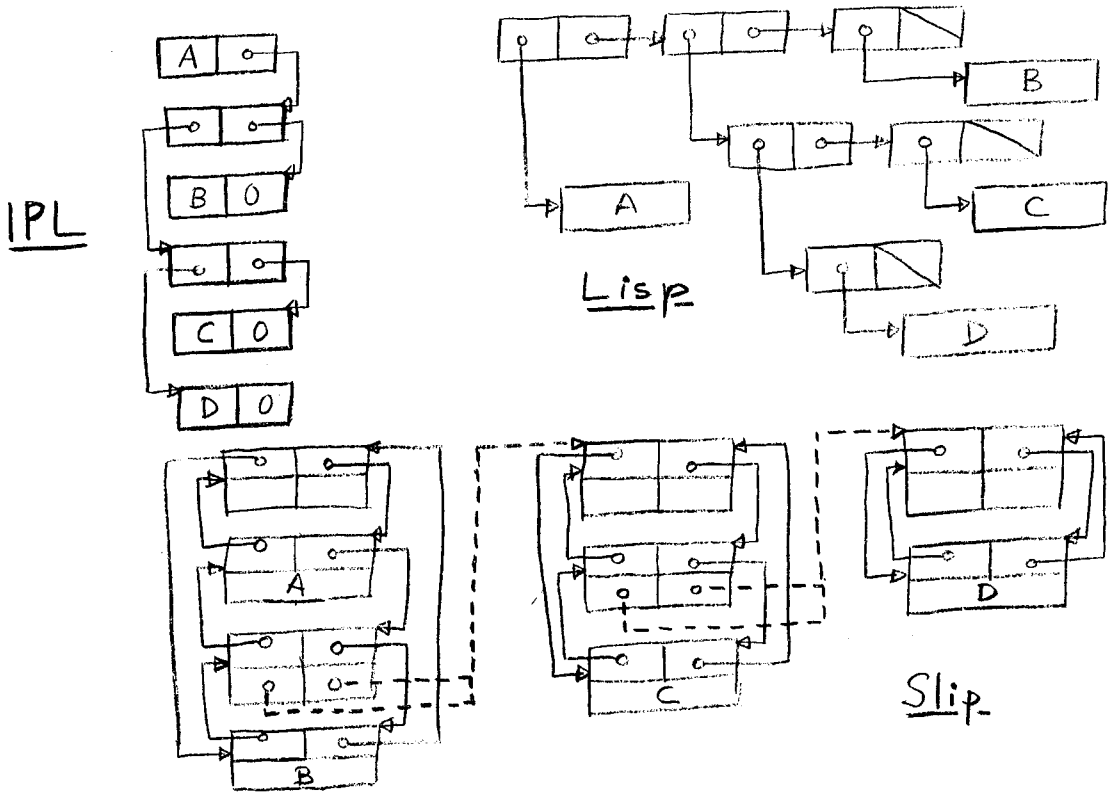
I

L	A	
	9-1	
	B	0
9-1	9-2	
	C	0
9-2	D	0

II (A . (((D.NIL).(C.NIL)).(B.NIL)))

III (A ((D) C) B)

En Slip, il n'existe aucune représentation extérieure définie des structures de liste, ce qui est une carence assez grave. La représentation interne dans chacun des trois langages sera expliquée plus clairement par les schémas suivants :



Ces rangements se font en IPL et Lisp de façons à peu près identique; Slip au contraire est très différent pour plusieurs raisons: les listes sont symétriques, c'est-à-dire qu'étant donné un mot on peut trouver aussi bien celui qui le précède que celui qui le suit; l'élément d'information est rangé directement dans la liste chaque fois que cela est possible, alors qu'en IPL ou en Lisp on ne peut l'obtenir qu'indirectement; enfin, si la liste occupe plus de place qu'en IPL ou en Lisp, elle contient une surabondance d'informations qui en facilite énormément l'utilisation.

4°) Références aux éléments des ensembles.

Une fois défini un ensemble de données, tableau, structure ou liste, il faut pouvoir s'y référer, en gros ou en détail. Le problème est très complexe, rarement résolu de façon satisfaisante, et en tout cas jamais complètement. L'appel d'un élément de tableau se fait toujours en indiquant la valeur des indices qui caractérisent cet élément, mais certains langages imposent des restrictions à cette écriture, pour des raisons de simplification et surtout d'optimisation. Fortran et Neliac n'admettent que des expressions arithmétiques extrêmement simples en indice, ce qui est souvent très gênant. Pour Fortran, Neliac et Jovial, l'expression arithmétique doit être de type entier. Finalement, seuls Algol et PL/I permettent l'utilisation d'une expression arithmétique, conditionnelle ou non, du type le plus général. PL/I autorise de plus la référence à un élément de tableau de niveau plus élevé, ligne ou colonne pour un tableau à deux dimensions, et définit là-dessus une gamme très limitée d'opérations dont l'utilité n'est pas immédiatement évidente, en particulier la multiplication de deux matrices terme à terme.

La référence aux éléments de divers niveaux d'une structure se fait par le moyen de qualifications de symboles, mais aucune règle précise de syntaxe n'a jamais été donné par aucun des trois langages qui les utilisent: si l'on veut appeler un élément du niveau le plus bas, on peut le qualifier par tous les niveaux qui sont au-dessus de lui, mais ce procédé est souvent redondant, et il peut être possible de supprimer certaines de ces qualifications sans créer d'ambiguité; cependant ceci dépend complètement de la structure utilisée et de la façon dont elle a été définie, personne n'a jamais songé à donner de règles générales, et l'anarchie qui règne là est encore plus grande que celle que l'on rencontre au moment de la définition de la structure. Il semble que l'utilisation des numéros de niveau, par exemple, ou la normalisation des quelques règles confuses qui existent actuellement, permettraient d'établir une certaine classification dans ce domaine, où les tableaux éléments de structures utilisés par Cobol et les tables de structures de Jovial encore au désordre.

La référence aux éléments d'une liste est un problème encore plus complexe étant donnée la nature même de ce que l'on traite, et ceci se fait en réalité par l'utilisation de très nombreuses fonctions. En IPL, Lisp et Slip, on peut ainsi faire appel à une liste prise dans son ensemble, à l'en-tête d'une liste, au mot suivant un mot donné, au dernier mot d'une liste. Pour IPL et Lisp il est très difficile de se déplacer dans une liste de la queue vers la tête, alors qu'en Slip il est aussi simple de rechercher le mot précédant un autre que celui qui le suit, et le dernier mot d'une liste que le premier. Il faut de plus pouvoir se référer non pas seulement aux termes d'une liste mais à ce qu'ils représentent. Ceci se fait d'une façon différente dans les trois langages. En IPL, on trouve dans une liste le nom d'un terme ou le nom d'une liste, mais les deux ne se distinguent que par le contexte. En Lisp, on n'a jamais d'accès direct à la donnée elle-même, chaque élément étant en fait le nom d'une liste qui indique sa nature, sa valeur ou ses diverses propriétés. En Slip on retrouve une solution proche de celle d'IPL, avec la différence qu'un élément simple est rangé directement dans la liste, puisque la place est prévue pour le mettre, et que le nom d'une liste se reconnaît à sa structure même. Slip ajoute en plus la possibilité du lecteur, qui sert à balayer les divers éléments d'une liste en tenant compte des diverses structures rencontrées et de la nature des éléments balayés; ceci simplifie énormément toutes les recherches, qui dans les deux autres langages sont lourdes ou coûteuses parce qu'elles ne se souviennent jamais du résultat précédent.

4 - Affectations et conversions.

1°) Instruction d'affectation.

L'instruction d'affectation est l'une des plus importantes de la plupart des langages, mais sa simplicité n'est qu'apparente. Son rôle est d'évaluer une expression quelconque d'une part, une partie gauche d'autre part, et de ranger le résultat de l'une dans l'endroit désigné par l'autre. Dans la plupart des langages d'ailleurs,

ceci se réduit au calcul d'une expression arithmétique et à l'affectation de sa valeur à une variable unique, simple ou indicée. Les quelques perfectionnements sont les suivants : Algol, Neliac, Jovial, Cobol et PL/I admettent l'instruction d'affectation multiple, où une même valeur est affectée à plusieurs variables; Jovial utilise l'instruction d'échange, dont l'utilité est très limitée; PL/I, Jovial et Cobol permettent le rangement d'une structure dans une autre, en bloc ou "par nom", c'est-à-dire en tenant compte des noms d'éléments qui se correspondent dans les deux structures; PL/I utilise d'une part l'instruction d'affectation de tableaux complets ou de sous-ensembles de tableaux, d'autre part l'instruction d'affectation d'étiquettes, qui est une nouveauté très importante. Cependant tout ceci est fait de façon assez anarchique et assez timide, et presque tous les langages semblent avoir peur d'une écriture du type :"expression d'adresse := expression" (par exemple si A< B alors A sinon B :=.....), l'expression à droite étant elle aussi la plus générale possible, et pas seulement une expression arithmétique. Seul en fait, Lisp atteint à cette généralité* : il est possible d'affecter par exemple une définition de fonction à quelque chose qui est le résultat de l'évaluation d'une autre fonction, et toute variable est en fait une fonction. En Snobol, l'instruction d'affectation, qui ne porte évidemment que sur des chaînes, a ceci de particulier qu'elle est souvent destructrice, c'est-à-dire qu'il s'agit en réalité d'un véritable transfert avec effacement. En IPL ou en Comit, il n'y a pas d'instructions d'affectation mais plutôt des instructions de rangement, de transfert ou de remplacement.

* Egalement CPL (référence 11.6), qui permet explicitement l'écriture indiquée plus haut.

2°) Conversion en affectation.

Un des problèmes les plus graves que pose l'instruction d'affectation est celui des conversions; il se pose chaque fois que le type de la valeur à ranger n'est pas celui de l'endroit où l'on range, si ce dernier type est défini. Le problème se pose avec le plus d'acuité dans les langages où les types sont nombreux et différenciés. Algol et Neliac sont très limités puisqu'ils ne connaissent que trois types, le type booléen étant d'ailleurs autonome. Ce dernier point de vue semble assez logique, et c'est celui qu'adoptent également Jovial et Fortran: une valeur booléenne ne peut être affectée qu'à une variable booléenne. Le type dual de Jovial, de par sa nature même, est soumis aux mêmes restrictions. En Fortran il est possible, par le biais d'équivalences entre variables déclarées de types différents, de supprimer la conversion, ce qui est parfois utile. En PL/I tout mélange de types quels qu'ils soient est explicitement permis : il est possible d'ajouter un nombre complexe à une chaîne et de ranger le résultat dans un entier; une expression mal écrite ou erronnée pourra donc provoquer de nombreuses conversions plus ou moins intempestives, et dont le résultat est au moins difficile à prévoir. En Lisp, tout est symbolique donc il n'y a aucune conversion: il est cependant possible de changer le type d'une variable arithmétique au moyen d'un appel à la fonction appropriée.

Les conversions quelles qu'elles soient peuvent en tout cas se produire en plusieurs points: d'une part pendant l'évaluation de l'expression, d'autre part au moment de l'affectation elle-même. La chose n'est pas simple, surtout pour la détermination du type d'une expression, les affectations multiples la compliquent encore, les règles ne sont que peu ou pas définies, et en fait chaque compilateur travaille à peu près comme il l'entend, ce qui est un peu dangereux et très dommage.

3°) Conversions en appels de procédures.

Il est un autre point encore où peuvent être nécessaires

des conversions, c'est au moment des appels de procédures ou de sous-programmes. En Neliac, en Jovial, et sur la plupart des compilateurs Algol, on contrôle la correspondance entre les types des paramètres formels et ceux des paramètres effectifs, le plus souvent pour interdire les discordances. En Fortran, on ne fait aucun contrôle de type et aucune conversion, ce qui est souvent très dangereux. En PL/I, la plupart des fonctions standards sont des fonctions génériques, c'est-à-dire que la fonction qui est appelée en fait dépend du type des paramètres effectifs, et qu'il n'y a pas non plus de conversion; le programmeur peut définir ses fonctions personnelles de la même manière. Dans le cas de procédures plus classiques, il n'y a pas non plus de de conversion; le programmeur peut définir ses fonctions personnelles de la même manière. Dans le cas de procédures plus classiques, il n'y a pas de conversion à cause de la correspondance entre les paramètres formels et effectifs. En Lisp il n'y a évidemment aucun problème, quant à IPL les fonctions arithmétiques peuvent effectivement faire des conversions car une variable porte son type indiqué par un certain index.

4°) Changement de nom et réutilisation.

Il peut être intéressant dans certains cas de donner plusieurs noms différents à la même chose, par exemple pour supprimer les conflits de type. Il peut être très utile aussi, quand un certain nombre de variables cessent d'être utilisées, de récupérer la place qu'elles occupaient. Ces problèmes sont résolus de façon diverse et inégale, et pas par tous les langages. En Fortran, on dispose des "communs" et de l'équivalence. La première possibilité sert habituellement de communication entre divers sous-programmes, mais elle sert aussi à faire se recouvrir des zones de travail de sous-programmes quand elles n'ont pas à être utilisées conjointement. La deuxième possibilité est un artifice qui facilite un peu la réutilisation des zones devenues inutiles, mais qui réclame beaucoup de soin dans l'écriture des programmes, car il n'y a évidemment aucun contrôle. En Algol et PL/I, la structure de bloc permet la réutilisation de zones inutiles,

mais le fait de lier cette possibilité à la structure même du programme plutôt qu'à son déroulement réel est assez arbitraire, et PL/I ajoute un certain nombre d'ordres qui permettent le contrôle de cette réutilisation; l'inconvénient est le même qu'en Fortran, cela réclame beaucoup d'attention pour éviter les erreurs. En Jovial, l'équivalence entre zones de mémoires est définie de même façon qu'en Fortran, mais on peut aussi donner par un autre moyen un nom à un fragment de tableau, de structure ou même de chaîne, ce qui réalise un peu le même travail. En Cobol, il est possible de définir l'équivalence de fichiers, c'est-à-dire en fait de zones de lecture, ce qui permet de gros gains de place. Cependant, seuls les langages qui utilisent les structures de liste permettent une réutilisation complète et rationnelle des zones inoccupées. Il est vrai qu'ils le font au prix de la perte de place provoquée par les listes elles-mêmes. En IPL, Lisp, Slip, Comit et Snobol, tout l'espace disponible est organisé sous forme d'une liste, dans laquelle on puise des mots en fonction des besoins. Le plus gros problème est celui du retour à cette liste libre des mots devenus inutiles. Dans le cas de Comit et Snobol la chose est simple, car les listes utilisées sont sans aucun branchement, mais pour les trois autres langages le problème est beaucoup plus délicat, car une même donnée (une même liste) peut être utilisée en plusieurs endroits différents. IPL résoud le problème en distinguant deux types de listes, les listes régionales qui peuvent être utilisées par n'importe quelles autres et ne sont détruites que quand on le demande explicitement, et les listes locales qui ne sont utilisées que par une liste précise et disparaissent en même temps qu'elle. Lisp utilise le procédé du "ramassage d'ordures", qui considère comme utilisés tous les mots accessibles à partir d'un certain nombre de têtes de listes fixes, et récupère tous les autres. Slip utilise dans l'en-tête de chaque liste un compteur d'utilisation, qui indique combien de fois une liste est utilisée par d'autres listes, et permet d'effacer une liste dès que sont compteur est nul. La différence entre les procédés de Lisp et de Slip est donc

la suivante : en Lisp, quand la liste libre arrive à épuisement, on récupère tous les mots devenus libres pour refaire une nouvelle liste; en Slip, chaque fois qu'en prenant un mot dans la liste libre on trouve le nom d'une liste, on regarde si elle est devenue inutilisée, auquel cas on la rajoute à la liste libre.

5°) Utilisation de mémoires auxiliaires.

Je veux parler ici de la possibilité d'utiliser les mémoires lentes d'une machine, bandes, disques, tambours, etc...., comme un moyen de rangement normal quand il n'y a pas de place en mémoire rapide. Cette possibilité semble très intéressante car elle est beaucoup plus souple que l'utilisation de fichiers et surtout n'a pas à être programmée explicitement. Elle pose de très gros problèmes d'efficacité quand il faut éviter des temps de recherche prohibitifs. Seuls deux langages ont osé aborder le problème: il s'agit, de façon assez surprenante, de deux extrêmes, en l'occurence IPL et PL/I, et c'est IPL qui le fait de la façon la plus générale et la plus pratique. En PL/I la seule possibilité prévue est celle du rangement des tableaux de très grandes dimensions; ceci a un très gros inconvénient dans le cas où c'est le compilateur lui-même qui effectue le choix entre rangement lent et rangement rapide, et que l'on n'en tient pas compte dans l'écriture du programme. En IPL, deux choses différentes peuvent être rangées en mémoires auxiliaires: les structures traitées par un programme , tout d'abord, le rangement en mémoire lente étant spécifié par le programme, mais la recherche étant automatique lors de l'appel d'une structure, et aussi les sous-programmes eux-mêmes, qui sont appelés en mémoire rapide en cas de besoin, et dont le mode de rangement est indiqué au moment de leur chargement. Cette dernière possibilité, tout à fait exceptionnelle, semble particulièrement intéressante, mais ces deux facilités d'IPL ne semblent possibles que grâce à la structure même du langage et de ce qu'il traite.

CHAPITRE C - UTILISATION

1 - Opérations arithmétiques.

Le premier point qui va différencier les langages dans le domaine des opérations arithmétiques sera celui de l'écriture même d'une opération. Dans bon nombre de langages, on peut écrire une expression arithmétique complète sous une forme très proche de la notation algébrique normale, sans s'occuper de la façon dont elle sera évaluée en fait, et en particulier en tenant assez peu compte de l'ordre effectif des opérations, et pas du tout du rangement des résultats intermédiaires. Pourtant ceci n'est pas vrai pour tous les langages : en IPL, les opérations arithmétiques s'écrivent à un niveau très proche de celui du langage machine, avec l'obligation du rangement de tous les résultats intermédiaires. En Lisp, l'écriture est plus souple, et relativement simple si l'on se souvient qu'il s'agit d'une notation polonaise préfixée. En Comit, l'écriture des opérations arithmétiques se fait d'une façon extrêment lourde et sans aucun rapport, même lointain, avec celle d'une expression. En Snobol, l'écriture est plus agréable mais elle se limite à la notation opérande-opérateur-opérande, ce qui est à peine au-dessus du langage machine. Cobol permet l'écriture d'expressions arithmétiques à peu près normales, mais il utilise plus souvent une écriture verbeuse et horriblement lourde. A titre d'exemple, voici les diverses façons d'écrire l'instruction d'affectation à l'identificateur A de la valeur de l'expression (B+C) X D :

Algol : A:= (B+C) X D ; Fortran, Apt: A = (B+C) * D

Neliac: (B+C) X D → A ; Jovial : A = (B+C)*D ∅

PL/I : A = (B+C)*D; Cobol(1^e forme): CØMPUTE A = (B+C)*D.

Cobol(2^e forme) : ADD B TØ C GIVING M, MULTIPLY M BY D GIVING A.

Lisp : (SETQ A (TIMES (PLUS B C) D))

IPL : 10A0 ≠ 10D0 ≠ 00J90 ≠ 10C0 ≠ 10D0 ≠ 00J110 ≠ 00J112

Comit :* B+C = 1/.I*2+1+2 *≠* B+D = 1/.M.*2+2 */

* A+B = A /.*2 *

Snobol: M = B + C / A = M*D

(/ signifie "saut à la ligne)

2°) Utilisation.

Une fois définie l'écriture des opérations arithmétiques, reste à savoir à quoi elle peut s'appliquer. Nous allons retrouver ici la plupart des conventions et des restrictions que nous avions rencontrées lors de l'instruction d'affectation. Les diverses possibilités dépendent en effet de la variété des types reconnus par un langage, et des mélanges de types qu'il autorise dans une expression. Algol, Néliac IPL et Lisp ne connaissent que les types entier et flottant, et admettent l'écriture d'une opération portant sur un entier et un flottant en même temps : le nombre entier sera converti en représentation flottante avant l'opération. En Apt, en Snobol ou en Comit, il n'y a qu'un type possible donc aucun problème. En Cobol, le type d'une variable est quelque chose d'extrêmement mal défini, et l'on ne sait trop de quelle façon se fera une opération arithmétique. En Fortran et Jovial, les types sont nombreux mais généralement assez hermétiquement cloisonnés, et les mélanges ne sont pas toujours permis. Fortran admet une expression mixte comportant des nombres flottants, complexes ou en double précision, car cela n'implique aucune conversion, mais il interdit les écritures mixtes entier-flottant, ce qui est une de ses restrictions les plus gênantes et les plus graves, quoique expliquée par le souci d'optimisation. Jovial n'interdit définitivement le mélange que pour la variable duale, ce qui est normal, mais il a des difficultés à cause des nombres de longueur variable qu'il autorise, et qui peuvent, lors des diverses conversions au cours de l'évaluation d'une expression, occasionner une perte de précision exagérée. Le problème est un peu le même pour PL/I, qui a encore plus de types et de longueurs possibles, et doit utiliser des règles très complexes pour déterminer les divers ajustements et conversion nécessaires pendant l'évaluation d'une expression arithmétique. A ce sujet, deux langages permettent de spécifier la façon dont doit être approché le résultat d'une opération (tronqué ou arrondi); en Jovial, on l'indique pour chaque variable au moment de sa déclaration, en Cobol on l'indique pour chaque opération au moment du rangement de son résultat. Tous les autres langages

ignorent carrément le proclème, c'est-à-dire que c'est le compilateur qui décide de la façon dont le résultat sera approché.

2 - Opérations logiques.

1°) Opérations élémentaires.

Les opérations logiques élémentaires diffèrent d'un langage à l'autre, d'abord par leur nature même, mais aussi par ce à quoi elles s'appliquent: variables logiques ou variables booléennes (suivant la terminologie adoptée au chapitre B, paragraphe 2). Quoi qu'il en soit, tous les langages utilisant des variables logiques ou booléennes connaissent au moins les trois opérations élémentaires : union, intersection, négation. Algol ajoute l'implication et l'équivalence, Lisp l'union exclusive; PL/I reconnaît toutes les fonctions booléennes, mais sauf pour les trois principales il ne les considère pas comme des opérateurs simples (par exemple l'union exclusive sera BF0110, l'implication sera BF1101); il les applique, chiffre binaire par chiffre binaire, à des variables logiques. Pour Cobol et Lisp, qui sont sans variables booléennes véritables, mais qui utilisent cependant les relations, il est nécessaire de définir des opérateurs portant sur la valeur (booléenne) de ces relations : encore une fois on trouve l'union, l'intersection et la négation, dont Lisp possède donc deux exemplaires distincts de chaque. Sur les variables logiques assez particulières qu'il définit avec ses valeurs d'indices, Comit effectue des opérations logiques encore plus particulières: la complémentation n'est rien d'autre que la négation, mais la confluence est un compromis entre intersection et simple remplacement, dont la raison d'être ne s'explique guère.

2°) Relations.

Une relation n'est pas considérée par tous les langages comme une véritable expression booléenne, c'est-à-dire qu'elle ne peut se trouver affectée à une variable, que celle-ci soit booléenne ou logique. Ceci sera le cas en Fortran, Algol, Neliac, Jovial et Lisp; pour IPL, une comparaison a pour effet de modifier un registre de test, ce n'est absolument pas une expression portant une valeur; en PL/I, la

valeur d'une relation n'est pas une valeur booléenne puisque celle-ci n'existe pas, mais une chaîne de un chiffre binaire, qui pourra subir des opérations de concaténation: A < B || C = D peut être considéré comme un entier compris entre 0 et 3. Les six opérateurs de raltion sont utilisés dans Algol, Fortran, Neliac, Jovial, Cobol et PL/I, mais les écritures diffèrent : "supérieur ou égal" s'écrira " > " en Algol et Neliac, ".GE." en Fortran, "GQ" en Jovial, "GREATER THAN...ØR EQUAL TØ ...." en Cobol, >= en PL/I. En IPL et Lisp, la redondance de ces opérateurs est réduite, puisqu'ils ne reconnaissent que <, >, = et ≠ ; en Comit on trouve uniquement <, > et =; en Snobol enfin on ne peut que faire la différence des deux opérandes et regarder sa valeur ou son signe. Lors d'une comparaison, se posent les mêmes problèmes de conversions et d'incompatibilités de types que lors des expressions arithmétiques; ils sont résolus dans chaque langage de la même façon que pour ces dernières, ce qui est logique.

3°) Expressions conditionnelles.

Les différences entre les langages s'accentuent encore pour les expressions conditionnelles, suivant la généralité de leur forme et surtout suivant les possibilités d'utilisation. Une expression conditionnelle de la forme la plus générale est de type indéterminé, c'est-à-dire que son résultat pourra être le calcul d'une valeur logique, d'une valeur arithmétique, une adresse, ou l'exécution d'une instruction; cette dernière forme est la plus courante et la plus restreinte. C'est la seule que connaissent Fortran, Apt, Neliac, Jovial, PL/I et Cobol; ils l'écrivent d'ailleurs de façons très diverses: pour les quatre premiers, l'expression conditionnelle est en fait une instruction, dont l'effet est de sauter ou non l'instruction suivante; cela implique des difficultés dans le cas où un sinon suit le si, difficultés qui sont résolues d'une manière différente par chacun. Fortran et Apt ne connaissent pas le sinon, si bien que l'instruction contrôlée par l'expression conditionnelle est presque toujours une

instruction de transfert; Neliac impose l'existence du sinon dans tous les cas, si bien qu'il est fréquemment suivi d'une instruction vide; Jovial utilise deux formes différentes pour l'opérateur si (IF et IFEITH) suivant que celui-ci est suivi ou non d'un sinon; PL/I enfin est obligé d'imposer le sinon quand il est nécessaire d'établir les correspondances entre des si et des sinon imbriqués, d'une part parce que l'instruction gouvernée par le si est forcément terminée par un point-virgule, d'autre part, parce qu'elle peut à son tour être une instruction conditionnelle. Cobol utilise une syntaxe beaucoup plus générale, mais plus proche de la langue parlée c'est-à-dire moins rigoureuse et moins bien définie. IPL, Comit et Snobol ignorent les expressions conditionnelles, et finalement les seuls langages où l'on rencontre une expression conditionnelle complètement générale sont Algol et Lisp, la palme revenant d'ailleurs sans conteste à Lisp. En Algol, une expression conditionnelle sera booléenne, arithmétique ou de désignation, les deux première pouvant être affectées à des variables de même type, la troisième étant utilisée de façon beaucoup moins générale puisque Algol ne connaît pas d'étiquette variable ni d'instruction d'affectation à partie gauche conditionnelle ou bien sera une instruction conditionnelle, dont la forme n'est pas parfaite puisqu'il faut imposer une restriction pour interdire une ambiguité. En Lisp, une expression conditionnelle est une suite de doublets d'expressions; l'expression complète est balayée de gauche à droite, la "partie gauche" d'un doublet est évaluée, et si sa valeur est autre que le "faux" de Lisp (NIL), on évalue sa "partie droite", dont la valeur sera celle de l'expression complète, et on arrête le balayage; dans ceci, "partie gauche" et"partie droite" peuvent être n'importe quoi, puisqu'en Lisp toute expression est un appel de fonction et a une valeur, et en particulier l'une et l'autre pourront être à nouveau des expressions conditionnelles. Il semble vraiment difficile de faire mieux.

3 - Manipulations d'ensembles.

1°) Tableaux.

Nous avons vu au chapitre B la façon dont l'on pouvait, dans les langages qui les utilisent, se référer en tout ou en partie aux éléments d'un tableau; il reste à savoir maintenant comment l'on peut utiliser ces références. La chose est simple, puisque, à deux exceptions près, tous les langages qui utilisent les tableaux procèdent de la même manière: une fois appelé l'élément du tableau, suivant les conventions et en tenant compte des restrictions du langage propre, on considère cet élément comme une variable simple et on l'utilise exactement de la même façon, dans tous les cas. La première exception est PL/I; dans le cas où l'on se réfère effectivement à un élément simple, il procède comme les autres langages, mais il définit de nouvelles conventions dans le cas où l'on se réfère à un ensemble déléments, ou au tableau lui-même considéré non pas comme un bloc mais comme un ensemble de variables; les quatres opérations arithmétiques se font donc terme à terme, en ajoutant aux règles d'ajustage et de conversion des opérandes des règles d'ajustage des dimensions des tableaux que l'on traite; l'instruction d'affectation se fait également terme à terme, avec les mêmes règles d'ajustage; à une procédure qui attend comme paramètre un vecteur, on peut donner par exemple une ligne ou une colonne d'une matrice (ce dernier point semble être le plus digne d'intérêt). La seconde exception est Lisp; nous avons vu que la déclaration de tableau était une définition de fonction d'adresse; ceci fait que l'utilisation n'en est pas exactement identique à celle des autres langages, la différence se présentant lors de l'instruction d'affectation: pour ranger X dans l'élément de coordonnées I, J du tableau A, on n'écrira pas (SETQ (A I J) X), mais (A(QUØTE SET) X I J). En Lisp, les tableaux sont rangés sous la forme d'une liste de vecteurs de longueur fixe; une extension qui n'a jamais été faite prévoyait d'autres modes de rangements.

2°) Structures.

Comme nous l'avons déjà vu, seuls Jovial, PL/I et

Cobol définissent des structures, sous des formes d'ailleurs diverses. La principale différence entre structures et tableaux, au moment de l'utilisation, est que les opérations sur les structures portent normalement sur des opérandes complexes, de dimensions et de formes variables. Une simple opération dont les deux opérandes sont des éléments de structure pourra donc représenter un grand nombre d'opérations de recherche, de dégroupage, d'ajustement et de conversion. Ceci explique que ces opérations soient limitées à des types très particuliers, avec de nombreuses restrictions supplémentaires. Ainsi Jovial a mis l'accent sur les conversions de type et les équivalences ou recouvrements d'éléments de structure, ce qui lui permet de prendre n'importe quelle partie élémentaire d'une structure et d'en faire n'importe quoi: en revanche, le seul ensemble plus large qu'il puisse traiter est une "entrée de table"complète, c'est-à-dire en fait une structure considérée comme un bloc, et cela simplement pour la transférer ailleurs. Cobol au contraire attache plus d'importance aux noms des éléments de la structure et à leur niveau, une instruction spéciale lui permet ainsi d'effectuer un transfert de données avec réarrangement pour tenir compte de ces noms; en contre-partie, un élément de structure doit être rangé dans une variable de travail avant de pouvoir être effectivement utilisé. PL/I adopte les deux points de vue à la fois, en y ajoutant son aptitude à accepter des opérations arithmétiques dont les opérandes sont des ensembles, mais il en restreint beaucoup l'usage, pour supprimer les ajustements et les conversions et simplifier les recherches d'éléments correspondants, puisqu'il impose que les structures opérandes soient exactement identiques. Ainsi chacun des trois langages fait porter son effort sur un point différent, et le problème semble trop complexe et entraîner trop de conséquences pour permettre l'élaboration d'une solution générale.

3°) Chaînes.

Une chaîne ne constitue pas à proprement parler un ensemble d'éléments (sauf en Comit), mais sa principale caractéristique

est d'être de longueur indéfinie, et je ne considérerai donc pas comme chaînes les variables alphanumériques de Fortran, Jovial, IPL ou Lisp (même si en Lisp il est très facile de traiter une chaîne en considérant comme telle une liste d'atomes alphanumériques). Algol, quoiqu'il définisse des chaînes de façons assez générale, limite leur usage à l'apparition comme paramètres de procédures, on ne peut donc parler d'utilisation. Les chaînes que permet d'utiliser Cobol sont seulement des constantes particulières, c'est-à-dire que le seul usage que l'on puisse en faire est de les ranger dans une variable ou de faire des comparaisons. PL/I reconnaît le type chaîne comme un type parmi les autres, mais la seule opération spécifique qu'il définisse dessus est celle de concaténation, ce qui est au moins limité; le reste du temps, une chaîne sera considérée comme une variable logique, comme un nombre en écriture binaire, comme la représentation alphanumérique d'une variable quelconque, etc...., mais non pas vraiment comme une chaîne. Finalement deux langages seulement restent en lice mais eux sont au contraire spécifiquement orientés vers le traitement des chaînes. Cependant, Comit et Snobol ne traitent pas les même données. Comit ne traite que 128 chaînes différentes, mais une chaîne est une suite de constituants qui sont à leur tour des chaînes de caractères; cette façon de considérer deux niveaux dans toute chaîne permet de simplifier et d'accélérer notablement la plupart des opérations de reconnaissance. Pour Snobol au contraire, une chaîne n'est qu'une suite de caractères, mais le nombre de chaînes que l'on peut traiter est quelconque, et toutes ont un nom, ce qui permet de traiter un grand nombre d'éléments courts, et finalement permet là aussi d'accélérer les processus de recherche. Les opérations sur les chaînes définies en Comit et Snobol sont cependant pour la plupart semblables sinon identiques. Les principales se divisent en deux groupes: opérations de recherche et opérations de modifications. Les opérations de recherche se font toujours par application sur la chaîne d'un modèle plus ou moins imprécis et plus ou moins élastique, et balayage de la chaîne

jusqu'à la rencontre d'une coïncidence. Snobol a l'avantage de permettre la recherche d'une structure de longueur explicitement variable (et non pas indéfinie), Comit a l'avantage que l'opération de recherche ne modifie jamais rien dans la chaîne objet, alors qu'en Snobol les deux choses sont assez mal séparées. Les opérations de modification sont beaucoup plus nombreuses et diverses en Comit qu'en Snobol, d'une part pour tenir compte des deux niveaux qu'il distingue dans une chaîne, d'autre part pour pallier aux difficultés entraînées par ses chaînes en nombre fixe et hermétiquement cloisonnées. Enfin Comit définit un grand nombre d'opérations qui n'entrent pas dans l'une ou l'autre des catégories ci-dessus: échanges de chaînes, traitements en pile ou en file d'attente, recherche dans des dictionnaires, etc.....

4°) Listes

Les opérations portant sur les listes peuvent elles aussi être divisées en opérations de recherche et opérations de modification. Cependant IPL, Lisp et Slip sont très nettement séparés par la façon dont ils abordent le problème. IPL et Lisp sont deux langages dont le but n'est pas de traiter des listes mais de l'information symbolique. Ils utilisent abondamment les listes, systématiquement même dans le cas de Lisp, mais ceci est un moyen et non pas une fin. Ce qui les distingue est leur niveau par rapport à l'information traitée. IPL en est très près, et pour manipuler de l'information on manipulera donc explicitement des listes : les opérations définies partiront donc des plus élémentaires, et s'arrêteront à un niveau relativement bas, les opérations les plus complexes étant sans doute celles qui traitent effectivement des structures de liste complètes avec toutes leurs sous-listes. En Lisp au contraire, on peut ignorer totalement si on le désire que le support effectif du programme est un ensemble de listes. Les opérations définies sont donc plutôt destinées à l'usage propre du système Lisp, elles ont souvent pour objet des listes d'un modèle bien défini, généralement complexes, sur lesquelles elles effectuent un

travail important, les opérations élémentaires étant peu nombreuses et pratiquement jamais utilisées par un programme. A la différence des deux précédents, Slip est au contraire destiné exclusivement au traitement des listes pour elles-mêmes, étant admis que ce traitement peut servir à quelque chose. Il ne possède donc pas d'opérations d'un niveau très élevé, et en ce sens se rapproche plutôt d'IPL. Ce qui l'en sépare nettement est la symétrie qu'il introduit dans les listes, et qui fait que la plupart du temps toute opération est définie à double exemplaire, l'un traitant les listes vers le bas (vers la droite) et l'autre vers le haut (vers la gauche). Compte tenu de tout ceci, les opérations d'un niveau élémentaire ou moyen sont à peu près les mêmes dans les trois langages, IPI ayant servi de modèle aux deux autres. Chacun des trois langages possède cependant des opérations qui lui sont particulières, et dont certaines sont très importantes: en IPL, les générateurs, qui servent à appliquer une procédure donnée à tous les éléments d'une liste, d'un arbre ou d'une structure; en Lisp, les nombreuses opérations qui s'appliquent spécifiquement à un atome, c'est-à-dire à une liste de propriétés; en Slip, les opérations qui utilisent les lecteurs et permettent de traiter une structure "de l'extérieur", dans n'importe quelle direction, à travers les divers niveaux de sous-listes, et en ne s'arrêtant qu'à des objets de nature précise.

4 - Déroulement du contrôle.

1°) Instructions de transfert.

Quoique l'une des plus courantes et des plus simples, l'instruction de transfert n'est pas sans poser de problèmes, car il n'est pas possible de transférer le contrôle n'importe où, et les restrictions sont importantes. Elles peuvent se diviser en deux catégories: celles qui sont liées à l'évolution dynamique du programme et celles qui sont liées à sa structure statique. Dans les premières on rencontre surtout l'interdiction d'entrer de l'extérieur au milieu d'une boucle répétitive (car la variable de contrôle serait indéfinie; ceci vaut pour tous les langages qui utilisent la boucle répétitive).

Dans les secondes on trouve l'interdiction d'entrer au milieu d'une procédure ou d'un sous-programme (à cause de l'appel des paramètres effectifs; PL/I tourne la difficulté en autorisant plusieurs points d'entrée dans une procédure), d'entrer dans un bloc autrement que par le début (à cause des déclarations dynamiques; cependant CPL admet cette possibilité). Les interdictions dynamiques sont évidentes et impossibles à supprimer, les restrictions statiques peuvent être levées moyennant certaines conventions.

Ceci vu, le transfert de contrôle peut se faire de nombreuses façons, mais l'instruction a toujours la forme "allerà" "expression d'adresse", et les divers types d'instructions se distinguent par la forme de cette expression d'adresse. La forme la plus simple est celle d'une étiquette unique: elle est utilisée par tous les langages, c'est la seule qui soit reconnnue par Apt, IPL, Lisp, Comit et Snobol. L'étiquette en question peut être désignée par son nom (ce sera donc une constante), mais dans certains cas elle peut être obtenue indirectement, en trouvant son nom dans une certaine variable. Ce dernier procédé est utilisé par PL/I, qui reconnaît la variable étiquette, par Comit et Snobol, pour qui l'étiquette est une chaîne que l'on peut chercher ailleurs que dans le programme, par IPL, qui admet aussi un double adressage indirect. Les plus pauvres sont Apt, pour qui l'instruction de transfert géométrique (déplacement de l'outil) est plus importante que le transfert arithmétique (modification du déroulement linéaire du programme), et Lisp, pour qui la linéarité d'un transfert de contrôle est assez incompatible avec la structure complètement parenthésée du langage.

La deuxième forme de l'expression d'adresse est celle de l'aiguillage, qui peut se présenter de façons très diverses. L'aiguillage peut être un tableau d'adresses dans lequel on fait un choix au moyen de la valeur d'un indice; ce moyen est utilisé par Fortran, de façon lourde et assez pauvre, par Algol, Neliac, Jovial et Cobol,

d'une façon qui a l'inconvénient d'être complètement statique (1), par PL/I enfin, de la façon la plus générale puisqu'une étiquette est une variable comme une autre. Des procédés plus particuliers peuvent être aussi utilisés : le "allerà" imposé de Fortran utilise une véritable instruction d'affectation d'étiquette, mais sa généralité est très restreinte; Jovial permet de lier un aiguillage à une variable déterminée, et d'attacher ainsi un transfert à certaines valeurs choisies de cette variable (ceci est encore une fois complètement statique); Algol permet d'utiliser partout où doit figurer une expression d'adresse l'expression de désignation conditionnelle, qui permet de rendre dynamique le choix du transfert; en Cobol enfin il est possible de modifier à distance une instruction de transfert, par un moyen qui est en fait très apparenté au "allerà" imposé de Fortran, même si les formes extérieures sont différentes.

2°) Boucles répétitives.

La boucle répétitive est un instrument puissant des langages qui l'utilisent, quoique elle ne soit théoriquement qu'une facilité d'écriture, puisque le processus qu'elle rend implicite pourrait presque toujours être écrit explicitement au moyen des instructions élémentaires du langage qui l'utilise. En fait, elle est conçue pour gagner du temps non seulement lors de l'écriture du programme, mais aussi, quoique pas toujours, lors de son exécution, par l'automatisation et l'optimisation des opérations qu'elle entraîne. Disons tout de suite qu'on ne la rencontre que dans les langages destinés aux calculs numériques, et que Apt, IPL, Lisp, Comit et Snobol n'en possèdent donc pas. Trois éléments principaux doivent être considérés dans l'instruction qui gouverne une boucle répétitive: la variable contrôlée, l'ensemble des valeurs qui lui sont attribuées, et l'ensemble des instructions répétées.

(1) sauf en Algol, où la dynamique peut réapparaître par le biais de l'expression de désignation conditionnelle, évaluée au moment de l'appel de l'aiguillage.

(a) Variables contrôlée.

C'est toujours un entier pour Fortran, Neliac et Jovial, c'est une variable arithmétique quelconque pour les autres. En Neliac et Jovial, c'est même une variable d'un type spécial, que l'on ne peut rencontrer nulle part ailleurs et qui n'est jamais déclarée. En Fortran il peut s'agir d'un entier quelconque, mais il est traité de façon très spéciale, et en particulier ne peut être modifé par les instructions qui composent la boucle.

(b) Valeurs de la variable.

L'ensemble des valeurs que prend la variable contrôlée est spécifié dans l'instruction de répétition sous un grand nombre de formes diverses. La forme la plus simple consiste à indiquer explicitement la ou les valeurs que doit prendre la variable : ceci est possible en Algol, Cobol et PL/I, et aussi en Jovial, mais dans ce dernier cas les instructions qui composent la boucle sont effectivement répétées pour cette unique valeur. La forme la plus courante consiste indiquer pour la variable une valeur initiale, une valeur d'incrémentation, et un test d'arrêt, qui peut lui-même prendre diverses formes. Ce test d'arrêt peut constituer en une valeur finale de la variable de contrôle: c'est la seule forme utilisée par Fortran et Neliac, elle est aussi reconnue par Algol, Jovial et PL/I. Fortran limite beaucoup l'utilisation de cette forme puisqu'il impose que la valeur initiale soit supérieure à zéro, que le pas soit positif, et que ces deux valeurs et la valeur finale soient données par une expression particulièrement simple, et non modifiées pendant l'exécution de la boucle. Les quatre autres langages sont beaucoup moins restrictifs, ils admettent des valeurs quelconques (cependant en Jovial la variable de contrôle doit toujours rester positive), de type également quelconque quoique arithmétique, et données par des expression arithmétiques générales. Le test d'arrêt peut être aussi une véritable expression conditionnelle. Ceci est possible en Cobol (c'est sa forme générale), en PL/I et en Algol,

avec pour ce dernier la restriction que l'arrêt par vérification d'une condition est incompatible avec l'incrémentation automatique de la variable de contrôle; cette restriction n'est jamais gênante, mais elle reste assez inexplicable. Dans ces trois langages également, il est possible de donner une liste des valeurs à attribuer successivement à la variable de contrôle, certaines étant des valeurs simples, d'autres des expressions avec incrémentation automatique et test d'arrêt de l'une ou l'autre forme. Une dernière possibilité existe en Jovial, c'est celle de faire attribuer automatiquement à la variable de contrôle toutes les valeurs qui permettent le balayage complet d'une table donnée.

(c) Instructions répétées.

En Fortran, on doit spécifier dans l'instruction de répétition une adresse qui est celle de la dernière instruction à répéter, ce qui impose, lorsque cette instruction est d'un type particulier (non exécutable ou conditionnelle), d'ajouter une instruction ineffective à la fin. Pour Neliac, les instructions sont forcément regroupées en une instruction composée, même s'il n'y en a qu'une. Pour Jovial et Algol, l'instruction de répétition ne gouverne qu'une seule instruction, mais celle-ci peut être composée. Pour PL/I, l'instruction de répétition constitue une ouverture d'instruction composée, qui peut être fermée n'importe où par "l'instruction" END, avec les particularités que celle-ci présente. Le DØ est utilisé seul comme ouverture d'instruction composée, afin de le distinguer de BEGIN qui ouvre un bloc, mais cela semble n'être qu'un report de confusion au lieu d'une clarification. En Cobol, deux formes sont possibles; dans la première, l'ensemble d'instructions gouvernées par la boucle normale s'arrête au premier point, symbole de fin d'instruction composée; la deuxième forme est utilisée avec le verbe spécial PERFØRM, les instructions répétées constituant un ensemble fermé placé ailleurs dans le programme (et qui peut être d'ailleurs être exécuté "en passant" à sa place normale), et désigné seulement par son nom; cette dernière forme peut ne pas utiliser de contrôles, par simple indication du nombre de

répétitions. Les instructions elles-mêmes gouvernées par la boucle répétitive sont dans certains cas soumises à des restrictions. Tout d'abord, s'il s'agit à leur tour de boucles itératives, tous les langages imposent la complète inclusion, c'est-à-dire qu'une boucle ne peut se terminer (de façon statique, c'est-à-dire que le saut reste possible) sans que toutes les boucles intérieures ne l'aient été auparavant; de plus, Fortran, Neliac et Cobol restreignent le nombre d'inclusions, généralement à trois, et pour Cobol toutes les évolutions d'indices se font au début de l'ensemble d'instructions. Jovial reconnaît une forme particulière, qui lui permet de faire évoluer des indices parallèlement, le premier étant celui sur lequel porte le test d'arrêt; une instruction spéciale lui permet de plus, au milieu des instructions gouvernées par la boucle, de forcer l'incrémentation de la variable de contrôle (ou des variables s'il y en a plusieurs en parallèle) et le test d'arrêt.

5 - Sous-programmes.

1°) Déclaration.

Les langages qui utilisent les sous-programmes en temps que tels sont Fortran, Algol, Jovial, Neliac, PL/I, IPL et Lisp. IPL est à laisser de côté du point de vue des déclarations, car tout programme IPL étant en fait un sous-programme, il n'y a pas de particularité à distinguer; Lisp présente une autre particularité, qui est que tout sous-programme est une fonction. A part IPL, tous les langages utilisent une forme à peu près semblable pour la déclaration. Celle-ci est d'ordinaire statique, sauf pour Lisp, où l'on déclare une fonction en la donnant comme paramètre effectif à la fonction DECLARE, et où l'on peut toujours modifier sa définition en cours de programme. Dans tous les cas cependant, la déclaration se compose d'un en-tête et d'un corps. L'en-tête contient :

1-) le nom du sous-programme, ce qui permet d'ailleurs de lui donner plusieurs noms;

2-) la liste des paramètres formels, où Jovial établit une distinction entre paramètres d'entrée et paramètres de sortie;

3-) la liste des spécifications, c'est-à-dire des indications sur la nature des paramètres et sur leur traitement (Lisp n'en a pas besoin) en fait , seul Algol distingue les spécifications des paramètres formels des déclarations de variables locales au sous-programme. Dans le cas ou le sous-programme est une fonction, il faut encore indiquer son type, ce que Algol, Fortran et Neliac font avant le nom du sous-programme, Jovial et PL/I parmi les autres spécifications.
Le corps est l'ensemble des instructions qui constituent le sous-programme; c'est une instruction simple pour Algol, donc le plus souvent une instruction composée ou un bloc, une instruction composée pour Neliac et Jovial, un ensemble d'instructions terminées par "l'instruction" END pour Fortran et PL/I. Quand un sous-programme est bien au point, il est intéressant qu'il ne soit pas compilé à nouveau chaque fois qu'il est nécessaire. La compilation séparée des sous-programmes est prévue explicitement en Fortran et en PL/I, moyennant dans ce dernier des conventions, des restrictions et des spécifications supplémentaires pour permettre la compatibilité avec la structure de blocs et la portée des déclarations. Elle est relativement facile en Jovial puisque ces deux problèmes ne se posent pas, et que tout identificateur doit être déclaré. Elle est beaucoup plus difficile en Algol, et les mêmes restrictions qu'en PL/I sont nécessaires. Tous ces langages utilisent de plus un certain nombre de sous-programmes "de bibliothèque" mais ils font partie intégrante du compilateur, y sont étroitement liées, et je n'en parlerai donc pas.

2°) Appel des paramètres.

L'activation d'une fonction est provoquée par l'apparition de son nom, avec la liste de paramètres effectifs s'il y a lieu, dans une expression arithmétique; celle d'un sous-programme est provoquée en Algol, Neliac et Jovial par son simple nom, alors qu'en Fortran et PL/I on doit le faire précéder de CALL. Deux problèmes se posent lors du remplacement des paramètres formels par les paramètres effectifs: le premier est celui de la correspondance des types et des

conversions éventuelles, il a été vu précédemment; le second est celui du choix entre appel par nom et appel par valeur. Dans l'appel par nom, toute utilisation du paramètre formel correspond à un traitement du paramètre effectif lui-même : cela est très simple dans le cas où ce paramètre est une variable simple, mais dans le cas où il s'agit d'une variable indicée, ou d'une expression arithmétique, qui peut d'ailleurs contenir des appels d'autres fonctions, avec toutes les possibilités d'effet de bord que cela suppose, le problème devient extrêment complexe, et beaucoup de langages ont reculé devant la difficulté de mise en oeuvre. En Algol et PL/I il est possible d'indiquer, au moment des spécifications, si un paramètre formel doit être traité par valeur ou par nom. En Lisp, tous les paramètres systématiquement sont appelés par nom, mais cela n'est guère possible que grâce à la structure même du langage. En Jovial, tous les paramètres sont appelés systématiquement par valeur, et les paramètres d'entrée sont distingués des paramètres de sortie au moment de la déclaration et au moment de l'utilisation. En Fortran et Neliac il n'y a pas de règles fixes, en général les variables simples sont appelées par nom et tout le reste par valeur; ceci n'est donc déterminé qu'au moment de l'utilisation des sous-programmes. En IPL, chaque sous-programme trouve ses paramètres empilés dans le registre de communication commun, et c'est à lui de décider si ce qu'il trouve est sa valeur, son adresse ou l'adresse de la fonction permettant de le calculer: tout est donc permis, mais rien n'est vérifié automatiquement. Dans la plupart des sous-programmes, certains paramètres ne sont pas communiqués par l'intermédiaire de la liste de paramètres mais extérieurement. Cela pose des problèmes dans le cas où ces sous-programmes peuvent être compilés séparément; en Fortran et Lisp, on utilise le procédé des zones communes, Lisp utilisant en plus des variables dites "spéciales", qui peuvent d'ailleurs être détruites quand on n'en a plus besoin; pour Neliac, Algol, Jovial et PL/I, on déclare ces paramètres à un niveau englobant le programme et le sous-programme, mais ceci est très difficilement adaptable à la

compilation séparée; PL/I utilise donc de nombreuses spécifications supplémentaires diverses, qui reviennent à peu près au même que les zones communes de Fortran.

3°) Particularités des sous-programmes.

La récursivité dans la définition ou l'appel des sous-programmes est plus qu'un artifice théorique et Lisp montre suffisamment son importance. Cependant il s'agit évidemment d'un problème très complexe, surtout dans le cas où il est imbriqué avec les possibilités d'appel des paramètres par nom. Fortran, Neliac et Jovial l'interdisent carrément, mais les deux procédures VISIT et TERM de Slip permettent de l'introduire en Fortran. PL/I le prévoit mais réclame dans la déclaration de sous-programme un attribut spécifiant qu'il peut être appelé récursivement. Algol et Lisp ne réclament aucune spécification et prévoient la récursivité explicitement. IPL enfin effectue la chose automatiquement grâce au procédé de communication des paramètres. PL/I permet de définir plusieurs points d'entrée à un même sous-programme, ce qui est utile principalement pour permettre l'utilisation de listes de paramètres différentes. Jovial admet l'omission de paramètres effectifs dans l'appel, ce qui revient souvent au même. Fortran, Jovial, PL/I et Lisp utilisent l'instruction spéciale de retour qui permet de sortir d'un sous-programme avant sa fin physique. Fortran permet la définition de fonctions spéciales, les fonctions arithmétiquement définies, dont le corps n'est qu'une expression arithmétique, et qui sont inaccessibles de l'extérieur du programme où elles sont définies. Jovial et Neliac utilisent les "instructions fermées", qui sont des sous-programmes sans paramètres, activés par une simple instruction de transfert, mais qui ne peuvent être exécutés "au passage" comme les procédures de Cobol.

6 - Facilités diverses.

1°) Utilisation du langage de la machine.

Il est un point admis généralement, qui est que le langage machine est toujours plus puissant que n'importe quel langage à

plus haut niveau, et que l'inclusion dans un programme d'une séquence ou d'un sous-programme écrit en langage machine est souvent utile sinon indispensable. Cela n'est pourtant pas possible dans tous les langages : Apt, Comit et Snobol ignorent carrément le problème, PL/I juge qu'il est lui-même plus puissant que le langage machine et que cette possibilité est donc inutile. Les autres langages procèdent de façons diverses : en Fortran et en IPL, le langage machine ne peut être inclus que dans un sous-programme extérieur au programme normal; pour Fortran, les possibilités de communication dépendent donc énormément du compilateur et du système d'exploitation. En Algol, le langage machine ne peut figurer que dans le corps d'une procédure, ce qui revient à peu près au même. En Lisp, le langage machine peut être utilisé dans les définitions de fonctions, par l'intermédiaire d'un assembleur spécial au langage. En Jovial, Neliac et Cobol enfin, des séquences de langage machine (avec le plus souvent des restrictions) peuvent être insérées entre deux crochets spéciaux (par exemple DIRECT et JOVIAL en Jovial, ENTER ASSEMBLY-PROGRAM. et ENTER COBOL. en Cobol).

2°) Auto-modification d'un programme.

La modification d'un programme par lui-même n'est possible que si les données traitées sont de la même forme que le programme. Ceci impose donc que le langage soit un langage machine ou qu'il soit interprété. C'est donc a priori impossible pour Fortran, PL/I, Cobol et Comit, et pour les versions compilées des autres langages. Cela semble très difficile pour Algol, Neliac ou Jovial même s'ils sont interprétés, car cela suppose une grande connaissance du compilateur lui-même. Seuls IPL et Lisp restent en course, IPL étant le maître en ce domaine à cause de sa structure de langage machine : un programme IPL peut construire des sous-programmes qu'il exécutera ensuite, il peut se modifier lui-même, s'amputer, se multiplier, etc.... En Lisp, les possibilités sont à peu près les mêmes, quoique placées à un niveau beaucoup plus élevé. Il est donc facile à un programme Lisp de résoudre l'exemple suivant: lire l'expression symbolique d'un polynôme; en cal-

culer toujours symboliquement la dérivée; évaluer la valeur de cette dérivée pour des valeurs données des variables. C'est évidemment le dernier point qui est le plus difficile dans un autre langage.

3°) Ecriture de compilateurs au moyen d'un langage.

L'écriture de compilateurs dans un langage à haut niveau est un procédé extrêmement intéressant au moins du point de vue théorique, le plus intéressant étant le cas où l'on écrit dans un certain langage le compilateur de ce propre langage. Ce procédé a été utilisé systématiquement pour Neliac, très largement pour Jovial, et également en Lisp. Une fois écrit un compilateur très simplifié en langage machine, on écrit dans le langage source un compilateur plus évolué, qui est compilé par le précédent; ce procédé peut être réitéré au fur et à mesure que l'on perfectionne le compilateur ou que l'on désire le modifier. En Lisp, le problème est un peu différent: le compilateur écrit en Lisp a été interpété par l'interpréteur Lisp puis chargé de se compiler lui-même afin de générer le véritable compilateur. Des compilateurs nombreux ont été écrits en Algol, Neliac, Jovial et Lisp qui sont les langages qui s'y prêtent le mieux. Ceci permet d'écrire sur une machine un compilateur pour une autre, avec l'inconvénient évident qu'au fur et à mesure que le procédé est renouvelé le programme objet est de plus en plus mauvais.

4°) Possibilités particulières à PL/I.

La partie la plus intéressante de PL/I est probablement ce qu'il contient d'original; ces possibilités peuvent se répartir en deux groupes: celles qui sont destinées à adapter le langage aux nouvelles possibilités des machines, et celles qui constituent des nouveautés au niveau même de la compilation. Les problèmes de multiprogrammation et de temps réel sont explicitement prévus en PL/I; à cette fin, il est possible de fractionner l'exécution d'un programme en diverses tâches qui s'initialisent les unes les autres, se déroulent simultanément et peuvent s'attendre mutuellement; le lancement d'une tâche se fait en particulier lors d'une opération d'entrée-sortie et

lors d'un appel de sous-programme; il est possible à une tâche d'attendre un signal donné d'une autre tâche avant de continuer son propre travail. La particularité la plus intéressante du procédé de compilation est que certaines instructions du langage source peuvent être destinées uniquement au compilateur, et servir par exemple à modifier des conventions, à définir de nouvelles instructions, à propvoquer la génération de séquences d'instructions complètes; ces instructions destinées au compilateur peuvent arriver à constituer de véritables programmes, avec déclarations, instructions d'affectation, de transfert, etc..... qui sont exécutées au moment même de la compilations, et nécessitent un pré-traitement du programme source.

C O N C L U S I O N S

Pour conclure brièvement, je signalerai simplement les points qui me semblent les plus marquants et les plus intéressants dans chacun des langages étudiés. Une table et un graphique, qui ont l'inconvénient évident de ne pouvoir être que grossiers et imparfaitement objectifs, permettront de juger d'un coup d'oeil de certains aspects intéressants.

FORTRAN : est un langage qui malgré ses adjonctions diverses accuse nettement son âge. Il reste cependant, et de très loin, le plus employé de tous les langages de programmation. Sa caractéristique la plus importante est qu'il est conçu dans le but de l'optimisation la plus grande possible du langage objet. Il est en général vite et facilement appris, à part les restrictions diverses qui restent extrêmement gênantes. La compatibilité des programmes d'une machine à l'autre est généralement bonne, car les incompatibilités sont le plus souvent dues à des adjonctions.

ALGOL : est un langage très discuté par les utilisateurs non proprement scientifiques. Le principal reproche que lui ont fait les habitués de Fortran semblerait pourtant un peu secondaire: c'est celui de la difficulté de compilation séparée des procédures. Ses deux points les plus marquants semblent être la rigueur d'écriture et la facilité de lecture, qui le rendent sans rival pour la communication d'algorithmes. Ses deux manques les plus graves sont probablement la définition extrêmement tardive des procédures normales d'entrée-sortie, et la carence totale quant au traitement des chaînes, des nombres complexes et en double précision. C'est un langage qui s'apprend assez facilement, quoique la possession totale de toutes ses finesses soit plus longue à acquérir.

JOVIAL : est un langage lui aussi assez ancien. Il a pris beaucoup de choses à Algol, mais ignore la structure de blocs qui est l'une des plus intéressantes. Il est très riche sous le rapport de la définition et du traitement des données, en particulier pour les tables. Il présente deux gros défauts : sa complication d'une part, et d'autre part le fait qu'aucun compilateur ne comprenne le langage complet, et que le sous-ensemble commun à toutes les définitions soit très pauvre. Ses points principaux sont vite appris, mais les multiples détails ajoutent beaucoup de difficultés.

NELIAC : représente en fait tout un groupe de langages, de la même famille mais souvent très différents. Leurs caractéristiques communes sont qu'il s'agit de dialectes d'Algol, et que l'écriture en Neliac de compilateurs Neliac est très facile. Les avantages marquants sont la simplicité et la rigueur de la syntaxe, la facilité des modifications à la définition d.. langage.Cet avantage devient un grave inconvénient quand il entraîne une totale incompatibilité entre machines et le manque de certitude sur la véritable définition du langage.

PL/I : est un langage très ambitieux puisqu'il veut remplacer tous les langages, y compris celui de la machine. Le résultat le plus évident est qu'il est d'une complexité qui en rend à peu près impossible la possession complète. Il est prévu la définition de sous-ensembles du langage, mais ceci peut mener à une véritable scission en plusieurs langages, qui seraient parents d'Algol, Jovial ou Cobol. Les deux points les plus intéressants sont probablement l'utilisation des procédés de multiprogrammation et de temps réel, et l'ensemble des moyens d'instructions destinées au compilateur. Un inconvénient grave pour un langage qui veut être universel est que PL/I est très fortement adapté à la structure des machines du Système 360.

COBOL : est un langage particulièrement adapté aux problèmes de gestion et surtout de traitements de fichiers, il a le malheur d'être un peu trop spécialisé. Ses inconvénients les plus évidents sont son épouvantable verbosité et ses restrictions souvent très arbitraires. C est un langage facile à apprendre, et un programme sera très facile à relire, mais les restrictions sont très gênantes et la compatibilité entre machines est très réduite, en particulier à cause de l'impossibilité d'une description formelle complète.

APT : est un langage très à part des autres étant donnée son utilisation. Dans les domaines qu'il conserve en commun avec les autres langages, on peut noter parmi les inconvénients sa syntaxe un peu enfantine et ses conventions d'écriture fort peu évidentes, parmi les points intéressants tout ce qui concerne les définitions et l'utilisation de concepts géométriques. Il semble d'une utilisation suffisamment simple pour être employé avec profit par un programmeur non spécialisé.

IPL : est un langage très complet mais à niveau très bas, et on ressent souvent le manque de la machine qui l'utiliserait directement, sur laquelle on pourrait alors écrire très simplement des compilateurs, et utiliser des langages de liste plus évolués. Il présente tous les avantages et tous les inconvénients d'un langage machine, avec en plus l'avantage d'être très utilisé et l'inconvénient d'être très ancien.

LISP : permet théoriquement de faire n'importe quoi, mais l'écriture est parfois lourde, et surtout très difficile à relire: on aimerait pouvoir écrire un programme en utilisant le méta-langage Lisp qui élimine des ennuis. Lisp est jugé généralement comme un langage très difficile , mais il me semble en réalité de difficulté moyenne, et particulièrement bien adapté à ce qu'il veut faire. Il présente l'inconvénient d'être relativement ancien, et surout mal connu à cause du mythe qui s'y rattache, de la rareté de la documentation et des compilateurs.

SLIP : présente le principal inconvénient de ne pas être un véritable langage. Il est par conséquent très lié, malgré les efforts de son auteur, au langage qui l'héberge, soit qu'il exploite certaines incertitudes sémantiques de ce langage hôte (en Fortran, l'absence du contrôle de type des paramètres effectifs), soit qu'il se trouve soumis à ses diverses restrictions (en Fortran, le principal ennui est celui des expressions mixtes, et un programme reste toujours à la merci de conversions inopportunes). En revanche, Slip contient plusieurs possibilités très intéressantes et très nouvelles sur les listes et leur traitement, et gagnerait donc sûrement à être un véritable langage.

COMIT : présente certains gros défauts, comme l'absence de désignation symbolique des chaînes, l'impossiblité du contrôle de la structure effective des données , l'arithmétique catastrophique. En revanche, la plupart des traitements de chaînes sont très faciles et très commodes, et les possiblités nombreuses. Comit est très adapté à son sujet, presque trop, un peu vieux et assez peu répandu, mais c'est le plus facile à apprendre de tous les langages de programmation. Comit II doit remédier à certains de ses gros inconvénients.

SNOBOL : possède pour le moment une syntaxe extrêmement simple, mais simple jusqu'à la pauvreté. Il ajoute à Comit l'énorme supériorité de nommer les chaînes au moyen d'autres chaînes, mais ignore toutes les possiblités de Comit autres que la recherche de structures: instructions de corps, aiguilleux, rayons, listes ou piles, comptages, indices, règles de liste, etc.... La définition rigoureuse est facile pour le moment, mais le langage est en évolution, car il est pour l'instant réduit à la plus simple expression possible pour un langage.

T R O I S I E M E P A R T I E

COMPARAISON ENTRE QUELQUES PROGRAMMES

COMPARAISON ENTRE QUELQUES PROGRAMMES.

Ces programmes ont été choisis très brefs mais suffisamment typiques, afin de permettre de juger d'un coup des ressemblances et des différences des langages. Dans chaque domaine, je n'utilise que les langages qui sont capables de traiter le problème donné de façon simple et directe. Pour terminer, je donne pour chaque langage un exemple de programme plus long, qui utilise si possible certaines des possibilités les plus intéressantes et les plus caractéristiques de ces langages.

1 - Programme de calcul différentiel.

Il s'agit d'une simple procédure et non pas d'un programme complet.

1°) Algol

```
procédure Euler (fct) résultat:(somme) précision:(epsilon) arrêt:(nb max);
     valeurs epsilon, nb max; entier nb max; réelle procédure fct;
     réels  somme, epsilon;
     début entiers i,k,n,t; tableau m [0:15] ;réels mn,mp,ds;
          i:=n:=t:=o; m [0] := fct (0); somme := m [0] /2;
 terme suivant : i:=i+1; mn:= fct (i);
          pour k:= 0 pas 1 jusqu'à n faire
          début mp:= (mn+ [k] )/2, m [k] :=mn; mn:= mp fin;
          si (abs(mn)< abs (m [n] ) ∧ (n<15) alors
          début ds:= mn/2; n:=n+1; m [n] := mn fin  ça marche
          sinon ds:= mn; somme := somme +ds;
    t:=   si  abs (ds)<epsilon alors  t+1 sinon 0;
          si t<nb max alors allerà terme suivant
fin  de la procédure Euler;
```

2°) Neliac.

```
procédure Euler : { entier t, tableau m [ 16] , réels mn, mp, ds ;
        0 → i → n → t, fct (0) → m [ 0], m [ 0] /2 → somme;
terme suivant : i+1 → i, fct (i) → mn;
        pour k=0 pas 1 jusqu'à n faire
        { (mn+m [ k])/2 → mp, mn → m [ k], mp → mn},
        si (abs (mn) < abs (m [ n]) ) ∧ ( n<15) alors
          { mn/2 → ds, n+1 → n, mn → m [ n ]}; sinon mn → ds;
        somme + ds → somme,
        si abs (ds) < epsilon alors t+1 → t; sinon 0 → t;
        si t < nb max alors allerà terme suivant; sinon ; } ;
```

3°) Fortran.

```
      SUBRØUTINE  EULER (FCT, SØMME, EPS, NB MAX)
      DIMENSIØN  M (16)
      REAL MN, MP, M
      I = 1
      N = 1
      IT = 1
      M(1) = FCT (0)
      SØMME  = M(1)/2.
C     TERME SUIVANT
1     I = I+1
      MN = FCT (I)
      DØ 2 K = 1,N
      MP = (MN + M (K) )/2.
      M(K) = MN
2     MN = MP
      IF ( (ABS(MN).LT.ABS (M(N) ) ). AND. (N.LT.15) ) GØ TØ 3
```

```
                DS = MN
                GØ TØ 4
3               DS = MN/2.
                N = N+1
                M(N) = MN
4               SØMME = SØMME + DS
                IF (ABS(DS). LT. EPS) GØ TØ 5
                IT = 0
                GØ TØ 6
5               IT = IT + 1
6               IF (IT.LT. NB MAX) GØ TØ 1
                RETURN
                END
```

4°) Jovial.

```
PRØC  EULER (FCT,EPSILØN, NBMAX = SØMME) $/ ITEM EPSILØN F $ ITEM NBMAX
    I 10 U $ ITEM SØMME  F $
BEGIN ARRAY MM16 F R $/  ITEM II 10 U $/  ITEM MN 10 U $ ITEM TT 10 U $
     ITEM MN F $  ITEM MP F $/  ITEM DS F $
     II = MN = TT = 0 $ MM ($ 0 $) = FCT (0) $ SØMME = MM ($ 0 $)/2 $
TERME'SUIVANT. II = II +1 $  MN = FCT -II) $/
     FØR K = 0, 1, MN $
     BEGIN MP = (MN + MM ($ K $) )/2 $ MM ($ K $) = MN $
                MN = MP $END
     IFEITH (ABS ( MN) LT ABS (MM($NN$) ) ) AND (NN LT 15) $
                BEGIN DS = MN/2 $ NN=NN+1 $ MM ($ NN $) = MN END $
     IFEITH ABS (DS) LT EPSILØN $ TT = TT+1 $ØRIF 1 $TT = 0 $
     IFIT LT NBMAX $ GØTØ TERME'SUIVANT $ END  END
```

5°) PL/I

```
EULER: PRØCEDURE (FCT, SØMME, EPSILØN,NBMAX) ;
BEGIN; DECLARE (M(0:15), MN, MP, DS) REAL, (I,K,N,T) INTEGER ;
          I=N=T=0; M(0) = FCT (0); SØMME = M(0)/2;
TERME_SUIVANT : I = I+1; MN = FCT(I);
          DØ K=0 TØ N; MP = (MN+M(K))/2; M(K)=MN; MN=P;END;
          IF (ABS(MN)<ABS(M(N)) & (N<15) THEN  DØ;
                    DS = MN/2; N=N+1;M(N)=MN; ELSE DS=MN;
          IF ABS (DS)<EPSILØN THEN T=T+1; ELSE T=0;
          IF T<NBMAX  THEN GØTØ  TERME_SUIVANT;
          RETURN; END EULER;
```

2 - Programme de calcul matriciel.

Il s'agit d'un programme très simple qui lit deux matrices rectangulaires, fait leur produit et écrit le résultat. Ce programme ainsi que certains des suivants utilise des procédures d'entrée-sortie, qui ne sont pas des procédures standards en Algol, Neliac et Jovial.

1°) Algol.

```
début entiers m, n, p; LIRE (m, n, p);
     début tableaux A[1:m, 1:n], B[1:n, 1:p],C[1:m,1:p];
          réels S; entiers i, j, k ;
          ENTREE (5, tableau, m×n, A[1,1]);
          ENTREE (5, tableau, n×p, B[1,1]);
     tableau : MØDELE ('(6E12.5)');
          pour i:= 1 pas 1 jusqu'à m faire
          pour j:= 1 pas 1 jusqu'à n faire
          début S:=.0; pour k:= 1 pas 1 jusqu'à p faire
               S:= S+A[i,k]×B[k,j] ; C[i,j] := S   fin;
     écriture: MØDELE ('(10 (1XE12.5) )');
          pour i:= 1 pas 1 jusqu'à m faire
          SORTIE (6, écriture, p, C[i,1])
fin  fin;
```

2°) Neliac

```
{ tableau A [100, 100] ,B[100, 100] , C [100, 100] , réel S, entier m,n,p ;
        Lire { > m,n,p < } ,
        pour i=1 pas 1 jusqu'à m faire
        { pour j=1 pas 1 jusqu'à n faire { Lire {>A [ i,j ]< } } } ,
        pour i=1 pas 1 jusqu'à n faire
        { pour j=1 pas 1 jusqu'à p faire {Lire {>B[i,j ]< } } } ,
        pour i=1 pas 1 jusqu'à m faire
        { pour j=1 pas 1 jusqu'à n faire
          { 0 → S, pour k=1 pas 1 jusqu'à p faire
            { S + A [i,k]×B[k,j]→S }, S → C[ i,j ] } } ,
        pour i=1 pas 1 jusqu'à m faire
        { pour j=1 pas 1 jusqu'à p faire
            { Ecrire { > C [ i,j ] < } } } } ;
```

3°) Fortran.

```
      DIMENSIØN  A (100,100), B (100,100), C(100,100)
      READ (5,1) M,N,P
1     FØRMAT (3I5)
      READ (5,2) ((A(I,J),J= 1,N),I=1,M)
2     FØRMAT (6E12.5)
      READ (5,2) ((B(I,J),J=1,P),I=1,N)
      DØ 3 I=1,M
      DØ 3 J=1,N
      S = 0.
      DØ 4 K = 1,P
4     S = S+A(I,K)* B(K,J)
3     C (I,J)=S
      WRITE (6,5) ((C(I,J),J=1,P),I=1,M)
      STØP
5     FØRMAT (10(1XE12.5) )
      END
```

4°) Jovial.

```
BEGIN ARRAY AA 100 100 F R ⌀ ARRAY BB 100 100 F R ⌀ ITEM MM 10 U ⌀
    ARRAY CC 100 100 F R ⌀ ITEM SS F R ⌀ ITEM NN 10 U ⌀
    ITEM PP 10 U ⌀
    READ (MM,NN, PP) ⌀
    FØR I = 0,1,MM ⌀
        BEGIN FØR J=0,1,NM ⌀  READ (AA(⌀ I, J ⌀) ) ⌀ END
    FØR I= 0, 1, NN ⌀
        BEGIN FØR J=0, 1, PP ⌀ READ ( (BB (⌀ I, J ⌀) ) ⌀ END
    FØR I = 0, 1, MM ⌀
        BEGIN FØR J= 0, 1,NN⌀
            BEGIN SS = 0 ⌀ FØR  K= 0, 1, PP ⌀
                SS = SS + AA (⌀I,K⌀)* BB (⌀K,J⌀) ⌀
                CC (⌀ I,J⌀) = SS ⌀ END END
    FØR I = 0,1,MM⌀
        BEGIN FØR J=0, 1,PP ⌀ WRITE (CC(⌀ I,J ⌀ ) )⌀ END END
```

5°) PL/I

```
PRØGRAMME : BEGIN ; READ DATA (M, N, P)  ;
        BEGIN; DECLARE (A (M,N), B (N,P), C(M,P) REAL;
              READ (A(*,*), B (*,*) ) ( E (12,5) );
EXTERNE : DØ I = 1 TØ M; DØ J =1 TØ N; S = 0;
INTERNE : DØ K = 1 TØ P; S = S + A (I,K)* B (K,J); END INTERNE;
        C (I,J) = S; END EXTERNE;
        WRITE (C(*,*) ) (E(13,5) ); END PRØGRAMME;
```

3 - Programme de traitement de fichiers.

Ce programme se contente de lire une série d'enregistrements incidquant le temps de travail d'employés et de calculer leur moyenne horaire.

1°) Cobol.

```
IDENTIFICATIØN DIVISIØN.
PRØGRAM-ID. MØYENNES.
ENVIRØNMENT DIVISION.
CØNFIGURATIØN SECTIØN.
SØURCE-CØMPUTER. ...
ØBJECT-CØMPUTER. ...
INPUT-ØUTPUT SECTIØN.
FILE-CØNTRØL. ...
DATA DIVISIØN.
FILE SECTIØN.
FD    FICHIER, LABEL RECØRDS ØMITTED, DATA RECØRD PØINTAGE.
01    PØINTAGE
      02 NØM PICTURE X(15).
      02 DPT PICTURE X(5).
      02 TEMPS PICTURE 99V99 ØCCURS 5 TIMES.
      02 TEMPS-TØTAL PICTURE 99V99.
      02 MØYENNE PICTURE 99V99.
      02 FILLER PICTURE X(32).
FD    FICHIER-MAJ, LABEL RECØRDS ØMITTED, DATA RECØRD PØINTAGE-MAJ.
01    PØINTAGE-MAJ PICTURE X(80).
WØRKING-STØRAGE SECTIØN.
77    N SIZE 1 NUMERIC CØMPUTATIØNAL.
PRØCEDURE DIVISIØN.
1. ØPEN INPUT FICHIER, ØUTPUT FICHIER-MAJ.
2. READ FICHIER, AT END GØ  TØ  3.
   MØVE ZERØS TØ TEMPS-TØTAL, PERFØRM 4 VARYING N FRØM 1 BY 1
   UNTIL N GEATER 5, DIVIDE 5 INTØ TEMPS-TØTAL GIVING MØYENNE
   RØUNDED, WRITE PØINTAGE-MAJ FRØM PØINTAGE, GØ  TØ  2.
3. CLØSE FICHIER AND FICHIER-MAJ,  STØP RUN.
4. ADD TEMPS (N) TØ TEMPS-TØTAL.
```

2°) Algol.

```
début réels nom, département, temps total, moyenne; tableau temps [1:5] ;
      entier n;
      lecture: ENTREE (5,modèle 1, 1, nom,1,département, 5, temps [1] );
      modèle 1: MØDELE ('(3A6, A5, 5F5.2)');
      temps total := 0; pour n:=1, 2, 3, 4, 5 faire temps total := temps
      total + temps [n] ; moyenne := temps total /5.0;
      SØRTIE (6, modèle 2, 1, nom, 1, département, 5, temps [1], 1, temps
      total, 1, moyenne); allerà lecture;
      modèle 2: MØDELE ('(5X3A6, 5XA5, 7 (4XF5.2) )' )
fin;
```

3°) Fortran.

```
      DIMENSION TEMPS(5)
      REAL NOM, MOYEN
1     READ (5,100) NOM, DPT, TEMPS
      TEMTOT=0.
      DO 2 N=1,5
2     TEMTOT= TEMTOT + TEMPS(N)
      MOYEN=TEMTOT/5.
      WRITE (6,101) NOM, DPT, TEMPS, TEMTOT, MOYEN
      GO TO 1
100   FORMAT (3A6,A5, 5F5.2)
101   FORMAT (5X3A6, 5XA5,7(4XF5.2))
      END
```

4°) Jovial.

```
FILE FICHIER H 10000 R 55 V(FINI) TAPE'A :
FILE FICHIER'MAJ H 10000 R 80 TAPE'B :
ITEM POINTAGE H 55 : ARRAY TEMPS 5 H 5 :
DEFINE NOM ''BYTE(:0,14:)(POINTAGE)'' :
DEFINE DPT ''BYTE(:15,19:)(POINTAGE)'' :
DEFINE T'TEMPS ''BYTE(:20,44:)(POINTAGE)'' :
DEFINE TEMPS'TOTAL ''BYTE(:45,49:)(POINTAGE)'' :
DEFINE MOYENNE ''BYTE(:50,54:)(POINTAGE)'' :
OVERLAY TEMPS = T'TEMPS :
      OPEN INPUT FICHIER : OPEN OUTPUT FICHIER'MAJ :
LECTURE.   INPUT FICHIER POINTAGE :
      IF FICHIER EQ V(FINI) : GOTO C'EST'TOUT :
      TEMPS'TOTAL = TEMPS'TOTAL + TEMPS(:N:) :
      MOYENNE = TEMPS'TOTAL/5 : OUTPUT FICHIER'MAJ POINTAGE :
      GOTO LECTURE :
C'EST'TOUT.   CLOSE FICHIER : CLOSE FICHIER'MAJ : STOP :
```

5°) PL/I

```
BEGIN ; DECLARE 1 POINTAGE, 2 NOM CHAR (15), 2 DPT CHAR (5) ,
      2 TEMPS (5) PIC (99.99) , 2 TEMPS_TOTAL PIC (99.99) ,
      2 MOYENNE PIC (99.99) ;
        DECLARE FICHIER FILE INPUT BLOCK (FIXED, 55, 10) , FICHIER_MAJ
        OUTPUT FILS ;
        OPEN FICHIER INPUT, FICHIER_MAJ OUTPUT ;
        LECTURE : READ FILE (FICHIER),(POINTAGE);ON ENDFILE GOTO FINI ;
              TEMPS_TOTAL = 0 ; Do N = 1 TO 5 ; TEMPS_TOTAL = TEMPS_TOTAL +
              TEMPS (N) ; END ; MOYENNE = TEMPS_TOTAL / 5 ;
              WRITE FILE (FICHIER = MAJ) , (POINTAGE) ; GOTO LECTURE ;
        FINI : CLOSE FICHIER , FICHIER_MAJ ; STOP ; END ;
```

4. Programme de traitement de chaînes

Il s'agit ici d'un programme lisant une formule arithmétique écrite sous forme complètement parenthèsée , et la réécrivant en notation polonaise post-fixée.

1°) Comit.

```
TEXTE *.=0     *
LIRE : // *RAK 1, *Q1 1    TEXTE
* : // *A1 1     *
COMP -=0     COMP
BALAI :1 // *L1     RECHERCHE
* :1 // *Q1 1     BALAI
FIN :=*.-*.+1+*.-*. // *A1 2, *WAM1 2 3     BOUT
-RECHERCHE *(=0     BALAI
       *+ // *S2 1     BALAI
       *- // *S2 1     BALAI
       ** // *S2 1     BALAI
       */ // *S2 1     BALAI
       *) // *N2 1,*Q1 1     BALAI
* :=-ERREUR-PARENTHESE // *WAM1     FIN
BOUT *
       END
```

2°) Snobol.

```
DEPART SYS .READ *TEXTE* ''::''
BALAI TEXTE *ELEMENT/''1''*   /F(FINI)
      ELEMENT ''('' =       /S(BALAI)
      ELEMENT ''+''         /S(BALAI)
      ELEMENT ''-''         /S(BALAI)
      ELEMENT ''*''         /S(BALAI)
      ELEMENT ''/''         /S(BALAI)
      ELEMENT '')''         /S(DEPILER)
RANGER POLONAISE = POLONAISE ELEMENT /(BALAI)
DEPILER PILE *ELEMENT/''1''*          /S(RANGER)
ERREUR SYS .PRINT ''ERREUR PARENTHESE''    /(FINI)
EMPILER PILE = ELEMENT PILE    /(BALAI)
FINI   SYS   .PRINT   POLONAISE
END   DEPART
```

3°) Lisp.

```
DEFINE (((POLONAISE (LAMBDA () (PROG () ((CLEARBUFF ()) (STARTREAD ())
(ADVANCE ()) (TERME ()) (ENDREAD ()) (TERPRI ()) (RETURN ()))))))
(TERME (LAMBDA () (PROG (X) ((COND ((NOT (EQ CURCHAR LPAR)) (ERREUR ())))
(OPERANDE ()) (ADVANCE ()) (COND ((NOT (OPCHAR CURCHAR)) (ERREUR ())))
(SETQ X CURCHAR) (OPERANDE ()) (ADVANCE ()) (COND ((NOT (EQ CURCHAR RPAR))
(ERREUR ())))) (PRINT1 X) (RETURN ()))))))
(ERREUR (LAMBDA () (COND ((EQ CURCHAR :EOR:) (PROG2 (ERREUR ()) (ADVANCE ())))
(T (ADVANCE ()))))))
(OPERANDE (LAMBDA () (PROG () ((ADVANCE ()) (COND ((EQ CURCHAR LPAR)
(PROG2 (TERME ()) (GO FIN)))) A (COND ((OR (DIGIT CURCHAR)
(LITER CURCHAR)) (PROG2 (PROG2 ((PACK CURCHAR) (ADVANCE ())) (GO A)))))
(PRINT1 (INTERN (MKNAM ())))) FIN (RETURN ()))))))) STOP
```

4°) IPL

```
P1      10L3
        40W24
        20W24
          J154
          J180
        30W24
9-1       L1
          60W0
        10(          9-4
          J2         9-2
        70      9-1
        11W0
        10+          9-3
          J2
        70      9-2  P2
        11W0         E1
        10-
          J2
        70      9-2  9-2
        11W0
        10
          J2
        70      9-2  9-1
        11W0
        10/          E2
          J2         R1
        70      9-2
        11W0
        10)
          J2
        70           9-3
        11W0
        10*
          J2
        70           0
        11W0
          E1         9-1
        11W0
        40P2
        20P2         9-1
        11P2
        30P2         9-4
          0          0
        10E2
        10E3
          R1
          J156
        70           9-1
          J155
          J154       9-2
        30W24
        30W25        0
        +010
        40W25
        20W25
        40W24
        20W24        0
L1      10I2
        10L3
          R1
9-2     10W25
          J125
        10I4
          J115
        709-1
          J154
          J180
9-1       J186
        709-2
        30W24
        30W25
I4      +0180
I2      +010
```

5°) Algol.

```
début tableaux carte [1:80], Ligne [1:120] ; réels PARDRØITE, PARGAUCHE,
      BLANC ; entiers colonne, trou;
      colonne :=1; trou :=0; ENTREE (5, tout, 80, carte [1]);
spécial : tout: MØDELE ('(80A1)');
      début procédure Ecrire caractère (x); réel x;
          début Ligne [colonne] := x; colonne := colonne +1;
              si colonne >120 alors début colonne :=1;
              SØRTIE (6,toute, 120, Ligne [1])fin fin ;
          réelle procédure Caractère lu ;
          début trou:= trou +1; si trou>80 alors
              début trou:=1; ENTREE (5, tout, 80, Carte [1])fin;
              Caractère lu:= Carte [trou] fin;
          procédure terme;
          début réel x; Opérande x:= Caractère lu; Opérande;
               Ecrire caractère (x) fin;
          procédure Opérande;
          début réel x; x:= Caractère lu; si x= PARGAUCHE alors
               début terme; allerà fin fin; Ecrire caractère (x);
               composition du mot:si x=BLANC alors allerà fin;x:=Caractère
           lu; Ecrire caractère (x); allerà composition du mot; fin: fin;
          ENTREE (5, spécial, 1, PARGAUCHE, 1, BLANC, 1, PARDRØITE);
          terme; pour trou := colonne pas 1 jusqu'à 120 faire Ligne[trou]:=
          BLANC; SØRTIE (6, toute, 120, Ligne [1]);
          toute: MØDELE ('(120A1)')
fin fin;
```

6°) Slip.

```
      COMMON /C/ CIGNE(120),COLON /D/ CARTE(80), TROU
      INTEGER COLON, TROU
      COLON =1
      READ (5=1) CARTE, PARGAU, BLANC, PARDRO
1     FORMAT (80A1)
      ASSIGN 1000 TO LILI
      CALL VISIT(LILI,PRESRV2(X,X))
      DO 2 TROU=COLON,120
2     CIGNE(TROU)=BLANC
      WRITE (6,3)CIGNE
3     FORMAT(120A1)
      STOP
1000  ASSIGN 10 000 TO JOJO
      CALL VISIT(JOJO,PARMT2(X,X))
      X=CARALU(BIDON)
      CALL VISIT (JOJO,PARMT2(X,X))
      CALL ECRIRE (X)
      CALL TERM(0,RESTOR(2))
10000 Y=CARALU(BIDON)
      IF (Y .NE. PARGAU) GO TO 4
      ASSIGN 1000 TO NANA
      CALL VISIT(NANA,PRESRV2(Y,Y))
5     CALL TERM(0(RESTOR(2))
4     CALL ECRIRE(Y)
      IF(Y .EQ. BLANC) GO TO 5
      Y= CARALU(BIDON)
      GO TO 5
      END
```

```
      SUBROUTINE ECRIRE (I)
      COMMON /C/ LIGNE(120), LOLON
      LIGNE(LOLON)=I
      LOLON=LOLON+1
      IF(LOLON .LE. 120) RETURN
      LOLON =1
      WRITE (6,1) LIGNE
1     FORMAT(120A1)
      RETURN
      END
      FUNCTION CARALU (BIDON)
      COMMON /D/CARTE(80),ITROU
      ITROU=ITROU+1
      IF(ITROU .LE. 80) GOTO 10
      ITROU=1
      READ(5,1) CARTE
1     FORMAT(80A1)
10    CARALU= CARTE(ITROU)
      RETURN
      END
```

5 - Programmes récursifs.

L'exemple choisi est celui de la fonction d'Ackermann, qui contient un cas de double récursivité, et semble presque impossible à programmer sans ce moyen, bien qu'elle reste très simple.

1°) Algol.

```
début entière procédure Ackermann (m, n);valeurs m, n; entiers m, n;
      Ackermann := si m=0 alors n+1 sinon sin= 0 alors Ackermann (m-1,1)
      sinon Ackermann (m-1, Ackermann (m,n-1) );
      ECRIRE (Ackermann (EDØNNEE, EDØNNEE) ) fin ;
```

2°) PL/I.

```
ESSAI: BEGIN; ACKERMANN: PRØCEDURE (M,N) RECURSIVE;
       BEGIN; IF M=0 THEN RETURN N+1;
              IF N=0 THEN DØ; J = ACKERMANN (M-1,1); RETURN J; END;
              J= ACKERMANN (M,N-1); K=ACKERMANN (M-1,J); RETURN K;
       END ACKERMANN; READ DATA (M,N); I=ACKERMANN (M,N);
       WRITE DATA (M, N,I); END ESSAI;
```

3°) Lisp.

```
DEFINE(((ACKERMANN (LAMBDA (M,N) (COND
      ((ZEROP M) (ADD1 N)) ((ZEROP N) (ACKERMANN (SUB1 M) 1))
      (T (ACKERMANN (SUB1 M) (ACKERMANN M (SUB1 N)))))))))
ACKERMANN (4,5) STOP
```

4°) Slip.

```
      INTEGER ACKER, VISIT, TOP
      COMMON MEMLIB, W(100)
      READ(5,100) MM,NN
      ASSIGN 1000 TO LIEU
      ACKER=VISIT(LIEU,PARMT2(MM,NN))
      WRITE (6,100) MM,NN, ACKER
      STOP
100   FORMAT(3I5)
1000  M= TOP(W(1))
      N=TOP(W(2))
2     IF (M .NE. 0) GO TO 1
      CALL TERM(N+1,RESTOR(2))
1     IF (N .NE. 0) GO TO 3
      M=M-1
      N=1
      GO TO 2
3     N=VISIT(LIEU,PARMT2(M,N-1))
      M=M-1
      GO TO 2
      END
```

5°) Comit.

```
ENTREE :=M/.4+N/.5+RETOUR/FINI // *S1 1,*S2 2,*S127 3 *
ACKER :=U+V //*N1 1,*N2 2 *
ACKER-BIS M/.GO ACKER-UN
* M+N=2/.I1   *
* N=A+RESULTAT/.*1 // *N127 1, *S3 2     :
ACKER-UN N/.GO  ACKER-DEUX
*    M+N=1/.D1+2/.1 ACKER-BIS
ACKER-DEUX M+N=1/.D1+1+2/.D1+RETOUR/ACKER-TROIS // *S1 1, *S1 2,  -
    *S2 3, *S127 4   ACKER
ACKER-TROIS : // *N3 1     *
*    RESULTAT= N/.*1 // *S2 1    ACKER
FINI  : // *A3 1, *WSM1    *
   END
```

6°) Snobol.

```
ENTREE ADRESSE = ''.FINI''
       SYS .READ *PARAM1*   ''/''
       SYS .READ *PARAM2*   ''/''
ACKER PARAM1  ''.'' *M* =
       PARAM2  ''.'' *N* =
ACKER-BIS M ''0'' =       /F(ACKER-UN)
       RESULTAT = N+ ''1''
       N =
       ADRESSE  ''.'' *RETOUR* =    /(:RETOUR)
ACKER-UN  N ''0'' = ''1''     OF(ACKER-DEUX)
       M = M -  ''1''     /(ACKER-BIS)
ACKER-DEUX P = M - ''1''
       PARAM1 = ''.'' M ''.'' P PARAM1
       PARAM2 = ''.'' N PARAM2
       ADRESSE = ''.ACKER-TROIS'' ADRESSE    /(ACKER)
ACKER-TROIS PARAM2 = ''.'' RESULTAT PARAM2    /(ACKER)
FINI   SYS  .PRINT ''RESULTAT    = ''    RESULTAT
END ENTREE
```

7°) IPL.

```
E0    10M0                          J111
      10N0                          10U0
        J51                         20W1        A0
        A0        J153      9-2       J41
A0    10W0                          10W1
        J117                        40H0
      709-1                           J111
      10W1                            A0
        J90                         20W1
        J110      J31               10W0
9-1   10W1                          40H0
        J117                            J111    A0
      709-2                 M0      +01         4
      10W0                  N0      +01         5
      40H0                  U0      +01         1
```

```
      DIMENSION A(30,30),B(30,30),X(30,30)
      READ(5,100) M,N,((A(I,J),J=1,M),I=1,M),((B(I,J),J=1,N),I=1,M)
100   FORMAT(2I3,(6F10.5))
      CALL  RGSL (A,B,X,M,N)
      WRITE(6,101)N,M,((A(I,J),J=1,M),I=1,M)
      WRITE(6,102)((B(I,J),J=1,N),I=1,M)
      WRITE(6,103)((X(I,J),J=1,N),I=1,M)
      STOP
101   FORMAT(24H1RESOLUTION COMPLETE DE I2,22H SYSTEMES LINEAIRES A
     1I2,10HEQUATIONS./31HOMATRICE DES PREMIERS MEMBRES.. (9(2XE12.5)))
102   FORMAT(/31HOMATRICE DES SECONDS MEMBRES.. (9(2XE12.5)))
103   FORMAT(/33HOMATRICE DES VECTEURS RESULTATS.. (9(2XE12.5)))
      SUBROUTINE RGSL (A,B,X,M,N)
      DIMENSION A(30,30),B(30,30),X(30,30)
      MM=M-1
      DO 1 K=1,M
      IF(A(K,K) .NE. 0.) GO TO 2
      KK=K+1
      DO 3 NM=KK,M
      IF (A(NM,KK) .EQ. 0.) GO TO 3
      L=NM
      GO TO 4
3     CONTINUE
4     DO 6 NM=K,M
      R=A(K,NM)
      A(K,NM)=A(L,NM)
6     A(L,NM)=R
      DO 7 NM=1,N
      R=B(K,NM)
      B(K,NM)=B(L,NM)
7     B(L,NM)=R
2     DO  8 I=KK,M
      S=A(I,K)/A(K,K)
      DO 9 J=KK,M
9     A(I,J)=A(I,J)-S*A(K,J)
      DO 10 J=1,N
10    B(I,J)=B(I,J)-S*B(K,J)
8     CONTINUE
1     CONTINUE
      DO 11 K=1,N
      DO 11 I=1,M
      II=M-I+1
      S=0.
      III=II+1
      DO 12 J=III,M
      JJ=M-J+III
12    S=S-A(II,JJ)*X(JJ,K)
11    X(II,K)=(B(II,K)+S)/A(II,II)
      RETURN
      END
```

2°) Algol.

Je donnerai ici deux exemples: le premier est un exemple de programme Algol normal, qui calcule une intégrale double par appel récursif de la procédure d'intégration Romberg. Le second est un programme Algol anormal, qui utilise au maximum les possibilités les plus bizarres et les plus inutiles du langage, et constitue donc un exemple de ce qu'il ne faut pas faire. Il calcule le produit de deux matrices par appel à multiples récursivités et effets de bord de la procédure GPS (General Problem Solver).

Premier programme:

```
début réelle procédure Romberg (fct, liminf, limsup, ordre);
      valeurs liminf, limsup, ordre; réels liminf, limsup; entier ordre;
      réelle procédure fct;
      début tableau t[ 1 :ordre +1 ] ;réels l,n,m; entiers f,h,j,n;
            l:= limsup-limif; t[ 1]:= (fct (limsup)+fct (liminf) )/2;
            n:=1;
            pour h:= 1 pas 1 jusqu'à ordre faire
            début u:=0; m:= l/(n+n);
                  pour j:=1 pas 2 jusqu'à n+n-1 faire
                  u:=u+fct (limsup + j×m);
                  t[h+1] := (u/n + t[h])/2; f:=1;
                  pour j:= h pas -1 jusqu'à 1 faire
                  début f:= 4×f;
                        t[j]:= t[j+1] + (t[j+1] -t[j] )/ f(-1)
                  fin ; n:= n+n  fin;
             Romberg := t[1]×l
      fin de la procédure Romberg;
      réelle procédure f (x); valeur x; réel x;
      début réelle procédure g (y); valeur y; réel y;
            g:= SIN (x×y); f:= Romberg (g,.0,1-x,4)
      fin de la procédure f;
      ECRIRE (Romberg (f,.0,1.0,4) )
fin du programme;
```

Deuxième programme :

```
début entiers m, n, p; LIRE (m,n,p);
      début tableaux A[1:m, 1:n ], B [1:n, 1:p] , C [1:m, 1:p ];
            entiers i, j, k;
            réelle procédure GPS (i, n, z, v); réels i, n, z, v;
            début pour i:=1 pas 1 jusqu'à n faire z:= v; GPS:=1 fin;
            ENTREE (5, tableau, m×n, A[ 1, 1] );
            ENTREE (5, tableau, n×p, B [1, 1 ]);
      tableau : MØDELE ('(6E12.5)');
            i:= GPS (i, 1.0, C [ 1,1 ], 0.0)×
                GPS (i, (m-1)×GPS (j, (p-1)×
                GPS (k,n, C [i,j], C [i,j] + A [i,k]×B [k, j] ),
                    C [i,j+1] , 0.0 ), C [ i+1,1 ] , 0.0);
            SØRTIE (6, tableau, m × p, C [ 1, 1 ]) fin fin ;
```

3°) Jovial

Il s'agit ici d'un programme de résolution par simple élimination d'un système d'équations linéaires très grand (jusqu'à l'ordre 1000) ; tout ce qui est en petits caractères est en fait du commentaire et ne figurerait pas dans le programme.

```
BEGIN SOLUTION 'D'UN' SYSTEME' LINEAIRE.
DEFINIE RANG "nombre d'équations et d'inconnues" $ DEFINE TAILLE
      " = (RANG + 1) * RANG " $   DEFINE MOT "longueur d'un mot" $
FILE COEFFICIENT Binaire TAILLE Rigide MOT V(PAS'PRET) V(PRET) V(OCCUPE)
      V(ERREUR) DRUM $
ARRAY SOLUTION RANG Floating Rounded $ ITEM MODE Floating Rounded $
ARRAY NORMALISE RANG Booléen $ ARRAY LIGNE RANG fixed 10 Unsigned $
PROCedure ENTREE'COEFFICIENT (LIGNE, COLONNE = ELEMENT , ERREUR.) $
ITEM LIGNE fixed 10 Unsigned $ ITEM COLONNE fixed  0 Unsigned $
        BEGIN POSition (COEFFICIENT) = (RANG+1) * LIGNE + COLONNE $
              INPUT COEFFICIENT ELEMENT $
        XX. IF COEFFICIENT EQ  V(OCCUPE) $ GOTO XX  $
            IF COEFFICIENT EQ  V(ERREUR) $ GOTO ERREUR $ END
PROCedure SORTIE'COEFFICIENT (LIGNE, COLONNE - , ELEMENT = ERREUR.) $
ITEM LIGNE fixed 10 Unsigned $ ITEM COLONNE fixed 10 Unsigned $
BEGIN POSition (COEFFICIENT) = (RANG + 1) * LIGNE + COLONNE $
              OUTPUT COEFFICIENT ELEMENT $
        XX. IF COEFFICIENT EQ V(OCCUPE) $ GOTO XX $
            IF COEFFICIENT EQ V(ERREUR) $ GOTO ERREUR $ END
        OPEN INPUT COEFFICIENT $
        FOR  R=0,1,RANG-1  $  NORMALISE ($ R $) = 0 $
        FOR  C=0,1,RANG-1  $
        BEGIN VALEUR'COLONNE'HAUTE = 0. $ FOR  R=0,1,RANG-1  $
              BEGIN IF NOT NORMALISE ($ R $)  $
                        BEGIN ENTREE'COEFFICIENT(R,C=ELEMENT,FAILLITE.)$
```

```
IF (/ELEMENT/) GR VALEUR'COLONNE'HAUTE  $
    BEGIN VALEUR'COLONNE'HAUTE = (/ELEMENT/) $
       LIGNE ($ C $) = R  $ END END END
IF VALEUR'COLONNE'HAUTE  EQ  0. $ GOTO FAILLITE. $
NORMALISE ($ LIGNE ($ C $) $) = 1 $ FOR I = C,1,RANG  $
    BEGIN ENTREE'COEFFICIENT (LIGNE ($ C $) , I = ELEMENT,FAILLITE.) $
       SORTIE'COEFFICIENT(LIGNE($ C $),I,ELEMENT/VALEUR'COLONNE'
       HAUTE = FAILLITE.) $ END
FOR R= 0,1,RANG-1  $
    BEGIN IF R NQ LIGNE ($ C $) $ BEGIN ENTREE'COEFFICIENT(R,C = ATEMP,
       FAILLITE.) $ FOR I=C,1,RANG  $
       BEGIN ENTREE'COEFFICIENT(LIGNE ($ C $),I = BTEMP, FAILLITE.) $
          ENTREE'COEFFICIENT(R,I = ELEMENT,FAILLITE.) $
          SORTIE'COEFFICIENT(R,I,ELEMENT-ATEMP * BTEMP = FAILLITE.) $
END END END END
FOR C=0,1,RANG-1  $ ENTREE'COEFFICIENT(LIGNE($ C $) , RANG =
    SOLUTION($ C $), FAILLITE.) $  SHUT OUTPUT COEFFICIENT $
    GOTO TOUT'S'EST'BIEN'PASSE  $
END "SOLUTION'D'UN'SYSTEME'LINEAIRE"
```

4°) Neliac.

Je donne ci-dessous un fragment du compilateur le plus élémentaire de Neliac, écrit en Neliac. Il s'agit de la partie qui traite les opérations d'entrée-sortie. Elle fait appel à des identificateurs et des procédures non définis ici. Générer Un En-Tête :

J → Adresse Voulue Saut, SAUT →[J] , J + 1 → J,

Première Adresse En-Tête →L, 0 →[L]→Compte En-Tête →Compte Espace En-Tête,

Lire En-Têtes : Trouver Prochain Symbole, si Symbole = Supérieur A :

Trouver Espacement Relatif,si non,; si Symbole = ZERØ : Lire

En-Têtes. si non, :

si Symbole = Signe Exposant: DIX + Compte Espaces En-Tête→Compte Espaces En-Tête, Retenir Symbole. si non,;

si Symbole = Signe Abosolu : 1+ Compte Espaces En-Tête →Compte Espaces En-Tête;

si non, 1 + Compte En-Têtes → Compte En-Têtes;

Retenir Symbole : [L]→Temporaire, si Temporaire < Presque Plein:

Temporaire x Taille Symbole + Symbole →[L];

si non, Temporaire x Taille Symbole + Symbole → [L],

L + 1 → L, 0 →[L]; Lire En-Têtes.

Trouver Espacement Relatif: si Compte En-Têtes + Compte Espaces

En-Tête >Longueur Ligne: 1 →Espacement; si non, (Longueur Ligne

- Compte En-Têtes) / Compte Espaces En-Têtes →Espacement;

Préparation A Convertir: PLEIN → Nouveau Diviseur, 0 → Nouveau Mot

L + 1 →Dernière Adresse En-Tête, Première Adresse En-Tête → L,

Obtenir Symboles En-Tête : Trouver Prochain Symbole En-Tête,

si Symbole = Signe Absolu : Ranger Nombreux Espaces, Test Mi-Chemin.

si non,; si Symbole = Signe Exposant : Ranger Espaces, Test Mi-Chemin.

si non,; Convertir Pour Flexo, si Index Case Courante = Index Case Haute: Ranger Symbole; si non, Ranger Changement Case;

Test Mi-Chemin: si L = Dernière Adresse En-Tête: Compiler En-Tête.

si non, Obtenir Symboles En-Tête.

Compiler En-Tête; Adresse Voulue Saut → M, [M] + J →[M], J+4 → CØDE [0] (0 →14) →CØDE [36] (0 →14), J+6 →CØDE [1] (0 →14) → CØDE [37] (0 →14), J+21 → CØDE [35] (0 →14), M+1→ CØDE [21] (0 →14), J+ 22 → CØDE [22] (0 →14) →CØDE [29] (0 →14), J+30 →CØDE [27] (0 →14), J+36 → CØDE [33] (0 →14), Compte Mots En-Tête-1 → CØDE [32] (0 →14), pour M = 0 (1) 35 {CØDE [M] →[J], J+1 → J, }, Restaurer NØN: Trouver Prochain Symbole, si Symbole = Crochet Droit: VIRGULE → Prochain Opérateur, Avancer. si non, Restaurer NØN. Ranger Espaces: { pour N = 1 (1) Espacement {Espace → Symbole, Ranger Symbole, }, }, Ranger Nombreux Espaces : {Espacement x 10 → Espacement, Ranger Espaces, Espacement /10 → Espacement, } ..

5°) PL/I.

Etant données les vocations multiples PL/I, il me semble intéressant de donner au moins deux exemples de programmes, l'un commercial, l'autre numérique. Il ne m'a pas été possible de trouver d'exemple de programmes utilisant les facilités "avancées" de PL/I, telles que les ordres au compilateur, les ordres SAVE ou WAIT. Le premier programme est un calcul de paie d'employés, le second résoud un système linéaire par la méthode de Crout, avec échanges.

Premier programme :

```
SALAIRES: PRØCEDURE MAIN ; DECLARE (1 PRINCIPAL SIZE (105) ,
        2(DEPT PIC (9(5)), NUM_EMP PIC (A(14)), SEXE PIC (A), NUM_ACT
        PIC (9(11)), TAUX_MENS PIC (9(4)V99) PØS (47),AVANCE PIC (999V99),
        TAUX_HØR PIC (9V99),CØDE PIC (A) PØS (104)),1 SUPPLEMENT SIZE(80),
        2 (NUM_EMP PIC (9(5)) PØS (6), TAUX_NØRM PIC (99V9) PØS (44),
        PRIME_HØR PIC (99V9),1 LISTE _CØNTRØLE SIZE (70),
        2 (DEPT PIC ((9(5)), NUM-EMP PIC (9(5)), NØM PIC (A(14)), SEXE
        PIC (A), NUM_ACT PIC (9(11)), TAUX_NØRM PIC (99V9) INITIAL (0),
        PRIME_HØR PIC (99V9) INITIAL (0), SUP_TØT PIC (999V99) INITIAL
        (0), SUP_FACT PIC (999) INITIAL (0), AVANCE PIC (999V99) INITIAL
        (0), PAIE_NETTE PIC (999V99) INITIAL (0)),1 ACCUM,
        2 (TAUX_NØRM PIC (99V9) INITIAL (0), PRIME_HØR PIC (999V9)
        INITIAL (0),SUP_TØT PIC (999V99) INIITAL (0), SUP_FACT PICT (999)
        INITIAL (0), CØNT_MAS PIC (9(5)) INITIAL (0), CØNT_CAR PIC (9(5))
        INITIAL (0)) CHAR, (SUP_MAX PIC (999V99),FACTA PIC (V9) INITIAL
        (5), FACTB PIC (V99) INITIAL (.85), FACTC PIC (9V9) INITIAL (1.5))
        FIXED, (FICH_M , FICH_S, FICH_C) FILE, (L1 (LIRCAR, CØMP1)INITIAL
        (LIRCAR), LR(ECRLIS, MØUVTEMP) INITIAL (ECRLIS)) LABEL, BLABLA
        LOGICAL;
    ØPEN FICH_M (INPUT); ØPEN FICH_S (INPUT);ØPEN FICH-C (ØUTPUT) ;
    LIRE_BANDE : READ FICH_M (PRINCIPAL) (A(105)),ØN ENDFILE (FICH_M)DØ ;
        CØNT_MAS  = '99999'; GØ TØ L1 ; END ;
    IF  PRINCIPAL $ SEXE EQ ' ' GØ TØ LIRE_BANDE ; IF PRINCIPAL$NUM_EMP LT
        CØNT_MAS PAUSE CØNT_MAS CAT ' ' CAT PRINCIPAL$NUM_EMP CAT
        'ERREUR SEQUENCE FICHIER PRINCIPAL' ; IF PRINCIPAL$NUM_EMP EQ
                                                        CØNT_MAS
        PAUSE PRINCIPAL$NUM_EMP CAT 'REPETITIØN FICHIER PRINCIPAL' ;
```

```
      CØNT_MAS = PRINCIPAL$NUM_EMP ; GØ TØ L1 ;
LIRCAR : READ FICH_S (SUPPLEMENT) (A(80)),ØN ENDFILE (FICH_S) DØ ;
      CØNT_CARTE = '99999' ; GØ TØ CØMP1 ; END ;
IF SUPPLEMENT$NUM_EMP LT CØNT_CARTE PAUSE SUPPLEMENT$NUM_EMP CAT
      'ERREUR SEQUENCE CARTES' ; IF SUPPLEMENT$NUM_EMP EQ CØNT_CARTE DØ;
      L2=MØUVTEMP ; ACCUM=ACCUM+LISTE_CØNTROLE, BY NAME ; END ;
      CØNT_CARTE = SUPPLEMENT$NUM_EMP ;
CØMP1 : L1 = LIRCAR ; IF CØNT_CARTE NE CØNT_MAS GØ TØ CØND_INEGAL ;
      IF CØNT_CARTE EQ '99999' GØ TØ CEST_FINI ;
CALCUL_SUP : LISTE_CØNTRØLE = SUPPLEMENT, BY NAME; LISTE_CONTRØLE $
                                                          SUP_TØT=
      SUPPLEMENT$TAUX_NØRM * TAUX_HØR + SUPPLEMENT$PRIME_HØR * (FACTC*
                                                          TAUX_HØR),
      RØUNDED ; LISTE_CONTRØLE$SUP_FACT=LISTE_CØNTRØLE$SUP_TØT * FACTB,
                                                          ROUNDED ;
      IF CØDE EQ '1' DØ ; PAIE_NETTE=PAIE_NETTE+LISTE_CØNTRØLE$SUP_FACT;
      GØ TØ LIRCAR ; END ; IF CØDE EQ '0' DØ ; SUP_MAX = (800-TAUX_MENS)
      * FACTA, RØUNDED ; IF SUP_MAX LE 0 GØ TØ  LIRCAR;IF LISTE_CØNTRØLE$
      SUP_TØT GT SUP_MAX DØ ; LISTE_CØNTRØLE$SUP-TØT=SUP_MAX;LISTE_CØN-
                                                          TRØLE $
      SUP_FACT=SUP_MAX * FACTB, ROUNDED ; END ; PAIE_NETTE=PAIE_NETTE+
      LISTE_CØNTRØLE$SUP_FACT ; END ; ELSE PAUSE PRINCIPAL$NUM_EMP CAT
      ' 'CAT CØDE CAT 'CØDE SUPPLEMENT. INCØNNU';GØ TØ LIRCAR ;
CØND_INEGAL : IF CØNT_MAS GT CØNT_CARTE DØ ; PAUSE SUPPLEMENT$NUM_EMP
      CAT 'PAS DE PRINCIPAL',CØMPLETE(BLABLA) ; GØ TØ LIRCAR ; END ;
      IF CØNT_MAS LT CØNT_CARTE DØ ; LISTE_CØNTRØLE=PRINCIPAL,BY NAME;
      PAIE_NETTE=PAIE_NETTE+PRINCIPAL$AVANCE;L1=CØMP1;END;GØ TØ L2 ;
MØUVTEMP : LISTE_CØNTRØLE = LISTE_CØNTRØLE + ACCUM, BY NAME;ACCUM=0,BY
      NAME ; L2 = ECRLIS ;
ECRLIS : WRITE FICH_C (LISTE_CØNTRØLE)(A(70));LISTE_CØNTRØLE$TAUX_NØRM=
      LISTE_CØNTRØLE$PRIME_HØR=PAIE_NETTE=LISTE_CØNTRØLE$SUP_TØT=0 ; GØ
      TØ LIRE BANDE ;
```

```
CEST_FINI: CLØSE FICH_M , SAVE ; CLØSE FICH_S ; CLØSE  FICH_C,SAVE  ;
     STØP ; END SALAIRES ;
```

Deuxième programme :

```
IP : PROCEDURE (M,N,U,V) ; DECLARE (U(*) , V(*), IPVAL)FLØAT;IPVAL=0 ;
     DØ I = M TØ N ; IPVAL=IPVAL+U(I)*V(I) ; RETURN (IPVAL);END IP ;
CRØUT : PRØCEDURE (N,A,B,SINGULIERE, Y, PIVØT, DETERMINANT) ;
     DECLARE (A(*,*), B(*),DET, TEMP, Y(*)) FLØAT,SINGULIERE LABEL,
     PIVØT (*)FIXED, R(N) DEFINED A(1SUB, 1SUB) ; INT = 1 ;
     DØ K = 1 TØ N ; TEMP = 0 ; DØ L = K TØ N ;
     A(L,K) = A(L,K) - IP(1,K-1),A(I,*), A(*,K) ; IF ABS (A(L,K)) GT
     TEMP DØ ; TEMP = ABS (A(L,K)) ; IMAX = L ; END ; END ;
     PIVØT (K) = IMAX ; IF IMAX NE K DØ ; INT = -INT ; DØ J = 1 TØ N ;
     TEMP = A(K,J) ; A(K,J) = A(IMAX,J) ; A(IMAX,J) = TEMP ; END ;
     TEMP = B(K) ; B(K) = B(IMAX) ; B(IMAX) = TEMP ; END ;
     IF A(K,K) EQ OEO DØ ; DETERMINANT = 0 ; RETURN TØ SINGULIERE ; END;
     DØ L = K+1 TØ N ; A(L,K) = A(L,K)/A(K,K) ; END ;
     DØ J = K+1 TØ N ; A(K,J) = A(K,J) - IP(1,K-1,A(K,*),A(*,J));END ;
     B(K) = B(K) - IP(1,K-1,A(K,*),B) ; END ;
     DETERMINANT = INT * PRØD(R) ; DØ K = N TØ 1 BY -1 ;
     Y(K) = (B(K) - IP(K+1,N,A(K,*),Y(*)))/A(K,K) ; END CRØUT ;
SØLUTIØN : PRØCEDURE (N,B,C,Z,PIVØT) ; DECLARE (B(*,*),C(*),TEMP)FLØAT,
     (N,PIVØT (*)) FIXED ; DØ K = 1 TØ N ; TEMP = C (PIVØT(K) ) ;
     C(PIVØT (K))=C(K) ; C(K) = TEMP ; C(K) = C(K)-IP(1,K-1,B(K,*),C(*))
     END ; DØ K = N TØ 1 BY -1 ; Z(K) = C(K)-IP(K+1,N,B(K,*),Z(*)))
     /B(K,K) ; END SØLUTION ;
```

6°) Cobol.

Pour des raisons de place, je ne puis donner ici en exemple qu'un programme qui du point de vue de Cobol est extrêmement simple. Il s'agit d'une mise à jour des comptes de clients d'une banque.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. BANQUE.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. ...
OBJECT-COMPUTER. ...
INPUT-OUTPUT SECTION.
      FILE-CONTROL. SELECT ENTREE, ASSIGN TO IN. SELECT MODIF,
         RENAMING ENTREE. SELECT SORTIE, ASSIGN TO OU. SELECT CONTR,
           RENAMING SORTIE.
DATA DIVISION.
FILE SECTION.
FD    ENTREE RECORD 80 LABEL RECORD OMITTED DATA RECORD CLIENTS.
01    CLIENTS.
        02 NUMERO PICTURE 9(6).
         02 CODE PICTURE AAA.
         02 NOM PICTURE A(61).
         02 COMPTE PICTURE S9(9).
FD    SORTIE RECORD 132 LABEL RECORD OMITTED DATA RECORD RESULTATS.
01    RESULTATS.
      02 NUMERO PICTURE Z(5)9.
      02 CODE PICTURE AAA.
      02 NOM PICTURE A(61).
      02 COMPTE PICTURE S9(8)9.
      02 FILLER PICTURE A(5).
      02 BARATIN PICTURE A(35).
      02 FILLER X(12).
WORKING-STORAGE SECTION.
      77 COMPTEUR PICTURE 9(4) USAGE COMPUTATIONAL.
      01 BIDON.
         02 CARTE OCCURS 100 TIMES.
            03 NUMERO PICTURE 9(6).
            03 CODE PICTURE AAA.
            03 NOM PICTURE A(61).
            03 COMPTE PICTURE S9(9).
PROCEDURE DIVISION.
1.    OPEN INPUT ENTREE, MODIF, OUTPUT SORTIE, CONTR. MOVE ZERO TO COMPTEUR
2.    PERFORM LECMO UNTIL CODE OF CLIENTS = SPACE. MOVE 1 TO COMPTEUR.
3.    READ ENTREE, AT END GO TO 4.
5.    IF NUMERO OF CLIENTS LESS THAN NUMERO OF CARTE (COMPTEUR ) THEN
      MOVE CORRESPONDING CLIENTS TO RESULTATS, WRITE RESULTATS, GO TO 3.
      IF CODE OF CARTE (COMPTEUR ) = 'MOD' THEN MOVE CORRESPONDING
      CARTE (COMPTEUR ) TO RESULTATS, ADD COMPTE OF CARTE (COMPTEUR ),
      COMPTE OF CLIENTS GIVING COMPTE OF RESULTATS, GO TO 5.
      IF CODE OF CARTE (COMPTEUR ) = 'SUP' THEN ADD 1 TO COMPTEUR, GO TO 5.
      MOVE CORRESPONDING CARTE (COMPTEUR ) TO RESULTATS.
```

```
6.     ADD 1 TO COMPTEUR, WRITE RESULTATS, GO TO 5.
4.     MOVE 1 TO COMPTEUR. PERFORM ECRIMO UNTIL CODE OF CARTE (COMPTEJR )
       = SPACE, CLOSE ENTREE, MODIF, SORTIE AND CONTR. STOP RUN.
LECMO.   READ MODIF. ADD 1 TO COMPTEUR. MOVE CORRESPONDING CLIENTS
       TO CARTE (COMPTEUR ).
TOTO. EXIT.
ECRIMO.  MOVE CARTE (COMPTEUR ) TO RESULTATS. IF CODE OF CARTE (COMPTEJR )
       = 'MOD' THEN MOVE 'MODIFICATION.' TO BARATIN, GO TO 7.
       IF CODE OF CARTE (COMPTEUR ) = 'SUP' THEN MOVE 'SUPPRESSION.'
       TO BARATIN, GO TO 7. MOVE 'NOUVEAU CLIENT.' TO BARATIN.
7.     WRITE RESULTATS, ADD 1 TO COMPTEUR.
LILI. EXIT.
```

7°) Apt.

Le programme suivant, qui est le seul exemple que je donnerai du langage Apt, est un programme de calcul d'une pièce simple formée de l'intersection d'une sphère et d'un sylindre avec trois plans perpendiculaires. Seule est calculée une des passes de l'outil, et non pas le total de celles qui sont nécessaires à l'exécution de la pièce.

```
PARTNO
      CLPRNT
      OUTTOL/.001          FF TOLERANCE D'APPROXIMATION DES COURBES.
      CUTTER/.5,.25    FF  DEFINITION DE L'OUTIL
ORIG=POINT/0,0,0    FF    DEFINITIONS GEOMETRIQUES CANONIQUES
PDEPAR=POINT/-8,-8,3
PLAN1=PLANE/0,0,1,.5
PLAN2=PLANE/0,1,0,0
PLAN3=PLANE/1,0,0,3
SPHER1,SPHERE/CENTER,ORIG,RADIUS,6
CYLIND=CYLNDR/CANON,1,0,0,0,0,1,5
      FROM/PDEPAR FF L'OUTIL PART DE PDEPAR,
      GO/TO,PLAN2,TO,PLAN1    FF  AVANCE SUR LE PLAN1 VERS PLAN2,
      TLRGT,GORGT/TO,SPHER1    FF  SE DEPLACE A DROITE VERS SPHER1,
      PSIS/SPHER1    FF     SE DEPLACE ENSUITE SUR LA SPHERE,
      GOUP/PLAN2    FF  LE LONG DE PLAN2, VERS LE HAUT,
      GORGT/CYLIND    FF  JUSQU'AU CYLINDRE, QU'IL SUIT
      GORGT/PLAN3,TO,PLAN1    FF    A DROITE JUSQU'AU PLAN3, PUIS AU PLAN1,
      GOTO/PDEPAR    FF    LE LONG DE LA SPHERE, JUSQU'A PDEPAR.
      END
      FINI
```

8°) IPL.

Etant donné qu'IPL est un langage machine, un programme même très simple est forcément long, et l'exemple que je donnerai ici ne peut constituer un programme complet. Il s'agit simplement d'un sous-programme destiné à être utilisé par ailleurs, qui lit des phrases quelconques et crée une liste des mots qu'elle contient.

```
9                          en-tête
2 B              100       réservations
2 M              100
2 N              100
2 R              100
2 X              100
3 X 1   01        80
5       01                 constantes
  M 1  -01         1
  N 1  +01         1
  N 5  +01         5
```

```
    N80    +01      80
    B1      21
5                   programme de lecture
    R95    10X1
             R96    mise en place
             J180   lecture
           70R97    sortie éventuelle
           11W25
             J184   recherche du premier
           30H0     caractère
           70R97    sortie éventuelle
```

-:-:-:-:-:-:-

```
          J90       création d'une
          J50       liste
9-10 10B1
          J90       Création d'un
          J121      mot local où
          J136      l'on range cinq
          J182      caractères lus
        709-200
        11W0
          J6
          J65       Rangement du
        11W25       mot en fin de
```

```
          40W21     Les termes
          40W25     constants caracté-
          40W30     risant la ligne lue
            J90     sont tous préservés
            J124    puis initialisés.
          20W21
            J90
            J124
          20W25
          10N5
            J120
          20W30
```

```
          10N80          liste
            J116         Sortie si la
          70      9-300  carte est vide.
          10M1           Achèvement du
            J161         mot avec suppres-
            J186         sion des blancs
          709-10         non significatifs
          50N1
            J161  9-10   Retour
9-200       J9           Effacement.
9-300     11W0           Liste terminée.
            J30
            R97    J4    Fin.
5                        Sous-programme de mise
  R96     40W24          en place.
          20W24
```

```
            J154  0      Fin
R97       11W21          Sous-programme de
          11W25          restauration.
          11W30
            J9           Effacement des
            J9           termes constants
            J9           créés lors de
          30W21          l'utilisation.
          30W24
          30W25
          30W30  0       Fin.
5           R95          Départ du program
                         me.
```

9°) Lisp.

Il s'agit ici du programme de vérification des théorèmes écrits dans le système de Wang sous forme de séquents, mais dans la notation fonctionnelle de Lisp.

```
DEFINE ((
(THEOREME (LAMBDA (S) (TH1 () () (CADR S) (CADDR S))))
(TH1 (LAMBDA (A1 A2 A C) (COND ((NULL A) (TH2 A1 A2 () () C))
      (T (OR (MEMBER (CAR A) C) (COND ((ATOM (CAR A)) (TH1 (COND
      ((MEMBER (CAR A) A1) A1)
      (T (CONS (CAR A) A1))) A2 (CDR A) C))
      (T(TH1 A1 (COND ((MEMBER (CAR A) A2) A2)
      (T (CONS (CAR A) A2))) (CDR A) C)))))))))
(TH2 (LAMBDA (A1 A2 C1 C2 C) (COND ((NULL C) (TH A1 A2 C1 C2))
      ((ATOM (CAR C)) (TH2 A1 A2 (COND ((MEMBER (CAR C) C1) C1)
      (T (CONS (CAR C) C1))) C2 (CDR C))) (T (TH2 A1 A2 C1 (COND
      ((MEMBER (CAR C) C2) C2) (T (CONS(CAR C) C2))) (CDR C))))))))
(TH (LAMBDA (A1 A2 C1 C2) (COND ((NULL A2) (AND (NOT (NULL C2))
      (THR (CAR C2) A1 A2 C1 (CDR C2)))) (T (THL (CAR A2) A1 (CDR A2)
      C1 C2)))))
(THL (LAMBDA (U A1 A2 C1 C2) (COND ((EQ (CAR U) (QUOTE NON)) (TH1R (CADR U)
      A1 A2 C1 C2)) ((EQ (CAR U) (QUOTE ET)) (TH2L (CDR U) A1 A2 C1 C2))
      ((EQ (CAR U) (QUOTE OU)) (AND (TH1L (CADR U) A1 A2 C1 C2)
      (TH1L (CADDR U) A1 A2 C1 C2))) ((EQ (CAR U) (QUOTE IMPLIQUE))
      (AND (TH1L (CADDR U) A1 A2 C1 C2) (TH1R (CADR U) A1 A2 C1 C2)))
      ((EQ (CAR U) (QUOTE EQUIV)) (AND (TH2L (CDR U) A1 A2 C1 C2)
      (TH2R (CDR U) A1 A2 C1 C2))) (T (ERROR (LIST (QUOTE THL)
      U A1 AB C1 C2)))   )))
(THR (LAMBDA (U A1 A2 C1 C2) (COND ((EQ (CAR U) (QUOTE NON)) (TH1L (CADR U)
      A1 A2 C1 C2)) ((EQ (CAR U) (QUOTE ET)) (AND (TH1R (CADR U) A1 A2 C1 C
      (TH1R (CADDR U) A1 A2 C1 C2))) ((EQ (CAR U) (QUOTE OU)) (TH2R (CDR U)
      A1 A2 C1 C2)) ((EQ (CAR U) (QUOTE IMPLIQUE)) (TH11 (CADR U) (CADDR U)
      A1 A2 C1 C2)) ((EQ (CAR U) (QUOTE EQUIV)) (AND (TH11 (CADR U)
      (CADDR U) A1 A2 C1 C2) (TH11 (CADDR U) (CADR U) A1 A2 C1 C2)))
      (T (ERROR (LIST (QUOTE THR) U A1 C1 C2)))   )))
(TH1L (LAMBDA (V A1 A2 C1 C2) (COND ((ATOM V) (OR (MEMBER V C1)
      (TH (CONS V A1) A2 C1 C2))) (T (OR (MEMBER V C2) (TH A1
     (CONS V A2) C1 C2)))   )))
(TH1R (LAMBDA (V A1 A2 C1 C2) (COND ((ATOM V) (OR (MEMBER V A1)
      (TH A1 A2 (CONS V C1) C2)))
      (T (OR (MEMBER V A2) (TH A1 A2 C1 (CONS V C2))))   )))
(TH2L (LAMBDA (V A1 A2 C1 C2) (COND ((ATOM (CAR V)) (OR (MEMBER (CAR V) C1)
      (TH1L (CADR V) (CONS (CAR V) A1) A2 C1 C2))) (T (OR (MEMBER
      (CAR V) C2) (TH1L (CADR V) A1 (CONS (CAR V) A2) C1 C2)))   )))
(TH2R (LAMBDA (V A1 A2 C1 C2) (COND ((ATOM (CAR V)) (OR (MEMBER (CAR V) A1)
      (TH1R (CADR V) A1 A2 (CONS (CAR V) C1 ) C2))) (T (OR (MEMBER
      (CAR V) A2) (TH1R (CADR V) A1 A2 C1 (CONS (CAR V) C2))))   )))
(TH11 (LAMBDA (V1 V2 A1 A2 C1 C2) (COND ((ATOM V1) (OR (MEMBER V1 C1)
      (TH1R V2 (CONS V1 A1) A2 C1 C2))) (T (OR (MEMBER V1 C2)
      (TH1R V2 A1 (CONS V1 A2) C1 C2)))   )))   ))
THEOREME ((FLECHE (P) ((OU P Q)))
THEOREME ((FLECHE ((OU A (NON B))) ((IMPLIQUE (ET P Q) (EQUIV P Q))))) STOP
```

10°) Slip.

Les deux sous-programmes ci-dessous servent, l'un à écrire, l'autre à lire, sur une unité extérieur quelconque, une structure de liste complète avec sa liste descriptive et ses sous-listes.

```
      FUNCTION ECRLIB (LISTE, IO)
      COMMON LISLIB, W(100)
      ASSIGN 1 TO LILI
      ECRLIB = VISIT(LILI,NEWTOP(LRDROV(LISTE),W(1)))
      RETURN
1     LR = INTGER(TOP(W(1)))
      NOM = LOFRDR (LR)
      X = CONT (LOCT(NOM)+1)
      JJ=2
      WRITE(IO)JJ,X
      IF (NAMEDL(NOM) .EQ. 0) GO TO 6
      CALL VISIT(LILI,NEWTOP(LRDROV(NAMEDL(NOM)),W(1)))
8     LR = INTGER(TOP(W(1)))
6     X=ADVLWR(LR,F)
      IF(F .NE. 0) GO TO 2
      IF (NAMTST(X) .EQ. 0) GO TO 5
      JJ=0
      WRITE(IO)JJ,X
      GO TO 6
5     JJ=1
      WRITE(IO)JJ,X
      CALL VISIT (LILI, NEWTOP(LRDROV(X),W(1)))
      GO TO 8
2     JJ=3
      KK=0
      WRITE(IO)JJ,KK
      LR=IRARDR(LR)
      CALL TERM(NOM,RESTOR(1))
      END
      FUNCTION LECLIB(LISTE,IO)
      COMMON LISLIB, W(100)
      ASSIGN 1 TO LILI
      LECLIB = INTGER(VISIT(LILI,NEWTOP(LISTE,W(1))))
      RETURN
1     READ(IO)NOM,DONNEE
      IF(NOM .NE.2) CALL ERROR
      CALL LIST(LISTE)
      IF(LNKL(DONNEE) .NE. 0) CALL SETIND(-1,VISIT(LILI,
     X   NEWTOP(LISTE,W(1))),-1,LOCT(LISTE)+1)
2     READ(IO)NOM,DONNEE
      IF (NOM .EQ. 3) CALL TERM (LISTE,RESTOR(1))
      IF(NOM .EQ. 1) CALL STRIND(VISIT(LILI,NEWTOP(LISTE,W(1))),
     X      LOCT(NEWBOT(DONNEE,LISTE))+1)
      IF (NOM .EQ. 0 .OR. NOM .EQ. 1) GO TO 2
      CALL ERROR
      END
```

11°) Comit.

Le programme suivant construit un jeu de cartes, le mélange, distribue les cartes aux joueurs et calcule le nombre de points, suivant les ordres données par des cartes lues.

```
A       *.=-                                        *
B       :=1+A // *Q1 1, *RAK 2                      A
*       :=*Q+A // *A1 2                             *
D       *Q+-=1                                      D
E       *Q+:+-=1+2+3 // *L2                         F
*       :=A+RETOUR/RETOUR // *N10 1, *S127 2     :
*       *Q+:+-=-CODE-DEPART-INCONNU*.+3+2+1//*WAM1 2 3   D
TERMINE  :=A+RETOUR/RETOUR // *N10 1, *S127 2 :
-F      CREER=0/CREER                               *
        DONNER=0/DONNER                             *
        MELANGER=0/MELANGER                         *
        TOT-PARTIEL=0/TOT-PAR                       *
        TOTAL=0/TOTAL                               *
        FIN=0/FIN // *Q10 1                         TERMINE
*       *Q+:+-=1+2 // *Q10 2                        D
CREER :=1+AS/.14+ROI/.13+DAME/.12+VALET/.11+CARTE/.10       -
            // DONNE NORD , *Q126 1                 *
CARTES  VALET+CARTE/.G2=1+2/.D1+2                   CARTES
INDICES   :1=1/C COEUR+1/C PIQUE+1/C CARREAU+1/C TREFLE //    -
            *Q5 1 2 3 4                             INDICES
*       :=A+1 // *N127 1, *A126 2                   :
DONNER  : // *Q126 1, *N5 1                        *
DONNE  NORD  :1  // *S1 1, DONNE SUD               DONNER
        SUD // *S2 1, DONNE EST                    DONNER
        EST // *S3 1, DONNE OUEST                  DONNER
        OUEST // *S4 1, DONNE NORD                 DONNER
*       :=A+1 // *N127 1, *A126 2                   :
MELANGER  : // *Q126 1, *A5 1                      *
*       :26+:26 // *Q6 1, *Q7 2                    BATTRE
*       :=-IL-MANQUE-DES-CARTES*.+1+RETOUR/MELANGER // *WSM1 2,     -
            *S127 3                                 CREER
BATTRE  DROITE : // *N6 1                          *
        GAUCHE  // *N7 1                           *
*       =1 // *Q5 1                                BATTRE
*       :=A+A // *A6 1, *A7 2, *Q5 1 2, *N127 1,*A126 2    :
TOT-PAR  :1+:=1+*D+1+TOTAL/.0+2 // *A*1 3      *
SOMME :1+:+*D+:1+TOTAL=1+2+3+4+5/.I.*4 // *N*1 4       SOMME
ZUT    *D+:1+TOTAL=A+3 // *N127 1                  :
TOTAL    :=NORD/.1+SUD/.2+EST/.3+OUEST/.4+1 // *Q126 5       *
*       :=1+RETOUR/1 // *S127 2                    TOT-PAR
1       :1+:1+:=NORD-+2+3+RETOUR/2 // *WAM1, *WSM2, *S127 4    TOT-PAR
2       :1+:1+:=SUD-+2+3+RETOUR/3 // *WAM1, *WSM2, *S127 4      TOT-PAR
3       :1+:1+:=EST-+2+3+RETOUR/4 // *WAM1, *WSM2, *S127 4     TOT-PAR
4       :1+:1+:=OUEST-+2 // *WAM1, *WSM2, *N127 1, *A126 I      :
FIN     :=-FIN-*.+1 // *WAM1 2                      *
        END
```

12°) Snobol.

Le programme suivant lit une suite de mots. Il réécrit cette liste rangée par ordre alphabétique, puis il raccourcit chaque mot jusqu'à une certaine taille minimum en enlevant des lettres en commençant par les plus fréquentes.

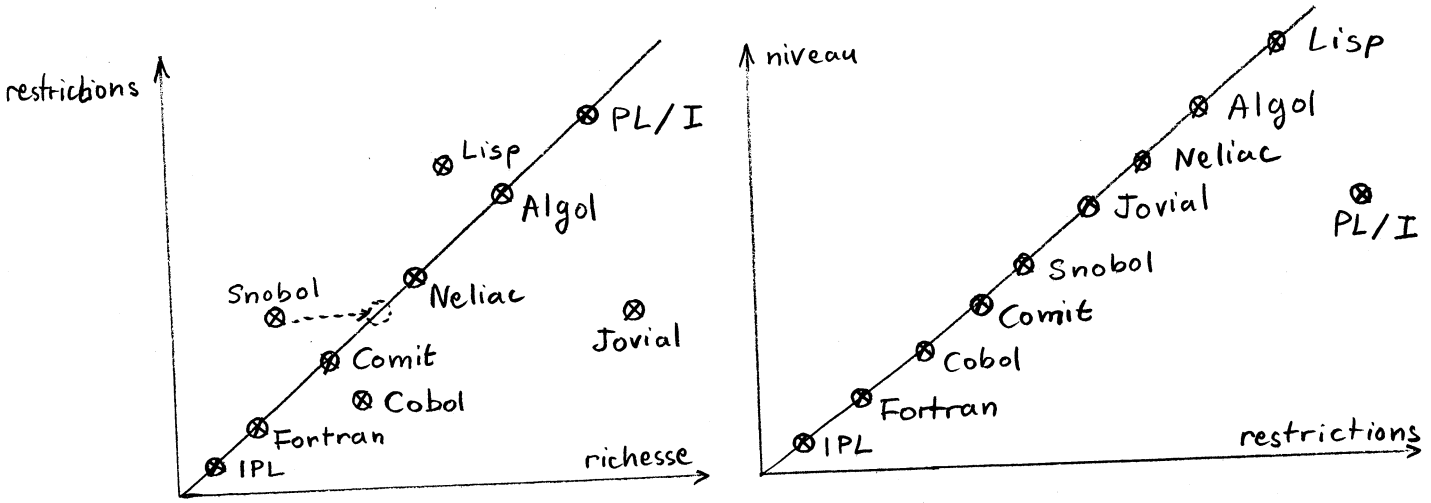
```
DEBUT SYS  .READ *TAILLE* ''/'' *MAXIMUM* ''/''
LECTURE   SYS  .READ *MOT* '' ''     /F(DEPART)
      LISTE = LISTE MOT   /(LECTURE)
DEPART  SYS  .PRINT ''LISTE A ORDONNER ET RACCOURCIR .. '' LISTE
      NATURE = ''INITIALE''
      RETOUR = ''RACCOURCIR''     /(ORDONNER)
RACCOURCIR  FREQUENCE = ''ESARTINULOCDPMBFGHQVJKXYWZ''
K0    FREQUENCE *LETTRE/''1''* =      /F(FINI)
      LISTE *MOT* '',''          /F(FINI)
K1    MOT *TETE/MAXIMUM*    /F(AUTROU)
      MOT LETTRE =                        /S(K1)
      COFFRE = COFFRE MOT '',''
K2    LISTE *MOT* '',''                   /S(K1)
      LISTE *COFFRE* =                    /(K0)
AUTROU TROU = TROU MOT '',''              /(K2)
FINI  LISTE = COFFRE TROU
      RETOUR = ''FINIFINI''
      NATURE = ''RACCOURCIE''
ORDONNER SYS  .PRINT VOID
      COFFRE =
      TROU =
L1    ALPHABET = ''ABCDEFGHIJKLMNOPQRSTUVWXYZ''
L2    TAILLE = TAILLE - ''1''
      TAILLE ''0''                        /S(FIN)
L3    LISTE *MOT* '','' =                 /F(L5)
      MOT *TETE/TAILLE* *TROU/''1''*      /F(L4)
      :TROU = :TROU MOT '',''             /(L3)
L4    COFFRE = COFFRE MOT '',''           /(L3)
L5    COFFRE *LISTE* =
L6    ALPHABET *TROU/''1''* =             /F(L1)
      LISTE = LISTE :TROU
      :TROU =                             /(L5)
FIN   SYS  .PRINT ''LISTE'' NATURE ''ORDONNEE ..  '' LISTE /(:RETOJR)
FINIFINI  SYS  .PRINT ''/''  ''  FIN DU PROGRAMME. ''
END   DEBUT
```

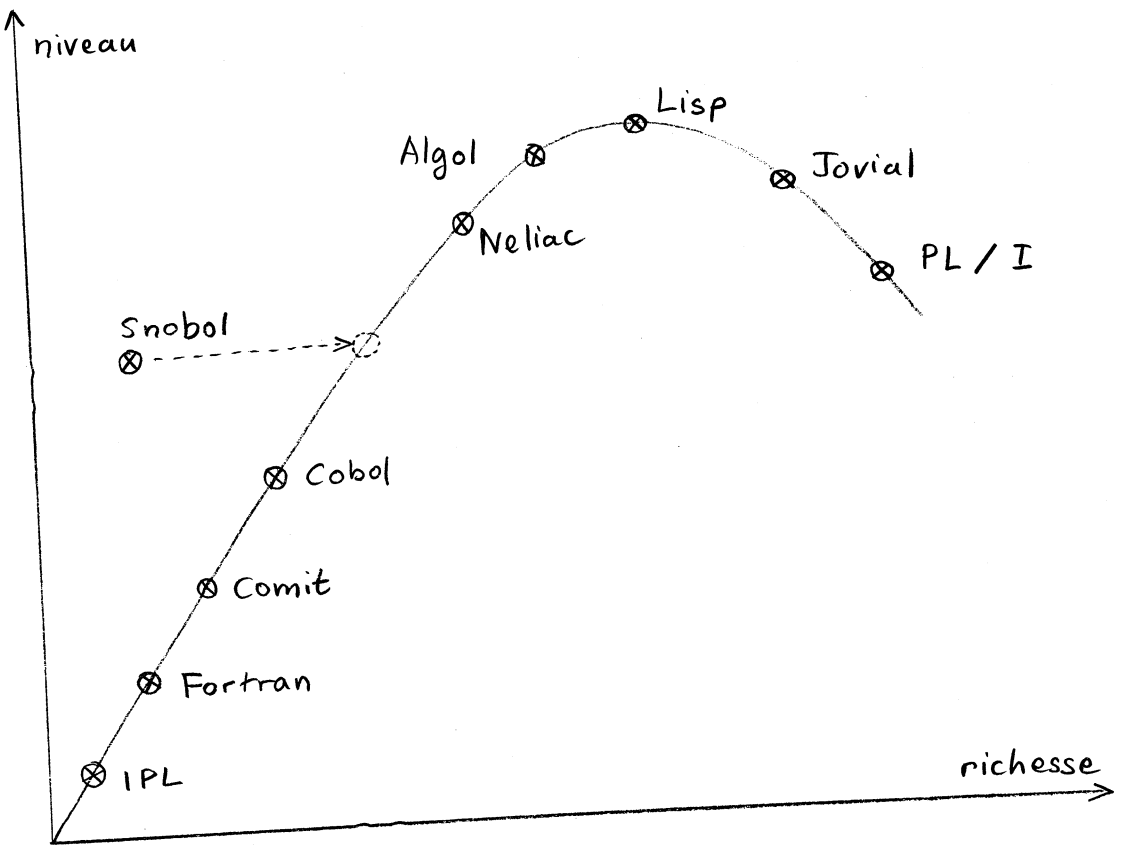
ANNEXE 1 -

	Fortran	Algol	Jovial	Neliac	PL/I	Cobol	Apt	IPL	Lisp	Slip	Comit	Snobol
Facilité d'écriture	⊗⊗	⊗⊗⊗	⊗	⊗⊗⊗	⊗	⊗⊗⊗	⊗⊗	⊗	-	⊗	⊗⊗	⊗⊗
Facilité des modifications	⊗⊗	⊗⊗	⊗⊗⊗	⊗⊗	⊗⊗	⊗⊗	⊗	⊗⊗	-	⊗	-	⊗⊗
Facilité de Lecture	⊗	⊗⊗⊗	-	⊗⊗	-	⊗⊗⊗	⊗	-	-	⊗	-	⊗⊗
Statut international	⊗⊗	⊗⊗⊗	⊗⊗	⊗	⊗⊗	⊗⊗⊗	⊗⊗	⊗	⊗	⊗	⊗	-
Efficacité du programme objet	⊗⊗⊗	⊗	⊗	⊗	⊗⊗	-	?	-	⊗⊗	⊗⊗	⊗	?
Facilité de regroupage de programmes	⊗⊗⊗	⊗⊗	⊗	-	⊗⊗	-	-	⊗⊗⊗	-	⊗⊗⊗	⊗⊗	⊗
Facilité de compilation	⊗⊗	⊗⊗	⊗	⊗⊗⊗	-	-	⊗	⊗⊗⊗	-	⊗⊗⊗	-	⊗
Définition claire et rigoureuse	⊗⊗	⊗⊗	⊗	-	⊗	⊗⊗	⊗	⊗⊗	⊗⊗⊗	⊗	⊗⊗⊗	⊗⊗
Possibilités d'extensions	⊗	⊗⊗	⊗	⊗⊗⊗	⊗⊗⊗	⊗	⊗⊗	⊗⊗⊗	⊗	⊗⊗⊗	-	⊗⊗
Allocation dynamique	-	⊗⊗	-	-	⊗⊗	-	-	⊗	⊗⊗	⊗⊗	⊗⊗⊗	⊗⊗
Fonctions standards	⊗	⊗	⊗⊗	-	⊗⊗⊗	-	-	⊗⊗	⊗⊗	⊗⊗	-	-
Traitements binaires	⊗⊗	⊗	⊗⊗⊗	⊗	⊗⊗	-	-	⊗	⊗	⊗⊗	-	-
Calcul logique	⊗⊗	⊗⊗	⊗⊗⊗	⊗	⊗	⊗	-	-	⊗	⊗	-	-
Traitement des chaînes	⊗	⊗	⊗	-	⊗⊗	⊗⊗	-	⊗	⊗⊗	⊗	⊗⊗⊗	⊗⊗⊗
Traitement des listes	-	-	-	-	⊗	-	-	⊗⊗⊗	⊗⊗⊗	⊗⊗⊗	⊗	⊗
Tables et structures	⊗	⊗	⊗⊗	-	⊗⊗	⊗⊗	-	⊗⊗	⊗	⊗	-	-
Arithmétique	⊗⊗	⊗⊗⊗	⊗⊗⊗	⊗⊗⊗	⊗⊗	⊗	⊗	⊗	⊗	⊗⊗	-	-
Fonctions et procédures	⊗⊗	⊗⊗⊗	⊗⊗⊗	⊗⊗	⊗⊗⊗	-	⊗	⊗⊗⊗	⊗⊗⊗	⊗⊗	-	-
Sous-programmes	⊗⊗	-	⊗⊗	-	⊗⊗⊗	⊗	-	⊗⊗⊗	⊗	⊗⊗	⊗	⊗
Compatibilité	⊗⊗⊗	⊗⊗	⊗	-	-	⊗	⊗	⊗⊗	⊗	⊗⊗	⊗⊗	-
Facilité d'apprentissage	⊗⊗⊗	⊗⊗	⊗	⊗⊗	-	⊗⊗	⊗	⊗⊗	⊗	⊗⊗	⊗⊗⊗	⊗⊗⊗
Utilité comme langage de communication	⊗	⊗⊗⊗	-	⊗	-	⊗⊗	⊗	-	⊗⊗	-	⊗	⊗

Graphiques de comparaison rapide :

On peut remarquer une anomalie dans chaque graphique: dans le premier, Jovial, dont les richesses constituent un ensemble mal bâti et trop peu général; dans le second, PL/I, qui est trop adapté à une machine précise en égard à son haut niveau; dans le troisième enfin, Snobol, qui une fois défini complètement retrouvera sans doute sa place normale. Apt et Slip sont trop à part pour figurer dans ces graphiques.





ANNEXE II

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

1. Algol

1.1 "Report on the algorithmic language Algol 60".
Peter Naur (Communications de l'A.C.M., Mai 1960).

1.2 "A syntax-directed compiler for Algol 60".
Irons (Communications de l'A.C.M. , Janvier 1961).

1.3 "A method of combining Algol and Cobol".
Jean Sammet (Sylvania Electronic Systems, 1961).

1.4 "Programmation Algol 60".
(I.M.A.G., 1961).

1.5 "A guide to Algol programming".
Mac Cracken (Wiley, 1962).

1.6 "Revised report on the algorithmic language Algol 60".
Peter Naur (Communications de l'A.C.M., Janvier 1963).

1.7 "Documentation problems : Algol 60".
Peter Naur (Communications de l'A.C.M., Mars 1963) .

1.8 "Algol : un nouveau langage scientifique. Guide pratique".
Bolliet, Gastinel, Laurent (Hermann, Avril 1964).

1.9 "Description en Algol d'un compilateur Algol simplifié".
Brasseur, Cohen (I.M.A.G., Avril 1964).

1.10 " A proposal for input - output conventions in Algol 60".
Subcomittee Algol (Communications de l'A.C.M., Mai 1964).

1.11 "Algol".
Woodger (I.E.E.E. Transactions, Août 1964).

Bibliographie complète jusqu'en Août 1963 dans [1.8].
Voir aussi [11.8], tome 1, [11.3], [11.14], [11.15], [11.16], [10.1], [3.8].

2. APT

2.1 "Automatic programming for numerically controlled tools".
Bates (Computer Applications, 1961).

2.2. "A description of the APT language".
Brown (Communications de l'A.C.M., Novembre 1963).

2.3. "Le système APT de programmation des machines-outils".
Mittman (Automatisme, Février 1965).

3. COBOL

3.1. "Univac special report on Cobol".
(Univac, 1961).

3.2. "A definition of the Cobol 61 procedure division using Algol 60 metalinguistics". Jean Sammet (Data Systems, Septembre 1961).

3.3. "Cobol Gamma 30".
(Bull, Mars 1962).

3.4. "Cobol : General description".
(Univac).

3.5. " Basic elements of Cobol 61"
Jean Sammet (Communications de l'A.C.M. , Mai 1962).

3.6. "Syntactical chart of Cobol 61".
Burroughs (Communications de l'A.C.M., Mai 1962).

3.7. "Cobol (Langage adapté aux problèmes commerciaux)".
(I B M, Février 1963).

3.8. "Etude critique et données de compilation du langage Cobol".
J.L. Baër (I.M.A.G., Juin 1963).

Voir aussi [11.8] tomes 1 et 3, [11.12], [1.3], [11.14], [11.16], [10.1], [11.3]

4. COMIT

4.1. "The Comit system for mechanichal translation".
Yngve (Information Processing, 1960).

4.2. "An introduction to Comit programming".
Yngve (M.I.T., Juin 1961).

4.3. "Comit programmers' reference manual".
Yngve (M.I.T., Septembre 1961).

4.4. "Comit : manuel de références pour programmeurs".
Traduction du précédent (I.M.A.G., 1961).

4.5. "Comit as an information retrieval language".
Yngve (Communications de l'A.C.M., Janvier 1962).

4.6. "Comit".
Yngve (Communications de l'A.C.M., Mars 1963).

4.7. "A Comit system for the 7040/44".
Yngve (M.I.T., Août 1963).

Voir aussi [11.8], tome 4, [12,11], [12,12]

5. FORTRAN

5.1. "The Fortran automatic coding system".
Backus, et al. (Proceedings of the Western Joint Computer Conference, 1957).

5.2. "Report on the algorithmic language Fortran II".
Rabinowitz (Communications de l'A.C.M., Juin 1960).

5.3. "A guide to Fortran Programming".
Mac Cracken (Wiley).

5.4. "Cours de Fortran II".
(E.D.F.)

5.5. "Fortran II".
(A.S.A., X 3.4.3. II, Mai 1963).

5.6. "Fortran IV".
(A.S.A., X 3.4.3. IV, Mai 1963).

5.7. "Fortran IV language".
(I B M, 1963).

5.8. "A Fortran Primer".
Elliot T. Organick (Addison-Wesley, 1963).

5.9. "Fortran : compatibility and standardization".
Heising (Datamation, Août 1964).

5.10. "The various Fortrans".
Oswald (Datamation, Août 1964).

5.11. "Fortran".
Backus, Heising (I.E.E.E. Transactions, Août 1964).

Voir aussi [11.8], tome 1, [11.14], [10.1], [10.2], [11.3].

6. I P L

6.1. "The elements of I P L programming".
Newell (Rand Corporation, 1960).

6.2. "I P L programmers' reference manual".
Newell (Rand Corporation, 1960).

6.3. "An introduction to Information Processing Language V".
Newell (Communications de l'A.C.M. , Avril 1960).

6.4. "Report on a general problem-solving program".
Newell (Information processing, 1960).
6.5. "Information Processing Language V manual".
Newell (Prentice, 1961)
6.6. "Documentation of IPL-V".
Newell (Communications de l'A.C.M., Mars 1963).
6.7. "The Baseball program: an automatic question-answerer".
Wolf, Chomsky, Green (Lincoln, Avril 1963).
6.8. "Information Processing Language V manual".
Newell (Rand Corporation, 1964).

Voir aussi [12.11] ,[12.6].

7 . Jovial

7.1. "The Jovial grammar and lexicon".
Shaw (System Development Corporation, Juin 1961).
7.2. "A programmer introduction to basic Jovial".
Shaw (System Development Corporation, Août 1961).
7.3. "The Jovial manual"
Shaw (System Developement Corporation, Décembre 1961).
7.4. "The Jovial Checker"
Wilkerson (Proceedings of Wester Joint Computer Conference,1961)
7.5. "The Jovial (J-2) language for the 7090 computer".
Schwartz (System Development Corporation, Janvier 1962).
7.6. "A specification of Jovial".
Shaw (Communications de l'A.C.M., Décembre 1963).
7.7. "On declaring arbitrarity coded alphabets".
Shaw (Communications de l'A.C.M., Mai 1964).

Voir aussi [11.8], tome 2, [11.13], [11.14], [10.1].

8. Lisp.

8.1. "Lisp 1 programmers' manual".
Mac Carthy (M.I.T., Mars 1960).

8.2. "Recursive fonctions of symbolic expressions and their computation by machine".
Mac Carthy (Communications de l'A.C.M., Avril 1960).

8.3. "Lisp 1.5. Programmers' manual".
Mac Carthy (M.I.T., 1962).

8.4. "Meteot:a Lisp interpreter for string transformations."
Bobrow (Maynard, 1964).

8.5. "The programming language Lisp: its operation and applications".
Berkeley (Maynard, 1964).

8.6. "The programming language Lisp: an introduction and appraisal".
Berkeley (Computers and automation, Septembre 1964).

Voir aussi [11.8], tome 4, [12.11], [12.9], [12.6], [12.12].

9. Neliac.

9.1."Neliac. A dialect of Algol".
Huskey (Communications de l'A.C.M., Août 1960).

9.2. "Compilation for two computers with Neliac".
Masterson (Communications de l'A.C.M., Novembre 1960).

9.3. "Machine-indepandant computer programming."
Halstead (Spartan, 1962).

9.4. "A syntactic description of BC Neliac".
Huskey (Communications de l'A.C.M., Juillet 1963).

Voir aussi [11.13], [11.14], [11.3].

10. PL/I.

10.1. "Acomparative evaluation of the New Programming Language".
Comba (IBM Confidential, Juillet 1964).

10.2. "The New Programming Language".
Mac Cracken (Datamation, Juillet 1964).

10.3. "The New Programming Language".
(IBM British Laboratories, Novembre 1964).

10.4. "NPL: Highlights of a new programming language".
Radin (Communications de l'A.C.M., Janvier 1965).

10.5. "PL/I : Language Specifications".
(IBM Operating System /360, Form C28-6571-0, Mai 1965).

11. Divers ; Langages d'analyse numérique.

11.1. "A first version of UNCOL".
Steel (Proceedings of Western Joint Computer Conference, 1961).

11.2. "Translation of retrieval requestes couched in a semiformal english-like language".
Cheatham (Communications de l'A.C.M., Janvier 1962).

11.3. "The language proliferation".
Shaw (Datamation, Mai 1962).

11.4. "Control and simulation language".
Buxton (Computer Journal, Octobre 1962).

11.5. "Fundamental principles of expressing a procedure for a computer application. The CLEO language."
Thompson (Computer Journal, Octobre 1962).

11.6. "The main features of CPL".
Barron (Computer Journal, Juillet 1963).

11.7. "Corc-The Cornell computing language".
Conway (Communications de l'A.C.M., Juin 1963).

11.8. "Etude prospective de la théorie des langages et des processus de transposition de l'un dans l'autre". (S.I.A., Octobre 1963).

11.9. "The external language KLIPA for the URAL-2 digital computer".
Greniewsky (Communications de l'A.C.M., Juin 1963).

11.10. "Atlas autocode programming manual".
Brooker (Ferranti, 1963).

11.11. "An experiment in a user-oriented computer system".
Klerer (Communications de l'A.C.M., Mai 1964).

11.12. "Fortran versus Cobol".
Fimple (Datamation, Août 1964).

11.13. "Procedure language for military command and control".
Steel (Symposium on computer programming for military systems, II.3, La Haye, Septembre 1964).

11.14. "Programming language selection for command and control applications".
Haverty (S.C.P.M.S., II.2).

11.15. "The problem of generating efficient object programs"
Huxtable (S.C.P.M.S., II.7).

11.16. "The requirements for a standardised real-time language".
Oakley (S.C.P.M.S., II-4.)

12. Divers : autres langages.

12.1. "A Fortran-compiled list-processing language".
Gelerntner (Journal de l'A.C.M., Avril 1960).

12.2. "Symbol manipulation by threaded lists ".
Perlis (Communications de l'A.C.M., Avril 1960).

12.3. "The use of the Fortran-compiled list-processing language".
Hausen (IBM, Juin 1960).

12.4. "Realisation of a geometry theorem proving machine".
Gelerntner (Information Processing, 1960).

12.5. "ALP: an autocode list-processing language".
Cooper (The Computer Journal, Avril 1962).

12.6. "Etude des langages de liste".
(S.I.A., Novembre 1963).

12.7. "Symmetric list processor".
Weizenbaum (Communications de l'A.C.M., Septembre 1963).

12.8. "A basis for a mathematical theory of computation".
Mac Carthy (in "Computer programming and formal systems ",
North Holland, 1963).

12.9. "An abstract computer with a Lisp-like machine language".
Gilmore (même référence que [12.8]).

12.10. "Snobol, a string manipulation language".
Farber (Journal de l'A.C.M., Janvier 1964).

12.11. "A comparison of list-processing computer languages".
Bobrow (Communications de l'A.C.M., Avril 1964).

12.12. "List processing and extension of language facility by embedding".
Bobrow (IEEE Transactions, Août 1964).

ANNEXE III.

Liste non exhaustive des compilateurs (arrêtée en Mai 1965)

1- Fortran.

I.B.M. 650.704.705.709.7010.7030.7040/44.7070/72/74.7080.7090/94.
1401.1410.1460.1620.

Remington = Univac : 111, 1107, 1050.

Control Data : 160.1604.3200.3400.3600.3800.6400.6600.

General Elctric : 205.215.235.400.415.425.435.500.

SDS : 900.910.920.930.9300.

Bull : Gamma 30. Gamma M40.

C A E : 510. 530. 130.

SETI : PB250. Pallas G. Pallas F.

Honeywell: 300. H800. H1880.

Philco 2000. Altac III. PB 440. RCA 301. DDP 224. PDP 7. DSI 1000.

Bendix G 20. AS 1210. ASI 420. RPC 4000. Cab 500.

2. Algol.

I.B.M. 1620. 7040/44. 7090/94.

Burroughs 205. 220. B5000.

Remington Univac 111 et 1107. CAE 510 et 530. SETI Pallas G.

General Electric 225. Control Data 3600. Bull Camma 30, Gamma 60,

Gamma M40. Cab 500. ICT 1500. NCR 503 et 803. RCA 601.

3. Jovial.

I.B.M. 7090/94. Control Data 1604.

4. Neliac.

I.B.M. : 1401. 704. 709. 7090/94

Remington Univac M-460 et M-490 - Burroughs 220. Philco CXPQ.

Control Data 160 A et 1604. Packard Bell 250.

5. Cobol.

I.B.M. : 704. 709. 7040/44. 7070/72/74. 7080. 7090/94. 1401. 1410. 1460.

Remington Univac : 11. 11[illegible].M-490.1050. 1107. 5580/90 II.

General Electric : 225. 400. 415. 425. 435.

Control Data : 1604 A. 3200. 3600. 3800.

ICT : 1300. 1301. 1302. 1500.
RCA 301. 501 et 601. Honeywell 400 et 800. NCR 304 et 315. Bull Gamma 30 Burroughs 5000.

6. Apt.
IBM 7090/94. Univac 1107. General Electric 625. Control Data 3600.

7. IPL.
IBM 1620, 7040/44, 7090/94. Control Data 1604 et G.20. Univac 1105 et 1107. Burroughs 220. Philco 2000.

8. Lisp.
I.B.M. 7040/44, 7090/94. Univac M-460. PDP 1.

9. Comit.
I.B.M. 709. 7040/44 et 7090/94.

10. Snobol.
I.B.M. 7090/94.

VU,

Grenoble, le

Le Président de la Thèse

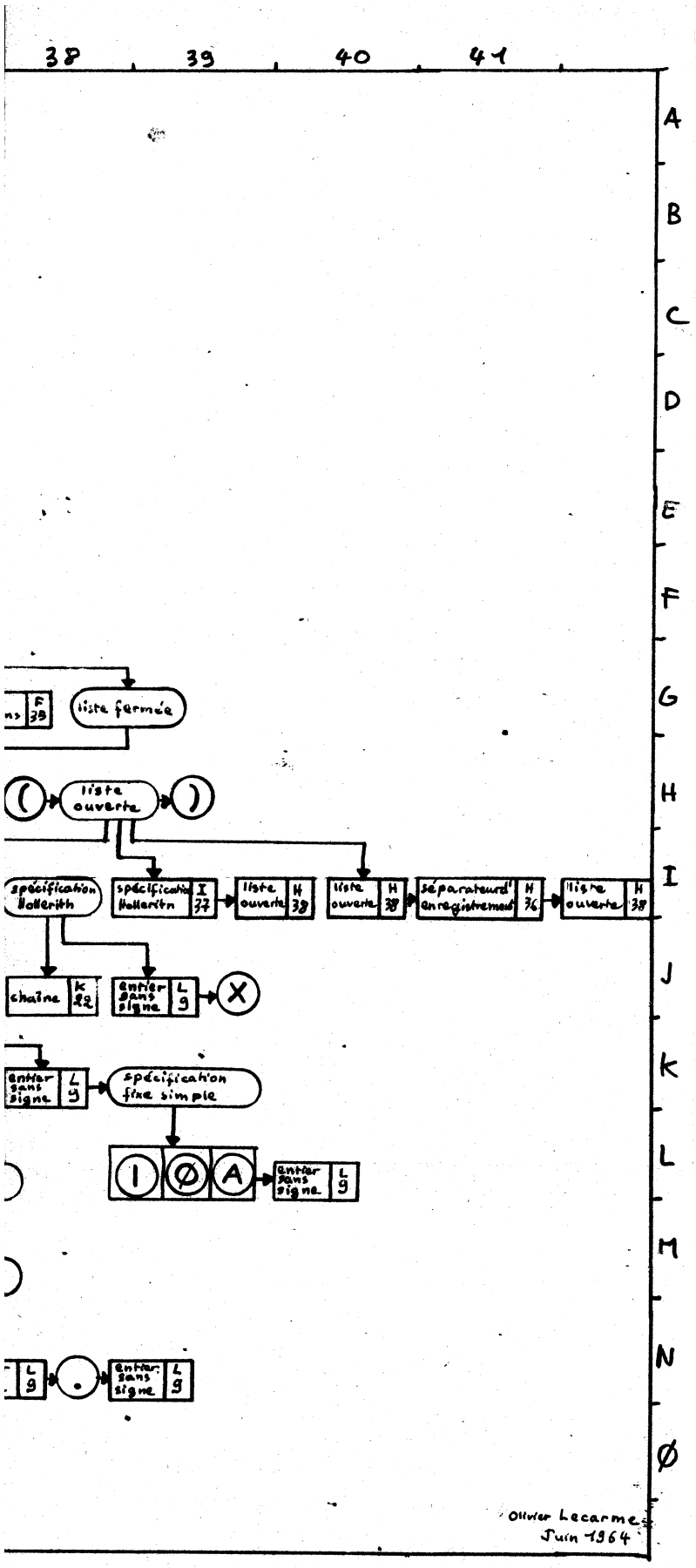
VU,

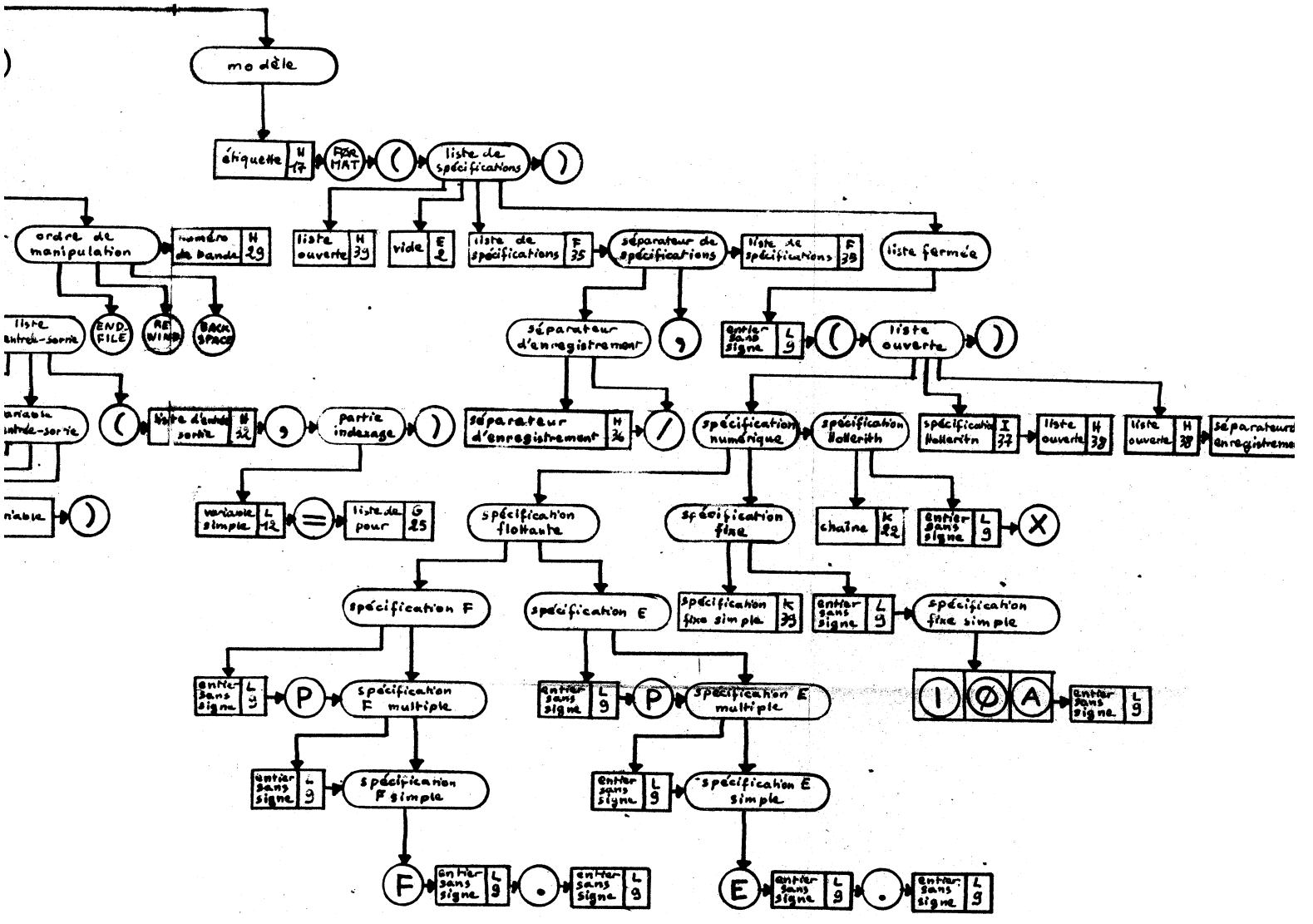
Grenoble, le

Le Doyen de la Faculté des Sciences

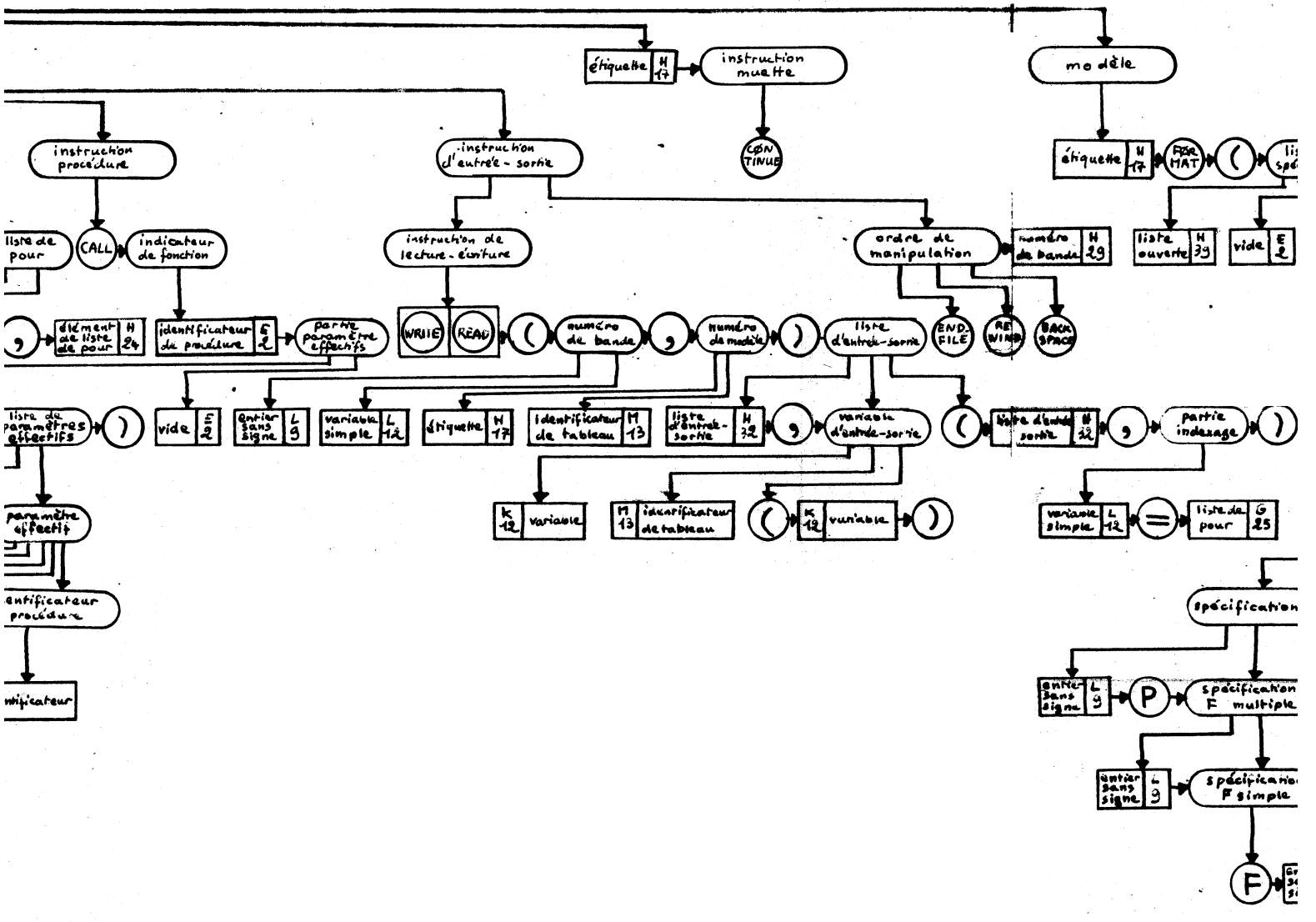
VU et permis d'imprimer,

Le Recteur de l'Académie de Greno

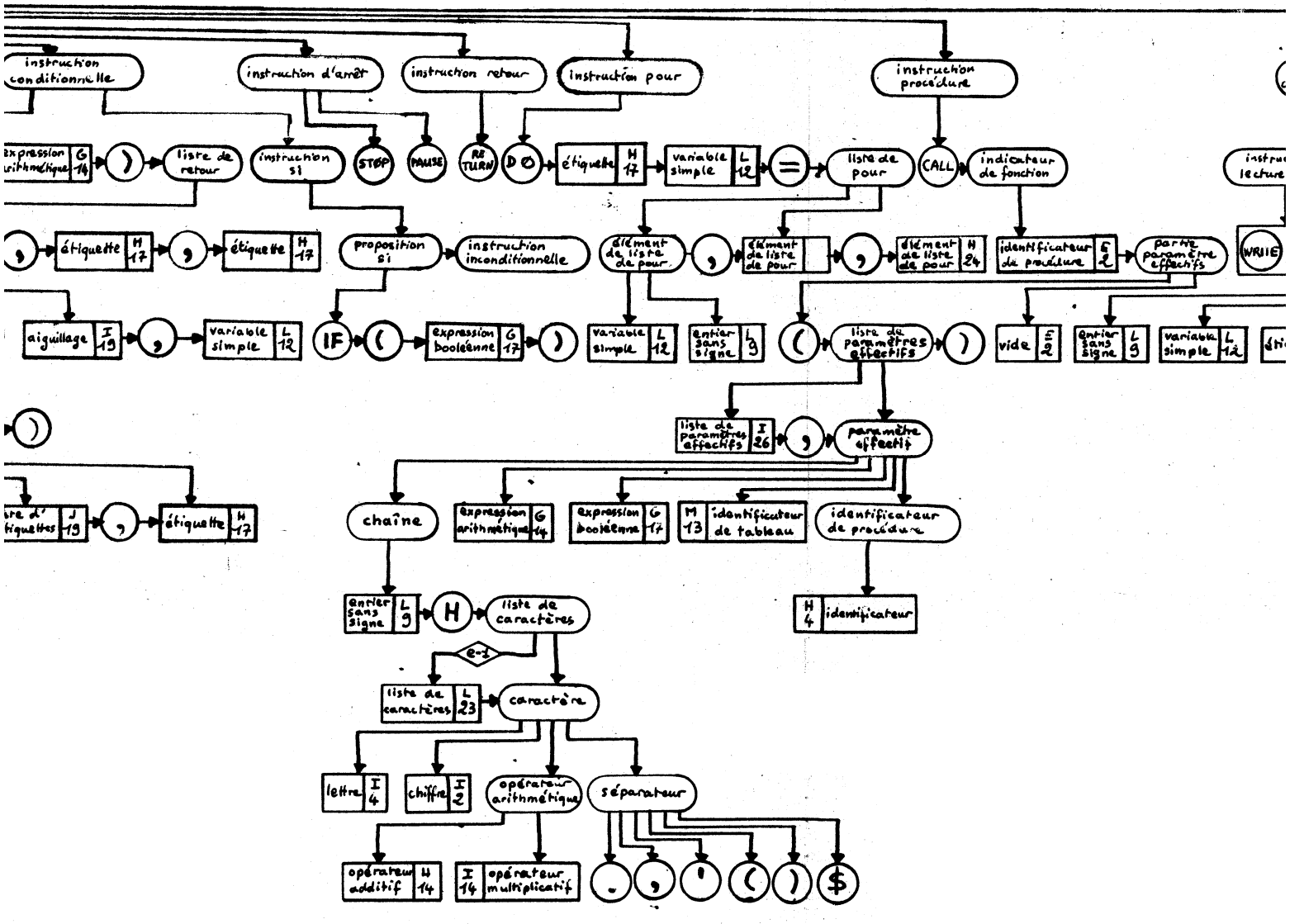

38
39
40
41
A
B
C
D
E
F
G
H
I
J
K
L
M
N
Ø
liste fermée
liste ouverte
spécification Hollerith
spécification Hollerith
liste ouverte
liste ouverte
séparateur d'enregistrement
liste ouverte
chaîne
entier sans signe
X
entier sans signe
spécification fixe simple
I Ø A
entier sans signe
entier sans signe
Olivier Lecarme
Juin 1964


modèle
étiquette H 17
FORMAT
(
liste de spécifications
)
ordre de manipulation
numéro de bande H 29
liste ouverte H 39
vide E 2
liste de spécifications F 35
séparateur de spécifications
liste de spécifications F 38
liste fermée
liste entrée-sortie
END FILE
REWIND
BACK SPACE
séparateur d'enregistrement
,
entier sans signe L 9
(
liste ouverte
)
variable entrée-sortie
(
liste d'entrée sortie H 22
,
partie indexage
)
séparateur d'enregistrement H 36
/
spécification numérique
spécification Hollerith
spécification Hollerith I 27
liste ouverte H 39
liste ouverte H 38
séparateur enregistrement
)
variable simple L 12
=
liste de pour G 25
spécification flottante
spécification fixe
chaîne K 22
entier sans signe L 9
X
spécification F
spécification E
spécification fixe simple K 33
entier sans signe L 9
spécification fixe simple
entier sans signe L 9
P
spécification F multiple
entier sans signe L 9
P
spécification E multiple
1 Ø A
entier sans signe L 9
entier sans signe L 9
spécification F simple
entier sans signe L 9
spécification E simple
F
entier sans signe L 9
.
entier sans signe L 9
E
entier sans signe L 9
.
entier sans signe L 9

IV

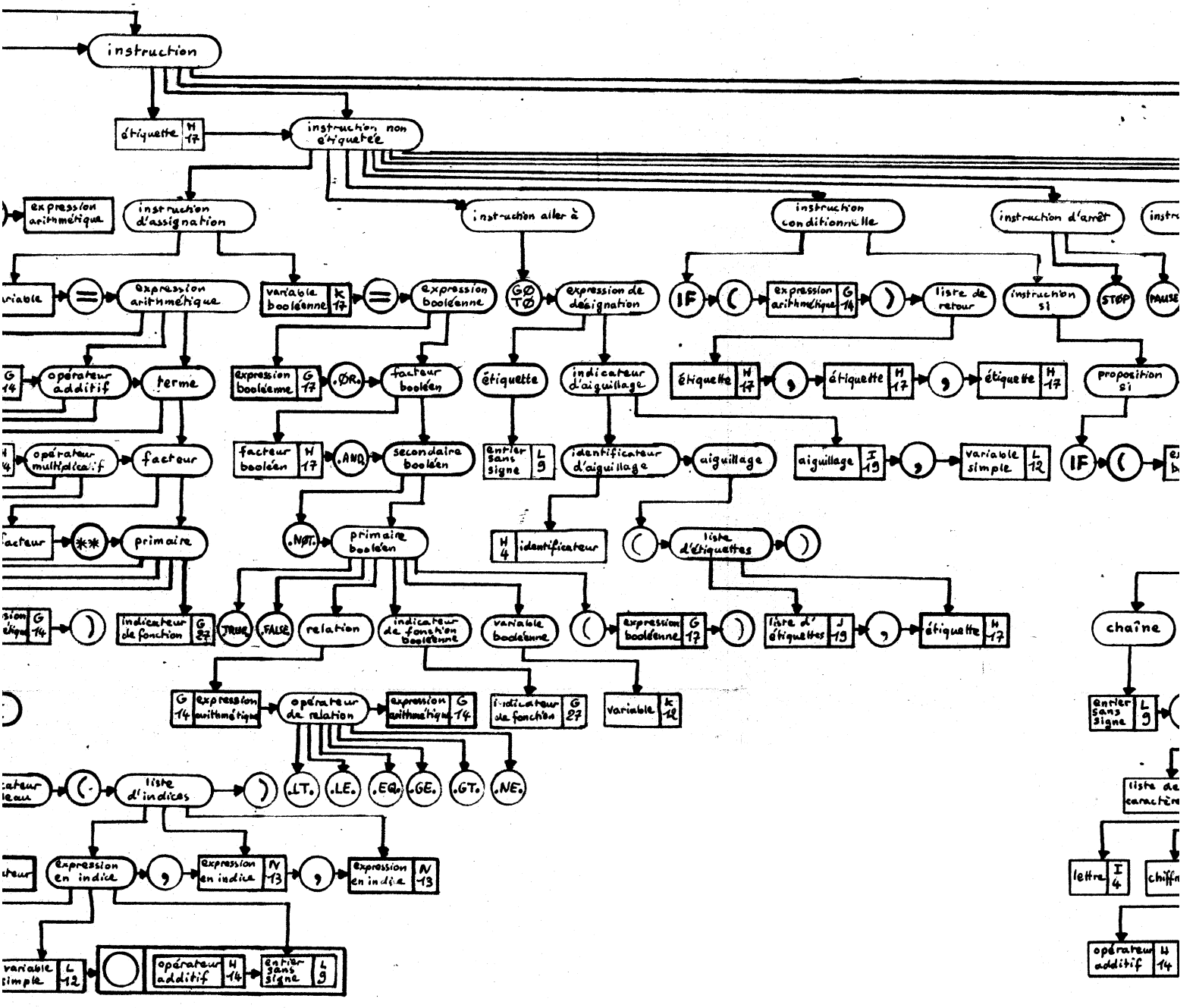

étiquette H 17
instruction muette
CONTINUE
modèle
instruction procédure
instruction d'entrée - sortie
liste de pour
CALL
indicateur de fonction
élément de liste de pour H 24
identificateur de procédure E 2
partie paramètre effectifs
instruction de lecture - écriture
WRITE
READ
numéro de bande
numéro de modèle
ordre de manipulation
numéro de bande H 29
liste d'entrée-sortie
END FILE
REWIND
BACK SPACE
étiquette H 17
FORMAT
liste ouverte H 39
vide E 2
liste de paramètres effectifs
vide E 2
entier sans signe L 9
variable simple L 12
étiquette H 17
identificateur de tableau M 13
liste d'entrée-sortie H 22
variable d'entrée-sortie
liste d'entrée-sortie H 22
partie indexage
K 12 variable
M 13 identificateur de tableau
K 12 variable
variable simple L 12
liste de pour G 25
spécification
entier sans signe L 9
P
spécification F multiple
entier sans signe L 9
spécification F simple
F

LANGAGE FORTRAN IV



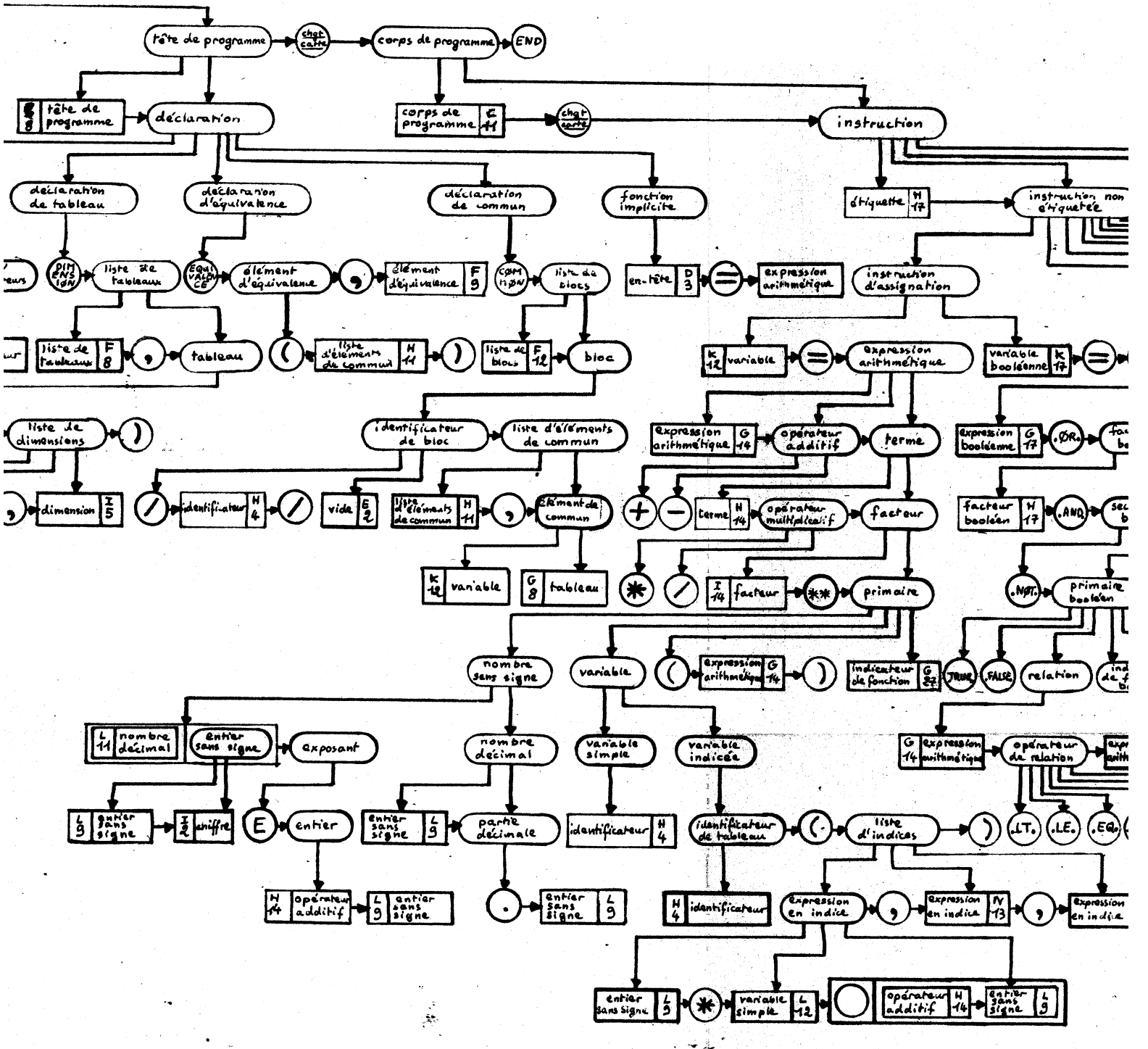
13 14 15 16 17 18 19 20 21 22

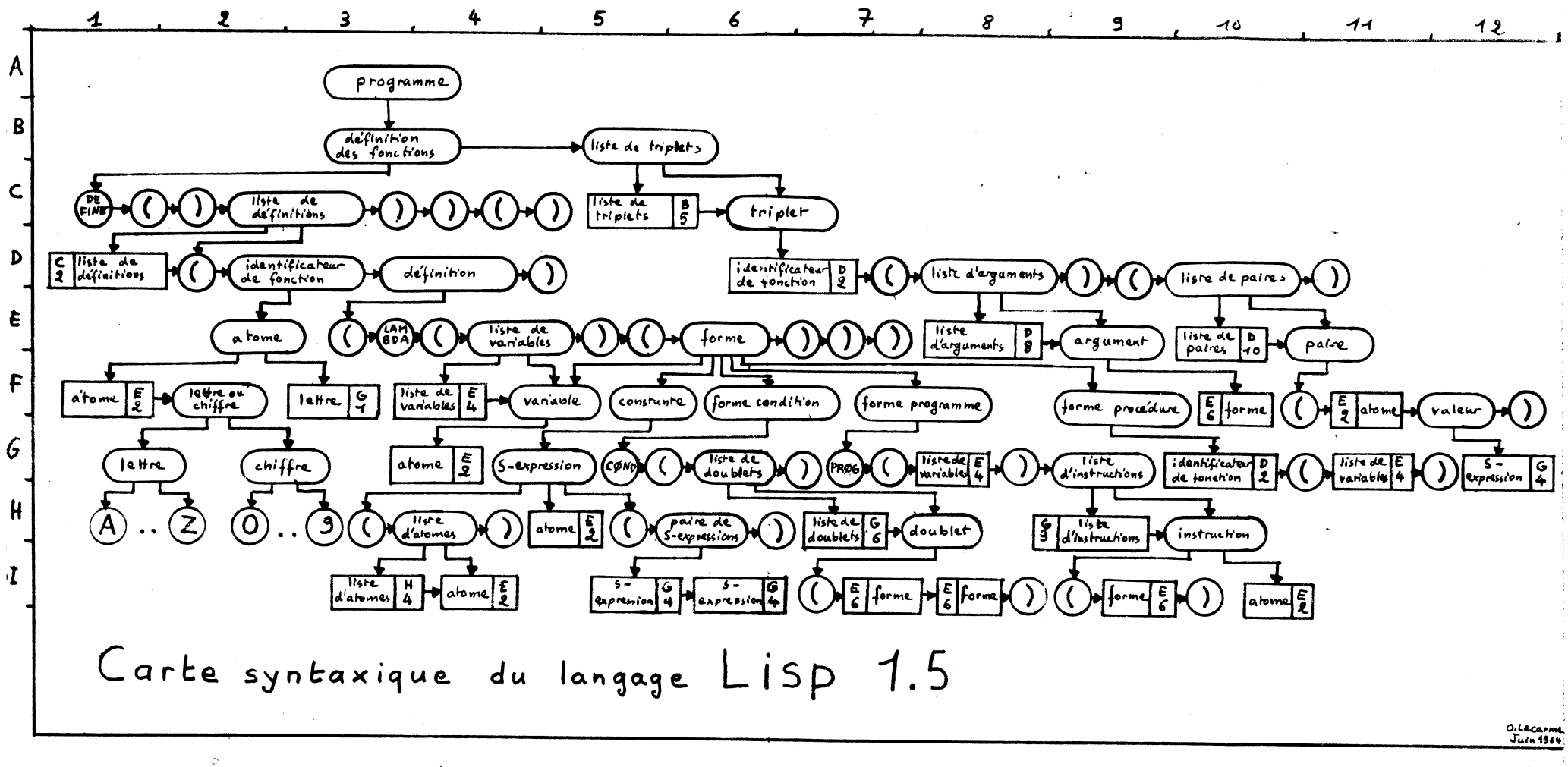
CARTE SYNTAXIQUE DU LANGAGE FC



7 8 9 10 11 12 13 14 15 16

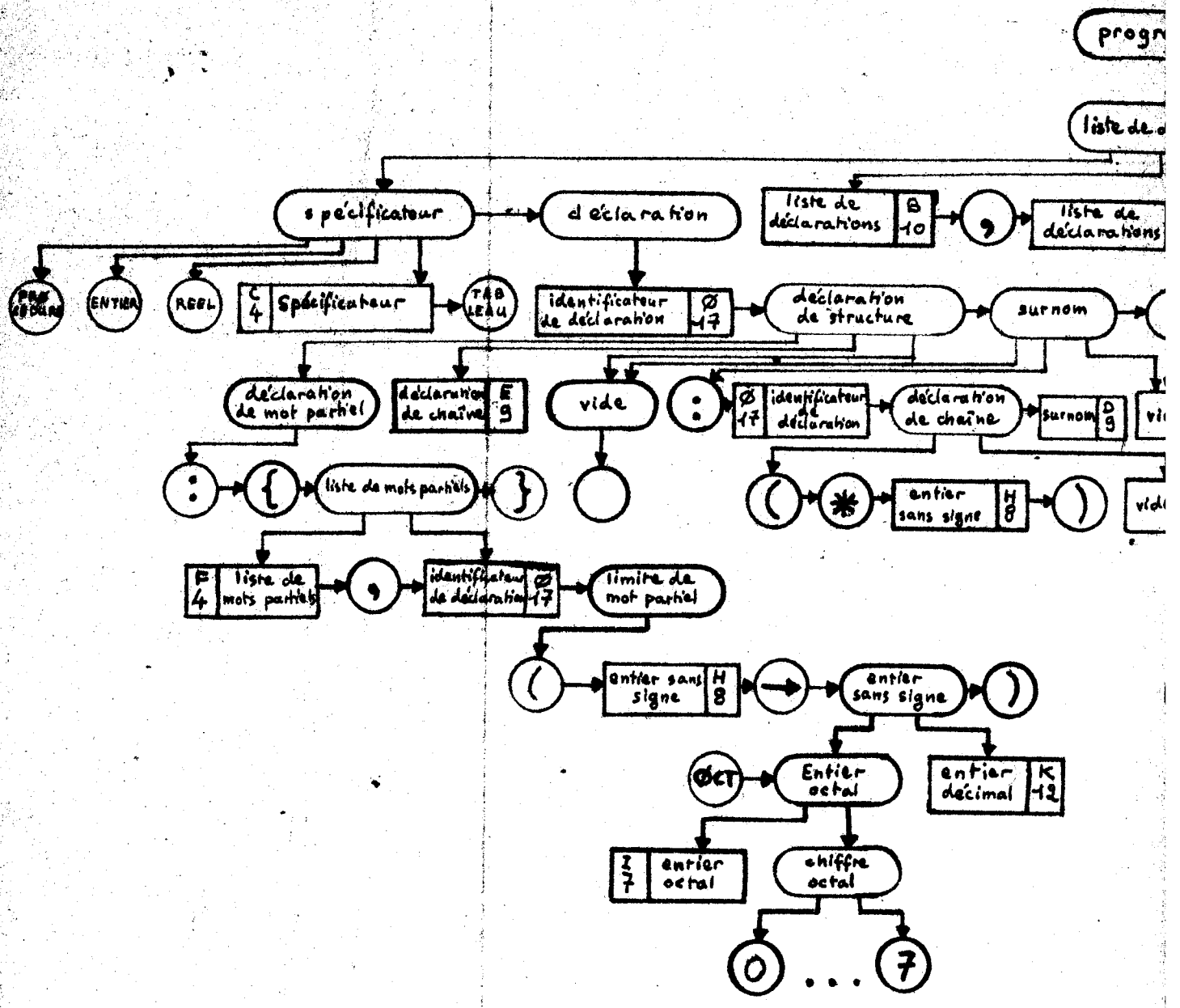
CARTE S



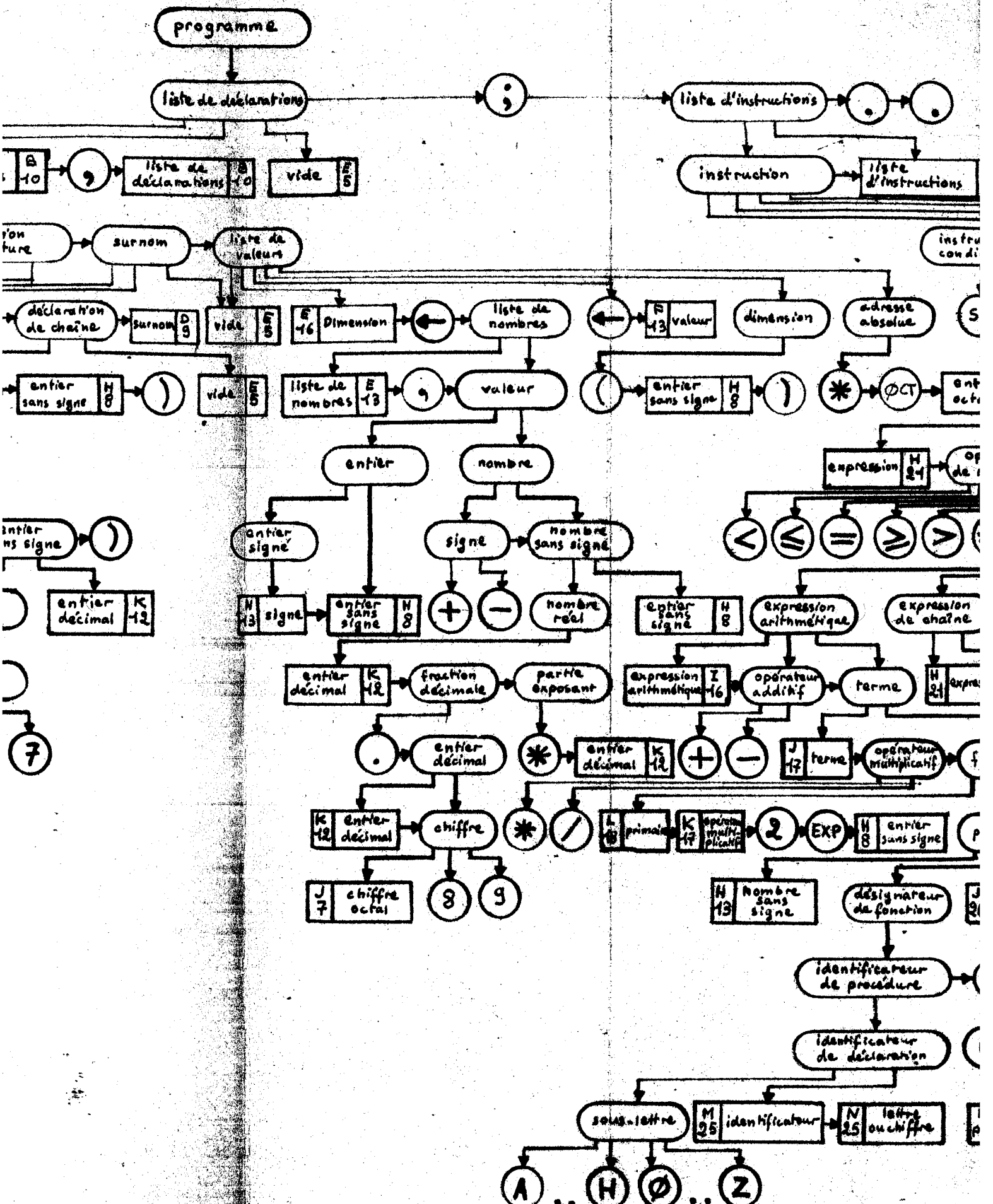


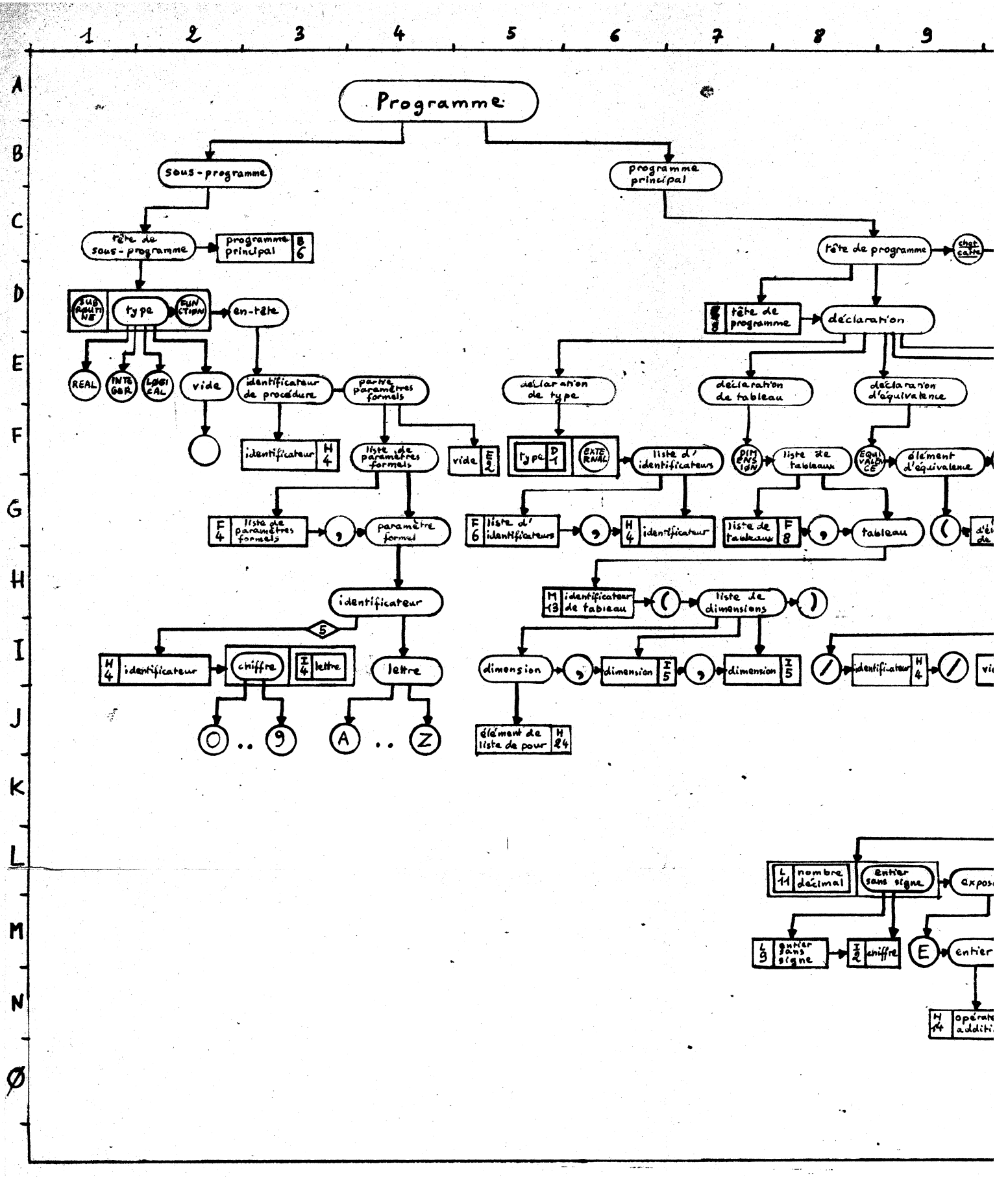
Carte syntaxique du langage Lisp 1.5

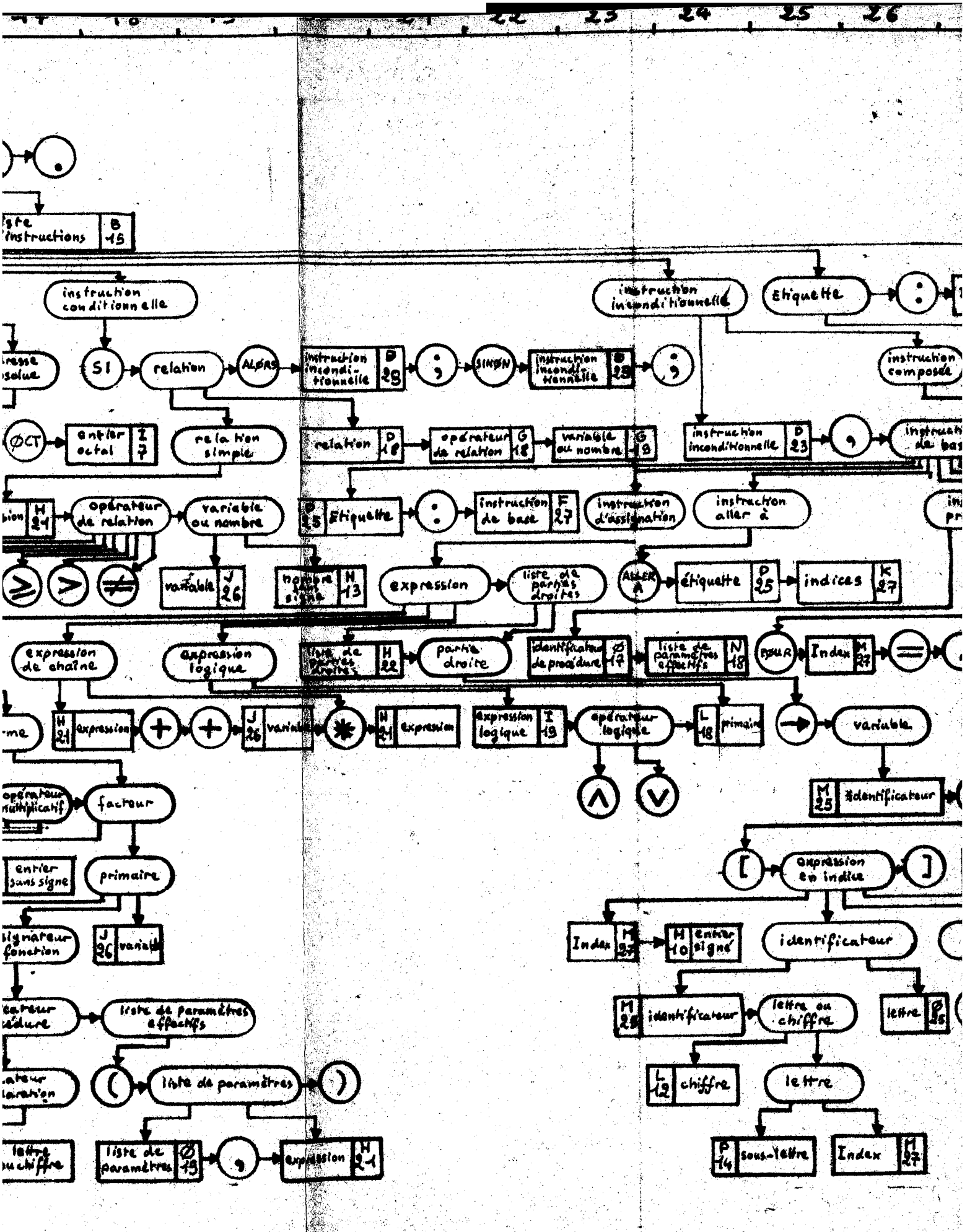
O. Lecarme
Juin 1969

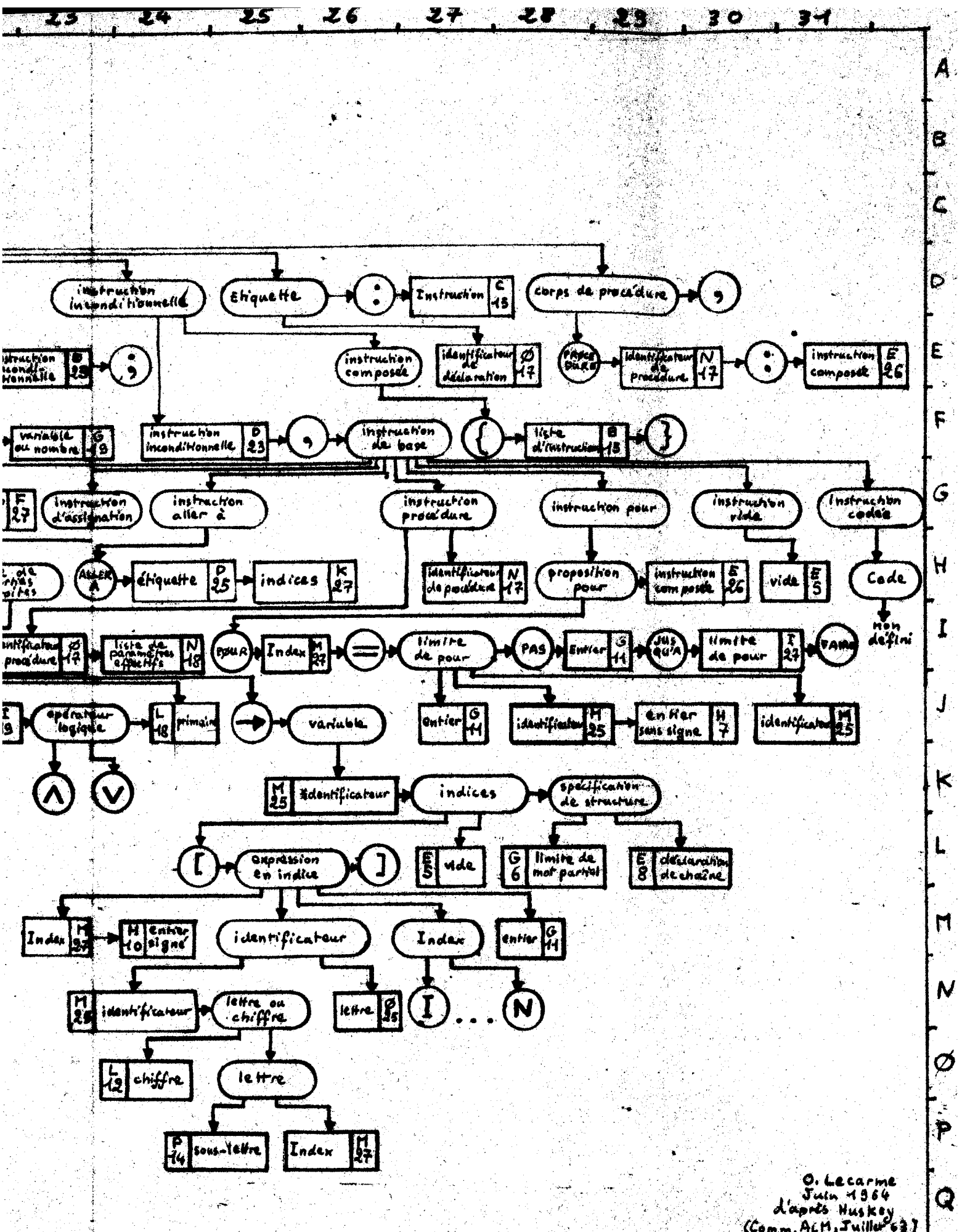


Carte syntaxique du langage NELIAC

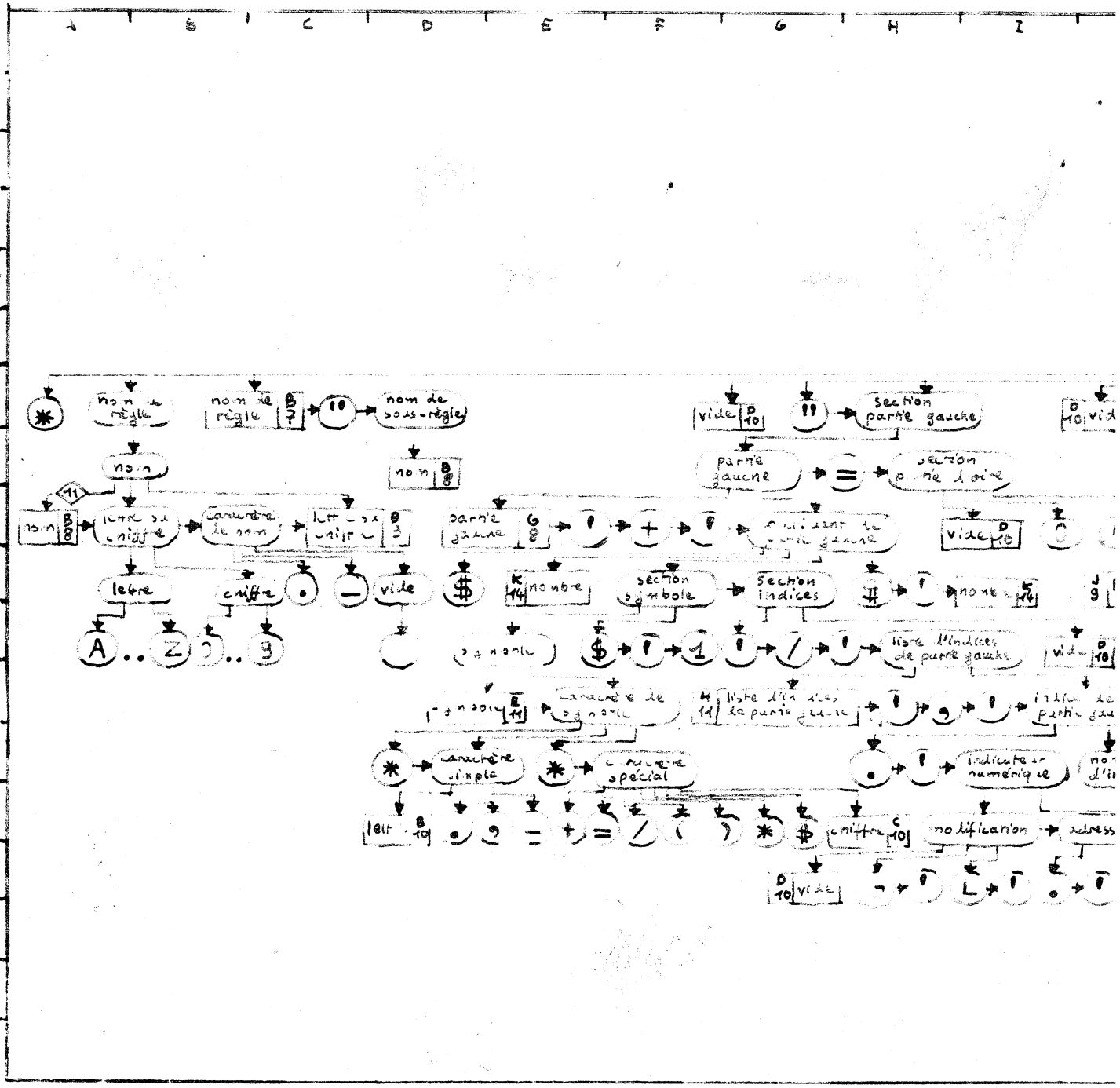

programme
liste de déclarations
;
liste d'instructions
.
.
B 10
,
liste de déclarations B 10
vide E 5
instruction
liste d'instructions
surnom
liste de valeurs
déclaration de chaîne
surnom D 9
vide E 5
E 16 Dimension
liste de nombres
F 13 valeur
dimension
adresse absolue
entier sans signe H 8
)
vide E 5
liste de nombres E 13
,
valeur
(
entier sans signe H 8
)
*
ØCT
entier
nombre
entier signé
signe
nombre sans signé
H 13 signe
entier sans signe H 8
+
−
nombre réel
entier sans signe H 8
entier décimal K 12
fraction décimale
partie exposant
.
entier décimal
*
entier décimal K 12
K 12 entier décimal
chiffre
J 7 chiffre octal
8
9
expression H 24
<
≤
=
≥
>
expression arithmétique
expression de chaîne
expression arithmétique I 16
opérateur additif
terme
+
−
J 17 terme
opérateur multiplicatif
*
/
L 18 primaire
K 17 opérateur multiplicatif
2
EXP
H 8 entier sans signe
H 13 nombre sans signe
désignateur de fonction
identificateur de procédure
identificateur de déclaration
sous-lettre
M 25 identificateur
N 25 lettre ou chiffre
A .. H
Ø .. Z

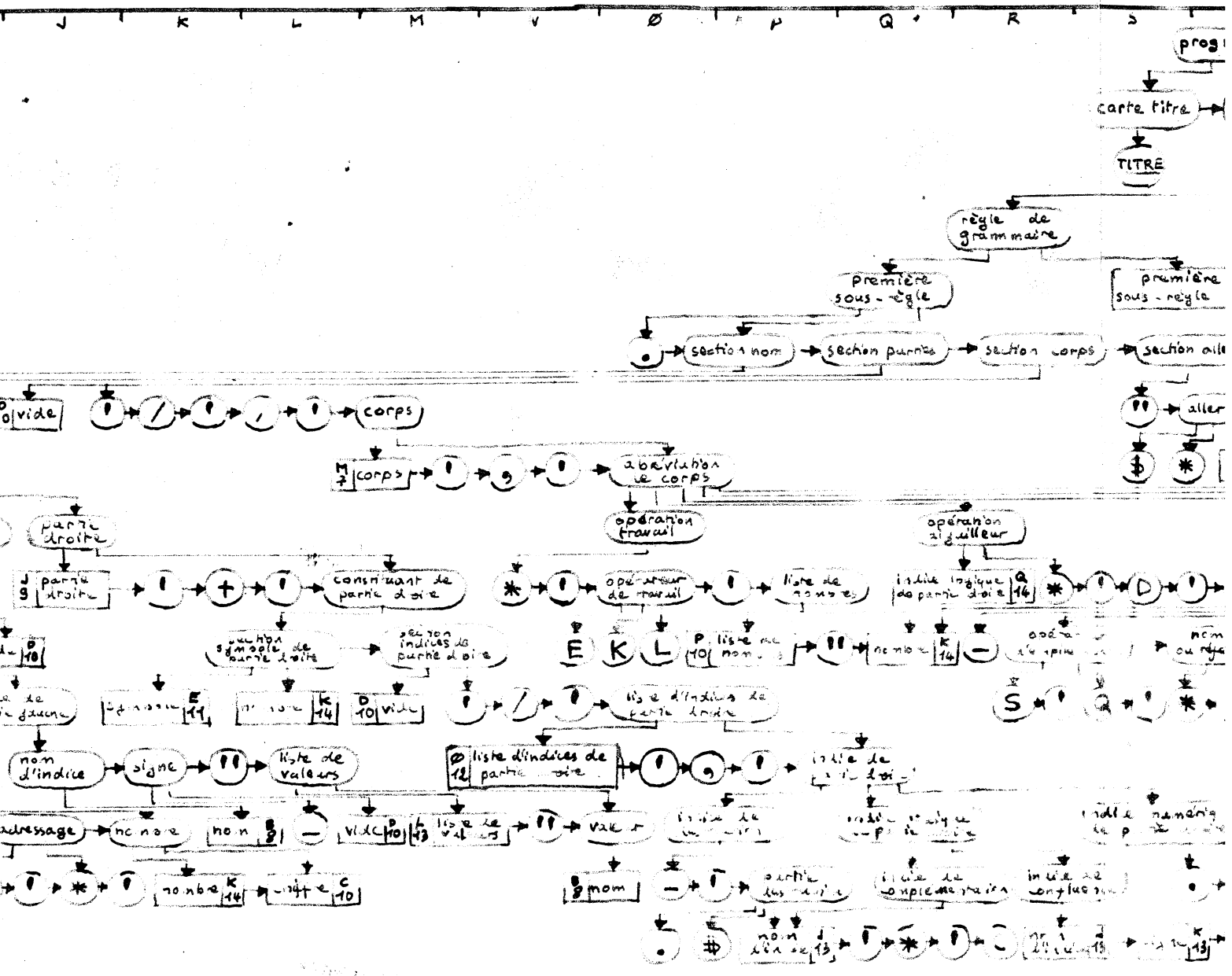

Programme
sous-programme
programme principal
tête de sous-programme
programme principal B 6
SUBROUTINE
type
FUNCTION
en-tête
REAL
INTEGER
LOGICAL
vide
identificateur de procédure
partie paramètres formels
identificateur H 4
liste de paramètres formels
vide E 2
F 4 liste de paramètres formels
,
paramètre formel
identificateur
5
H 4 identificateur
chiffre
I 4 lettre
lettre
0 .. 9
A .. Z
tête de programme
chaîne carte
C 8 tête de programme
déclaration
déclaration de type
déclaration de tableau
déclaration d'équivalence
type D 1
EXTERNAL
liste d'identificateurs
F 6 liste d'identificateurs
,
H 4 identificateur
DIMENSION
liste de tableaux
F 8 liste de tableaux
,
tableau
M 13 identificateur de tableau
(
liste de dimensions
)
dimension
,
dimension I 5
,
dimension I 5
EQUIVALENCE
élément d'équivalence
/
identificateur H 4
/
élément de liste de pour H 24
L 11 nombre décimal
entier sans signe
L 3 entier sans signe
I 2 chiffre
E
entier
H 14 opérateur additif

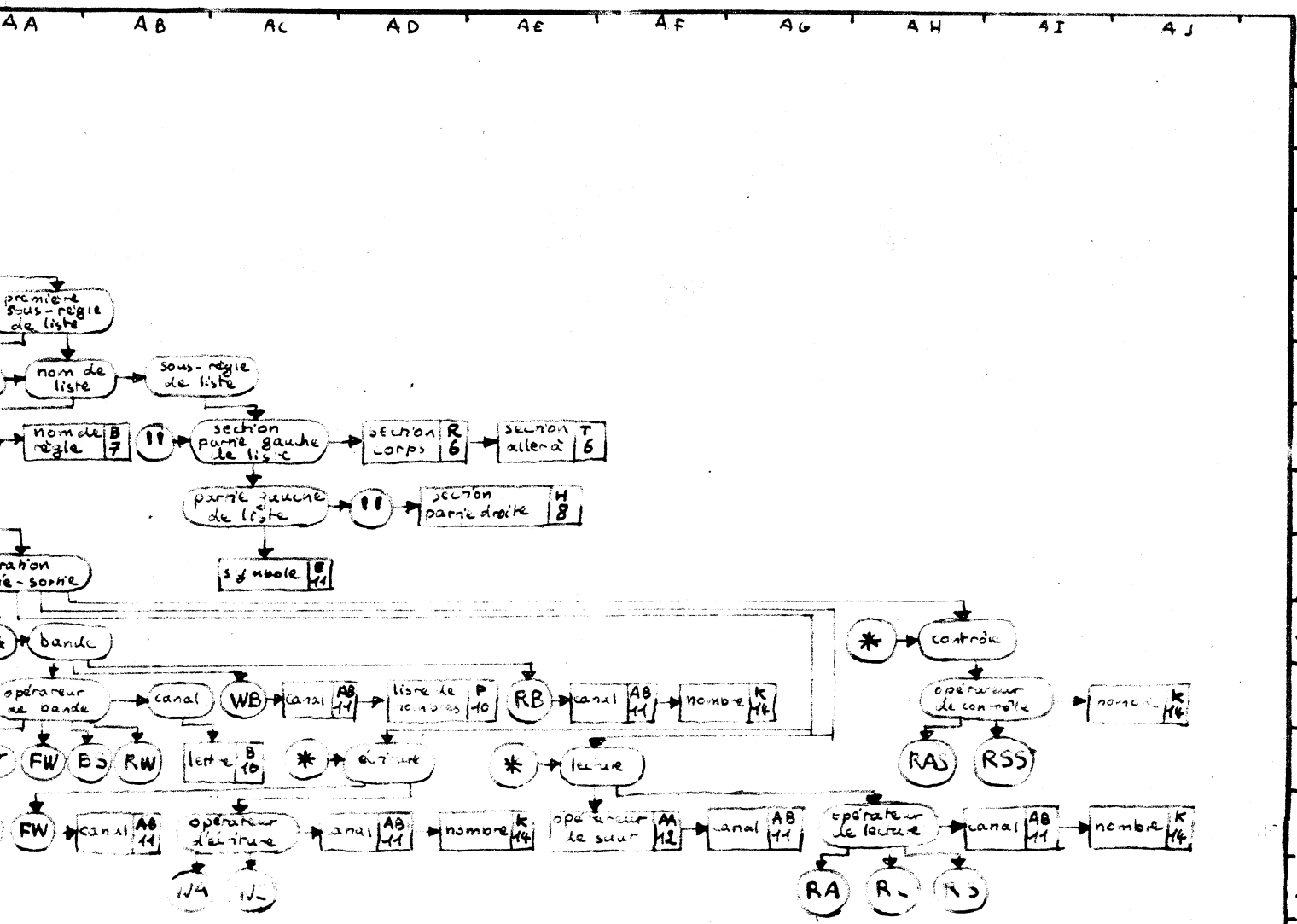

instruction conditionnelle
SI
relation
ALORS
instruction inconditionnelle D 23
SINON
instruction inconditionnelle D 23
instruction inconditionnelle
étiquette
instruction composée
relation D 18
opérateur de relation G 18
variable ou nombre G 18
instruction inconditionnelle D 23
étiquette D 25
instruction de base F 27
instruction d'assignation
instruction aller à
opérateur de relation
variable ou nombre
variable J 26
nombre sans signe N 13
expression
liste de parties droites
ALLER À
étiquette D 25
indices K 27
expression de chaîne
expression logique
liste de parties droites H 22
partie droite
identificateur de procédure Ø 17
liste de paramètres effectifs N 18
POUR
Index M 27
expression H 21
variable J 26
expression logique I 19
opérateur logique
primaire L 18
variable
facteur
entier sans signe
primaire
variable J 26
liste de paramètres effectifs
liste de paramètres
liste de paramètres Ø 19
expression N 21
identificateur M 25
expression en indice
Index M 27
entier sans signe H 10
identificateur
identificateur H 28
lettre ou chiffre
lettre P 15
chiffre L 12
lettre
sous-lettre P 14
Index M 27



O. Lecarme
Juin 1964
d'après Huskey
(Comm. ACM, Juillet 63)

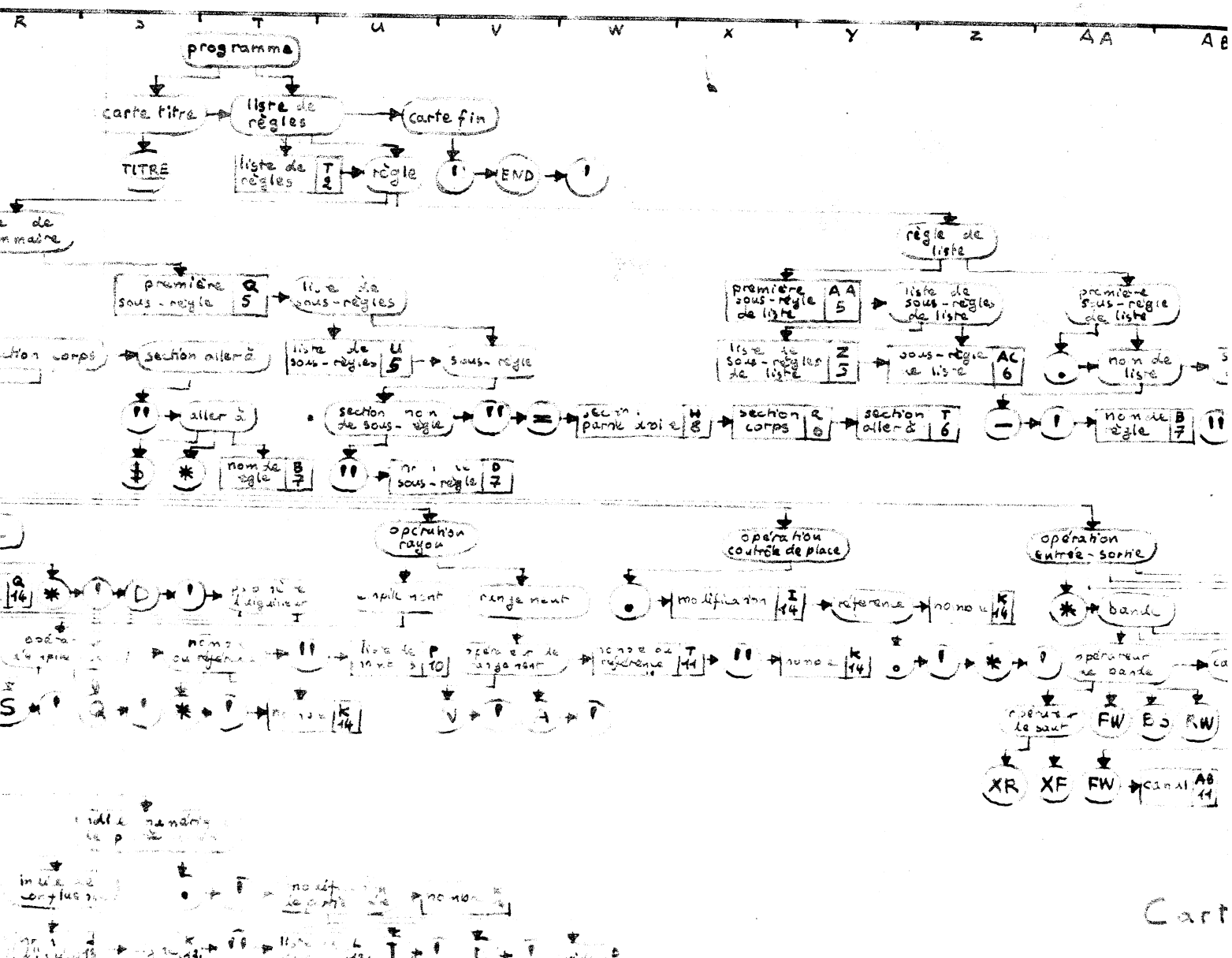

nom règle
nom de règle
nom de sous-règle
nom
lettre ou chiffre
caractère de nom
vide
lettre
chiffre
A .. Z
0 .. 9
nombre
section symbole
section indices
partie gauche
section partie droite
liste d'indices de partie gauche
caractère simple
caractère spécial
indicateur numérique
chiffre
modification
vide


J K L M V Ø P Q R S
prog
carte titre
TITRE
règle de grammaire
première sous-règle
première sous-règle
section nom
section partie
section corps
section alle
vide
corps
corps
abréviation de corps
opération travail
opération aiguilleur
partie droite
partie droite
constituant de partie droite
opérateur de travail
liste de nombres
indice logique de partie droite
aller
liste d'indices de partie droite
liste d'indices de partie droite
nom d'indice
signe
liste de valeurs
adressage
nombre
nombre
valeur
nombre
nombre



Carte syntaxique du langage Comit

O. Lecarme
4 Juin 64


R
S
T
U
V
W
X
Y
Z
AA
programme
carte titre
liste de règles
carte fin
TITRE
END
règle de liste
première sous-règle
liste de sous-règles
section corps
section aller à
nom de sous-règle
nom de règle
opération rayon
opération contrôle de place
opération entrée-sortie
modification
référence
bande
opérateur de bande
opérateur de saut
FW
EB
RW
XR
XF
canal

Cart