TU 220

N° d'ordre

THESE

présentée à

L'UNIVERSITE SCIENTIFIQUE ET MEDICALE DE GRENOBLE

pour obtenir

LE GRADE DE DOCTEUR DE TROISIEME CYCLE

"Mathématiques Spécialité Informatique"

par

DANIEL SUTY

AUTOCOMPILATION
ET
CONCEPTUALISATION
DES LANGAGES DE PROGRAMMATION

Thèse soutenue le 4 Novembre 1971, devant la commission d'examen:

MM.	N.	**Gastinel**	Président
	L.	**Bolliet**	Examinateur
	J.C.	**Boussard**	Examinateur
	J.P.	**Rossiensky**	Examinateur

UNIVERSITE SCIENTIFIQUE
ET MEDICALE DE GRENOBLE

LISTE DES PROFESSEURS

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

PROFESSEURS TITULAIRES

MM.	ANGLES D'AURIAC Paul	Mécanique des fluides
	ARNAUD Georges	Clinique des maladies infectieuses
	ARNAUD Paul	Chimie
	AYANT Yves	Physique approfondie
	BARBIER Jean-Claude	Physique expérimentale
	BARBIER Reynold	Géologie appliquée
	BARJON Robert	Physique nucléaire
	BARNOUD Fernand	Biosynthèse de la cellulose
	BARRIE Joseph	Clinique chirurgicale
	BENOIT Jean	Radioélectricité
	BESSON Jean	Electrochimie
	BEZES Henri	Chirurgie générale
	BLAMBERT Maurice	Mathématiques pures
	BOLLIET Louis	Informatique (IUT B)
	BONNET Georges	Electrotechnique
	BONNET Jean-Louis	Clinique ophtalmologique
	BONNET-EYMARD Joseph	Pathologie médicale
	BONNIER Etienne	Electrochimie Electrométallurgie
	BOUCHERLE André	Chimie et Toxicologie
	BOUCHEZ Robert	Physique nucléaire
	BRAVARD Yves	Géographie
	BRISSONNEAU Pierre	Physique du Solide
	BUYLE-BODIN Maurice	Electronique
	CABANAC Jean	Pathologie chirurgicale
	CABANEL Guy	Clinique rhumatologique et hydrologique
	CALAS François	Anatomie
	CARRAZ Gilbert	Biologie animale et pharmacodynamie
	CAU Gabriel	Médecine légale et Toxicologie
	CAUQUIS Georges	Chimie organique
	CHABAUTY Claude	Mathématiques pures
	CHATEAU Robert	Thérapeutique
	CHENE Marcel	Chimie papetière
	COEUR André	Pharmacie chimique
	CONTAMIN Robert	Clinique gynécologique
	COUDERC Pierre	Anatomie Pathologique
	CRAYA Antoine	Mécanique
Mme	DEBELMAS Anne-Marie	Matière médicale
MM.	DEBELMAS Jacques	Géologie générale
	DEGRANGE Charles	Zoologie
	DESSAUX Georges	Physiologie animale
	DODU Jacques	Mécanique appliquée
	DREYFUS Bernard	Thermodynamique
	DUCROS Pierre	Cristallographie
	DUGOIS Pierre	Clinique de Dermatologie et Syphiligraphie

	FAU René	Clinique neuro-psychiatrique
	FELICI Noël	Electrostatique
	GAGNAIRE Didier	Chimie physique
	GALLISSOT François	Mathématiques pures
	GALVANI Octave	Mathématiques pures
	GASTINEL Noël	Analyse numérique
	GERBER Robert	Mathématiques pures
	GIRAUD Pierre	Géologie
	KLEIN Joseph	Mathématiques pures
Mme	KOFLER Lucie	Botanique et physiologie végétale
MM.	KOSZUL Jean-Louis	Mathématiques pures
	KRAVTCHENKO Julien	Mécanique
	KUNTZMANN Jean	Mathématiques appliquées
	LACAZE Albert	Thermodynamique
	LACHARME Jean	Biologie végétale
	LATURAZE Jean	Biochimie pharmaceutique
	LEDRU Jean	Clinique médicale B
	LLIBOUTRY Louis	Géophysique
	LOUP Jean	Géographie
Mle	LUTZ Elisabeth	Mathématiques pures
MM.	MALGRANGE Bernard	Mathématiques pures
	MALINAS Yves	Clinique obstétricale
	MARTIN-NOEL Pierre	Séméiologie médicale
	MASSEPORT Jean	Géographie
	MAZARE Yves	Clinique médicale A
	MICHEL Robert	Minéralogie et Pétrographie
	MOURIQUAND Claude	Histologie
	MOUSSA André	Chimie nucléaire
	NEEL Louis	Physique du Solide
	OZENDA Paul	Botanique
	PAUTHENET René	Electrotechnique
	PAYAN Jean-Jacques	Mathématiques pures
	PEBAY-PEYROULA Jean-Claude	Physique
	PERRET René	Servomécanismes
	PILLET Emile	Physique industrielle
	RASSAT André	Chimie systématique
	RENARD Michel	Thermodynamique
	REULOS René	Physique industrielle
	RINALDI Renaud	Physique
	ROGET Jean	Clinique de pédiatrie et de puériculture
	SANTON Lucien	Mécanique
	SEIGNEURIN Raymond	Microbiologie et Hygiène
	SENGEL Philippe	Zoologie
	SILBERT Robert	Mécanique des fluides
	SOUTIF Michel	Physique générale
	TANCHE Maurice	Physiologie
	TRAYNARD Philippe	Chimie générale
	VAILLAND François	Zoologie
	VAUQUOIS Bernard	Calcul électronique
Mme	VERAIN Alice	Pharmacie galénique
	VERAIN André	Physique
Mme	VEYRET Germaine	Géographie
MM.	VEYRET Paul	Géographie
	VIGNAIS Pierre	Biochimie médicale
	YOCCOZ Jean	Physique nucléaire théorique

PROFESSEURS ASSOCIES

MM.	BULLEMER Bernhard	Physique
	RADHAKRISHNA Pidatala	Thermodynamique

PROFESSEURS SANS CHAIRE

MM.	AUBERT Guy	Physique
Mme	BARBIER Marie-Jeanne	Electrochimie
MM.	BARRA Jean	Mathématiques appliquées
	BEAUDOING André	Pédiatrie
	BERTRANDIAS Jean-Paul	Mathématiques appliquées
	BIAREZ Jean-Pierre	Mécanique
	BONNETAIN Lucien	Chimie minérale
Mme	BONNIER Jane	Chimie générale
MM.	CARLIER Georges	Biologie végétale
	COHEN Joseph	Electrotechnique
	COUMES André	Radioélectricité
	DEPASSEL Roger	Mécanique des Fluides
	DEPORTES Charles	Chimie minérale
	DESRE Pierre	Métallurgie
	DOLIQUE Jean-Michel	Physique des plasmas
	GAUTHIER Yves	Sciences biologiques
	GEINDRE Michel	Electroradiologie
	GIDON Paul	Géologie et Minéralogie
	GLENAT René	Chimie organique
	HACQUES Gérard	Calcul numérique
	JANIN Bernard	Géographie
Mme	KAHANE Josette	Physique
MM.	LATREILLE René	Chirurgie générale
	LAURENT Pierre	Mathématiques appliquées
	MULLER Jean-Michel	Thérapeutique
	PERRIAUX Jean-Jacques	Géologie et minéralogie
	POULOUJADOFF Michel	Electrotechnique
	REBECQ Jacques	Biologie (CUS)
	REVOL Michel	Urologie
	REYMOND Jean-Charles	Chirurgie générale
	ROBERT André	Chimie papetière
	SARRAZIN Roger	Anatomie et chirurgie
	SARROT-REYNAULD Jean	Géologie
	SIBILLE Robert	Construction Mécanique
	SIROT Louis	Chirurgie générale
Mme	SOUTIF Jeanne	Physique générale
M.	VALENTIN Jacques	Physique nucléaire

MAITRES DE CONFERENCES ET MAITRES DE CONFERENCES AGREGES

Mle	AGNIUS-DELORD Claudine	Physique pharmaceutique
	ALARY Josette	Chimie analytique
MM.	AMBLARD Pierre	Dermatologie
	AMBROISE-THOMAS Pierre	Parasitologie
	ARMAND Yves	Chimie

	BEGUIN Claude	Chimie organique
	BELORIZKY Elie	Physique
	BENZAKEN Claude	Mathématiques appliquées
Mme	BERTRANDIAS Françoise	Mathématiques pures
MM.	BLIMAN Samuel	Electronique (EIE)
	BLOCH Daniel	Electrotechnique
Mme	BOUCHE Liane	Mathématiques (CUS)
MM.	BOUCHET Yves	Anatomie
	BOUSSARD Jean-Claude	Mathématiques appliquées
	BOUVARD Maurice	Mécanique des Fluides
	BRIERE Georges	Physique expérimentale
	BRODEAU François	Mathématiques (IUT B)
	BRUGEL Lucien	Energétique
	BUISSON Roger	Physique
	BUTEL Jean	Orthopédie
	CHAMBAZ Edmond	Biochimie médicale
	CHAMPETIER Jean	Anatomie et organogénèse
	CHARACHON Robert	Oto-Rhino-Laryngologie
	CHIAVERINA Jean	Biologie appliquée (EFP)
	CHIBON Pierre	Biologie animale
	COHEN-ADDAD Jean-Pierre	Spectrométrie physique
	COLOMB Maurice	Biochimie médicale
	CONTE René	Physique
	CROUZET Guy	Radiologie
	DURAND Francis	Métallurgie
	DUSSAUD René	Mathématiques (CUS)
Mme	ETERRADOSSI Jacqueline	Physiologie
MM.	FAURE Jacques	Médecine légale
	GAVEND Michel	Pharmacologie
	GENSAC Pierre	Botanique
	GERMAIN Jean-Pierre	Mécanique
	GIDON Maurice	Géologie
	GRIFFITHS Michael	Mathématiques appliquées
	GROULADE Joseph	Biochimie médicale
	HOLLARD Daniel	Hématologie
	HUGONOT Robert	Hygiène et médecine préventive
	IDELMAN Simon	Physiologie animale
	IVANES Marcel	Electricité
	JALBERT Pierre	Histologie
	JOLY Jean-René	Mathématiques pures
	JOUBERT Jean-Claude	Physique du Solide
	JULLIEN Pierre	Mathématiques pures
	KAHANE André	Physique générale
	KUHN Gérard	Physique
Mme	LAJZEROWICZ Jeannine	Physique
MM.	LAJZEROWICZ Joseph	Physique
	LANCIA Roland	Physique atomique
	LE JUNTER Noël	Electronique
	LEROY Philippe	Mathématiques
	LOISEAUX Jean-Marie	Physique nucléaire
	LONGEQUEUE Jean-Pierre	Physique nucléaire
	LUU DUC Cuong	Chimie organique
	MACHE Régis	Physiologie végétale
	MAGNIN Robert	Hygiène et Médecine préventive
	MARECHAL Jean	Mécanique
	MARTIN-BOUYER Michel	Chimie (CUS)
	MAYNARD Roger	Physique du Solide
	MICOUD Max	Maladies infectieuses
	MOREAU René	Hydraulique (INP)

	NEGRE Robert	Mécanique
	PARAMELLE Bernard	Pneumologie
	PECCOUD François	Analyse (IUT B)
	PEFFEN René	Métallurgie
	PELMONT Jean	Physiologie animale
	PERRET Jean	Neurologie
	PERRIN Louis	Pathologie expérimentale
	PFISTER Jean-Claude	Physique du Solide
	PHELIP Xavier	Rhumatologie
Mle	PIERY Yvette	Biologie animale
	RACHAIL Michel	Médecine interne
	RACINET Claude	Gynécologie et obstétrique
	RICHARD Lucien	Botanique
Mme	RINAUDO Marguerite	Chimie macromoléculaire
MM.	ROMIER Guy	Mathématiques (IUT B)
	ROUGEMONT (DE) Jacques	Neuro-chirurgie
	STIEGLITZ Paul	Anesthésiologie
	STOEBNER Pierre	Anatomie pathologique
	VAN CUTSEM Bernard	Mathématiques appliquées
	VEILLON Gérard	Mathématiques appliquées (INP)
	VIALON Pierre	Géologie
	VOOG Robert	Médecine interne
	VROUSSOS Constantin	Radiologie
	ZADWORNY François	Electronique

MAITRES DE CONFERENCES ASSOCIES

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

CHARGES DE FONCTIONS DE MAITRES DE CONFERENCES

Mme	BERIEL Hélène	Physiologie
Mme	RENAUDET Jacqueline	Microbiologie

Je tiens à exprimer ma reconnaissance à :

Monsieur le Professeur N. GASTINEL, *Professeur à l'Université Scientifique et Médicale de Grenoble qui a bien voulu me faire l'honneur de présider le jury après m'avoir prodigué de nombreux conseils,*

Monsieur le Professeur L. BOLLIET, *Professeur à l'I.U.T. B qui m'a permis de commencer cette thèse et d'en délimiter les champs d'investigation et les domaines d'application,*

Monsieur J.C. BOUSSARD, *Maître de Conférences à l'Université Scientifique et Médicale de Grenoble, qui m'a bien souvent aidé et orienté dans mes travaux,*

Monsieur ROSSIENSKY, *Directeur du Département Software et Systèmes Spéciaux de CEGOS Informatique qui a bien voulu s'intéresser à ce travail.*

Je remercie également mes collègues du Laboratoire de Calcul, spécialement Monsieur CALLEGHER,

Enfin, je remercie les personnes qui ont participé à la réalisation matérielle de cette thèse, en particulier, Madame J. CARRY *pour la frappe et tous les membres du service tirage.*

TABLE DES MATIERES

-:=-:=-:=-:=-

TABLE DES MATIERES

INTRODUCTION 1

I - UNE METHODE D'AUTOCOMPILATION : LE BOOTSTRAPPING

I - 1. Introduction 9

I - 2. Principe général 9

I - 3. Définition 10

I - 4. Remarques 14

I - 5. Conclusion 16

II - EXEMPLES DE REALISATION

II - 1. Introduction 19

II - 2. P.L. 360 19

II - 3. Algol 60 23

II - 4. Fortran H 24

III - ESSAI DE FONDEMENT CONCEPTUEL DES LANGAGES

III - 1. Rapport avec le bootstrapping 27

III - 2. Définitions 28

III - 3. Sémantique des langages de programmation 29

III - 4. Conceptualisation. Essai 30

III - 5. Réalisations identiques 50

III - 6. Sous-ensembles optimum 51

III - 7. Conclusion 52

IV - ESSAI D'APPLICATION A FORTRAN

IV - 1. Introduction 54

IV - 2. Détermination des concepts Fortran 54

IV - 3. Détermination du sous-ensemble de Fortran 58

IV - 4. Compilation du sous-ensemble 60

IV - 5. Second compilateur 62

CONCLUSION 69

BIBLIOGRAPHIE 74

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

-:=-:=-

INTRODUCTION

Cette introduction se veut informelle, elle montre simplement comment notre étude a progressé et comment - principalement par la lecture d'articles - nous avons été amenés à étudier deux points particuliers :

1) L'étude d'une méthode d'autocompilation dites du Bootstrapping

2) Un essai de définition formelle des langages de programmation d'un point de vue sémantique l'étude du bootstrapping nous ayant conduit à envisager la création d'un "langage noyau".

Dans toute la suite du texte, nous emploierons très souvent le mot concept.

Par concept, nous entendons "idée générale" c'est-à-dire capable de généralisation et d'extension, la sémantique d'un langage de programmation étant l'ensemble des concepts contenus dans ce langage.

Depuis l'avènement des ordinateurs, la recherche en informatique a dépensé une somme d'efforts énormes afin de permettre la communication entre l'homme et la machine. Un des buts à atteindre étant un langage compréhensible pour tous, hommes et machines.

Hélas, le manque de coordination des recherches dans le domaine des langages de programmation a conduit à un nombre invraisemblable de langages, ce qui fait que l'on voit mal comment on pourra arriver un jour à un langage unique.

Nous allons revoir l'évolution des langages de programmation en essayant, pour chaque étape de dégager les motivations et les résultats obtenus.

La première "victoire linguistique" fut l'avènement des assembleurs.

A ce niveau le langage est spécifique de l'ordinateur considéré. Il contient toute sa puissance sémantique, ce qui permet une programmation de même niveau que celle en langage machine proprement dit, une

action étant décomposée en éléments de base n'effectuant qu'une opération élémentaire.

L'assembleur donne d'énormes facilités au programmeur. Ce dernier n'a plus à s'occuper des calculs d'adresses, ceux-ci étant effectués automatiquement. D'autre part, les concepts qu'il fournit lui donnent un rendement bien plus élevé que ce que pourrait fournir la seule constitution logique de la machine (identificateurs symboliques, calcul d'adresses, macro-instructions ...).

Néanmoins, l'assembleur nécessite une écriture très spécialisée des programmes, la syntaxe du langage étant toujours très limitée. De plus, la programmation en langage d'assemblage nécessite des dispositions spéciales qu'il est parfois difficile d'avoir, la décomposition en instructions élémentaires demandant un esprit d'analyse développé.

Bien que très proche de l'ordinateur - on fait souvent la confusion, en appelant langage machine le langage d'assemblage - il est nécessaire d'avoir un traducteur (assembleur) qui considérant un programme, le traduira en code interne compréhensible par la machine. On comprend donc que ce traducteur doive être écrit (au moins en partie) dans un langage de niveau moins élevé que celui qu'il veut traiter.

L'extrême souplesse d'emploi, sa puissance, font du langage d'assemblage un outil encore très employé dans l'écriture du software d'un ordinateur.

Mais la programmation en langage d'assemblage présente de nombreux inconvénients. Elle devient un véritable métier spécialisé et il est difficile de demander à un ingénieur non informaticien d'apprendre un langage relativement difficile et vite périmé du fait du développement rapide des ordinateurs.

D'autre part, la communication de programmes est un phénomène de plus en plus courant et ceci est difficile si le programme est écrit dans un langage peu explicite en lui-même et complètement incompréhensible pour qui ne connaît pas la machine pour laquelle il a été écrit.

L'étape suivante fut donc la recherche de langages indépendants des ordinateurs eux-mêmes, les buts principaux à atteindre étant :

- Totale indépendance du langage vis-à-vis de la machine
- communication aisée entre les différents utilisateurs
- programmation simplifiée

- détection d'un plus grand nombre d'erreurs durant la traduction du programme, ce qui conduit à une programmation plus efficace pour la mise au point.

Ce travail demande non seulement des recherches dans le domaine de la création des langages mais aussi dans celui de la création du traducteur (compilateur) qui analysant le texte qui lui est fourni, doit produire un programme tel que la machine puisse l'exécuter. Il est souhaitable d'avoir un traducteur rapide fournissant un code performant lors de son exécution.

Lorsque vers 1950 on aborda ces problèmes, on se pencha plus sur la création des traducteurs que sur la création des langages eux-
f IV-11) mêmes. On peut donner deux excuses à cela :

- La première est que du fait de l'utilisation relativement limitée des ordinateurs, la puissance sémantique demandée aux langages était faible.
- La seconde est que ces ordinateurs étant peu performants, il fallait optimiser au maximum le temps de compilation ainsi que le programme généré.

On peut donc regretter que pendant plus de dix ans, les recherches portèrent sur la syntaxe des langages et plus encore sur l'analyse syntaxique.

Si ces recherches furent évidemment des plus utiles, il semble qu'elles aient caché le problème de la conception des langages eux-mêmes. Tous les langages créés à partir de 1955 :

Fortran, IPL, 1956
Neliac, APL, 1958
Cobol, Jovial, 1959
Algol 60,

contiennent des parties communes et des concepts redondants ou inutiles.

Dans un langage de programmation, le but recherché est d'exprimer un concept simplement, même si celui-ci une fois traduit entraîne l'application d'un grand nombre de concepts de niveau inférieur. Dans les défi-

nitions des langages antérieurs à 1968, la conceptualisation était relativement anarchique; certains concepts ont été faussement généralisés sans que l'on s'aperçoive que l'on déformait ainsi le concept.

L'étiquette numérique d'Algol 60 est un exemple de cette généralisation. Fortran possedant l'étiquette numérique il parut normal qu'elle soit utilisable en Algol cf. (III-3-2).

D'autres ont été ajoutés sans que ceux-ci deviennent pour autant plus puissants, ce sont ce que nous appelons des "concepts compilatoires" nés de l'évolution de l'analyse syntaxique et de la complexité des programmes.

Par exemple, COMMON en Fortran est un concept compilatoire, il n'apporte rien à la solution du problème mais il permet au compilateur d'optimiser la place occupée en mémoire en ne réservant qu'une zone commune à deux parties du programme.

Tous les concepts compilatoires sont donc inutiles au langage proprement dit mais facilitent la compilation, optimisent la place tenue en mémoire par les données du programme, mais on peut très bien supposer un langage sans ces concepts si l'ordinateur utilisé est suffisamment puissant.

A notre avis ces concepts compilatoires sont des phénomènes transitoires qui pourront être éliminés par les progrès de la conception des ordinateurs eux-mêmes, l'indépendance hardware-software restant toute relative.

Depuis peu, des langages ont été créés qui semblent vouloir apporter une généralité que les autres n'avaient pas, nous voulons parler de PL/1 et d'Algol 68.

Pour PL/1 on a un langage énorme qui est en fait l'union des
(cf IV-9) langages les plus utilisés : Fortran et Cobol et adjonction des concepts d'Algol reconnues valables. Le résultat obtenu est un langage relativement hétérogène - ce qui entraîne une compilation lente - avec en plus - mais pouvait-on l'éviter ? - une relative dépendance par rapport à la série IBM/360.

Il ne nous semble pas que la solution de tels "langages à tout faire" soit à retenir car toute extension future ne peut reposer sur ce qui existe déjà, cela entraîne donc des langages sans continuité et constamment périmés. Par exemple, le jour où l'on considèrera un display comme un périphérique standard, il faudra avoir dans les langages des ordres permettant la représentation de diverses figures. L'extension à PL/1

de ces nouveaux ordres ne pourra pas se faire à partir des ordres existants.

Par contre lorsqu'un langage a été bien défini, il est possible de l'utiliser pour l'étendre après de créer un langage plus puissant. L'exemple le plus probant est actuellement celui d'Algol W.

(cf IV-2) L'emploi d'Algol 60 avait montré les lacunes de ce langage, aussi, tout en gardant la même structure générale d'Algol 60, a-t-on rajouté des concepts nouveaux. Structures, références, traitement des chaînes, bits, un nombre de types de base plus grand.

Le langage obtenu n'est pas compatible avec Algol 60. Les programmes écrits en Algol 60 ne sont pas utilisables en Algol W et il n'y a pas de correspondance directe entre certains concepts des 2 langages, et en ce sens on n'a donc pas de véritable extension - mais on obtient un langage puissant bien défini et vite appris pour qui connaît Algol 60.

Dans l'exemple d'Algol W, on peut regretter sa dépendance - très faible - par rapport à IBM/360 (chaîne de bits sur 4 octets minimum), types correspondants, mots, double mots) mais en fait cela vient de ce que l'évolution du hardware a été dans le même sens que l'évolution du software.

(cf IV-13) La consécration actuelle de toutes ces recherches est le rapport Algol 68. Ici et peut-être pour la première fois, un langage a été pensé en fonction de sa conceptualisation et de sa définition formelle.

On peut peut-être regretter que dans un souci de trop grande généralisation, on dispose d'une syntaxe trop riche qui alourdit la programmation et qui alourdira dans une mesure non négligeable les temps de compilation. Sauf peut-être pour la frappe rapide sur console nous ne voyons pas l'intérêt d'avoir les deux écritures suivantes autorisées

```
if a > b then b else a fi
(a > b/b/a)
```

Mis à part ceci, on dispose d'un langage très conceptualisé possédant une puissance sémantique très grande : les concepts d'Algol 60 ont été généralisés au maximum, notions de types, d'opérateurs laissés au soin du programmeur et non plus limitées par leur nombre comme dans les langages classiques. Possibilité de traitement en parallèle; pour cette notion on peut penser que cela relève plus de l'analyse syntaxique que de la conceptualisation du langage lui-même. En effet en supposant la

généralisation du hardware permettant le traitement en parallèle plutôt qu'en séquentiel c'est l'analyse du programme par le compilateur qui permettra de dire si l'on peut traiter le problème en parallèle ou non. L'écriture : (x := 1, y := 2, z := 3) est une facilité d'écriture apportée par Algol 68. C'est ce que nous appelerons un concept compilatoire (cf. III-3-3).

Comme cela apparaît dans la définition d'Algol 68, il nous semble que la voie à utiliser est celle des langages extensibles. (Le singulier étant difficilement applicable car il y aura certainement toujours des grandes familles de langages commerciaux, scientifiques.)

A partir d'un noyau contenant la puissance sémantique désirée pour le genre de langage que l'on veut obtenir, on peut étendre ce langage à l'aide de concepts syntaxiques ou à l'aide de regroupement de concepts sémantiques.

Comme nous l'avons dit au début de cette introduction, celle-ci doit être considérée comme une série de réflexions que nous avons été amenés à faire en fonction de l'étude proposée.

L'un des buts de cette étude a été de voir de façon approfondie la méthode dite du Bootstrapping qui s'applique bien aux langage extensibles.

Un autre but a été d'essayer de développer uniquement du point de vue sémantique un langage noyau capable de décrire les différents concepts contenus dans les langages de programmation.

Enfin nous avons regroupé ces deux idées dans un essai d'application du Bootstrapping sur un sous-ensemble de Fortran déterminé à l'aide des règles définies dans notre essai de conceptualisation.

PREMIERE PARTIE

U N E M E T H O D E D ' A U T O C O M P I L A T I O N :
L E B O O T S T R A P P I N G

-:=-:=-:=-:=-:=-

I - 1. INTRODUCTION

Dans l'introduction nous avons vu que tout langage devait être accompagné, afin de rendre effective l'utilisation de ce langage, d'un compilateur propre à l'ordinateur sur lequel on désire utiliser ce langage.

Si dans cette introduction, nous avons surtout insisté sur l'importance que nous donnions à la conceptualisation des langages de programmation, il faut bien voir aussi, que l'écriture d'un compilateur est un travail important. Les méthodes de compilation sont étudiées depuis longtemps du point de vue syntaxique et nous ne nous y attacherons pas, nous désirons simplement parler ici d'une méthode particulière appelée "Boostrapping". C'est une méthode d'autocompilation qui s'insère dans notre optique des langages extensibles. Nous verrons dans cette partie les avantages certains de cette méthode. Celle-ci n'est pas nouvelle, nous avons simplement tenté de rendre plus vigoureux son emploi et plus simple la recherche du compilateur initial (cf : III^{e} partie).

I - 2. PRINCIPE GENERAL

Le principe général de l'autocompilation est qu'un compilateur est un programme et que comme tel, il peut être écrit dans un langage évolué et en particulier dans le langage même qu'il compile. Dans ce dernier cas il faut donc déjà posséder un compilateur de ce langage et cela peut sembler paradoxal. En fait on verra que l'on utilise soit un compilateur écrit pour une autre machine soit un compilateur pour un langage de puissance sémantique inférieure.

De ce principe très général, nous voyons que l'on peut donc envisager deux applications assez différentes dans leurs résultats :

- Avoir le compilateur d'un langage et désirer le transférer sur une autre machine où il compilera le même langage.
- Avoir le compilateur d'un langage et sur la même machine désirer un compilateur pour une extension de ce langage.

C'est en effet ces deux méthodes que l'on peut voir traitées dans les différentes publications faites sur le bootstrapping. Si elles ont été bien souvent séparées, leur principe reste le même et à

partir du paragraphe I-4. nous ne parlerons plus que d'une seule méthode de bootstrapping.

Nous nous bornerons à montrer ce qu'est le bootstrapping uniquement sur les langages évolués à propos des compilateurs, mais il faut bien spécifier que cette technique est utilisable à tous les niveaux de langage et qu'elle est utilisée pour l'écriture des assembleurs; ceux-ci sont en effet écrits en langage d'assemblage sauf pour un noyau qui, lui, est écrit en langage machine et qui possède une puissance sémantique suffisante pour décrire tout le langage d'assemblage de l'ordinateur considéré.

I - 3. DEFINITION DU BOOTSTRAPPING

I - 3.1. Notation

Nous écrirons :

$$C(X \rightarrow Y)_Z \ [/C1/C2]$$

Un compilateur C écrit dans le langage Z qui traduit le langage X en langage Y et (facultatif) qui dépend de C1 pour des performances à la compilation et de C2 pour ses performances à l'exécution. Le langage machine d'une machine M sera appelé M également.

I - 3.2. Passage d'un compilateur d'une machine sur une autre

Pour un langage donné, chaque ordinateur possède un compilateur pour ce langage. Il semble donc dommage de ne pas utiliser ces compilateurs existants quand on désire en écrire un pour un nouvel ordinateur.

Le principe du bootstrapping est ici très intéressant à appliquer car il permet, on verra plus précisément pourquoi dans les paragraphes suivants, d'écrire un compilateur pour un ordinateur qu'on ne possède pas encore.

Le principe d'application du bootstrapping à cette méthode peut être schématisé dans les étapes suivantes :

- a) on désire donc écrire un compilateur pour le langage L sur une machine M2.

La chaîne résultante étant en langage machine de M2 soit

- b) supposons que l'on possède une machine M1 possédant elle-même un compilateur du langage L, soit

$$C1(L' \rightarrow M1)_{M1}$$

- c) écrivons en langage L un compilateur qui transforme le langage L en langage machine M2, soit

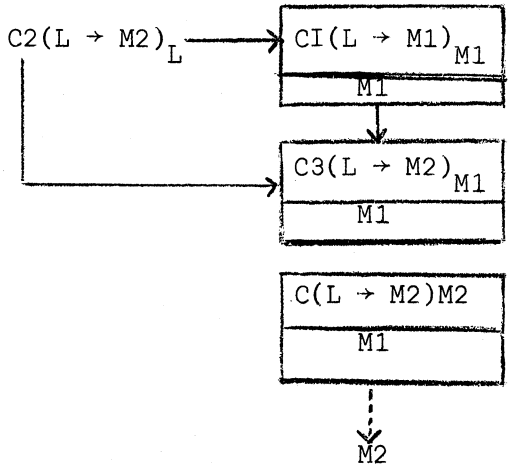
$$C2(L \rightarrow M2)_{L}$$

- d) donnons le à compiler à C_1, on obtient une chaîne en langage M1 capable de lire L et de générer du langage machine M2, on a donc obtenu le compilateur suivant :

$$C3(L \rightarrow M2)_{M1}$$

- e) redonnons C2 à compiler par C3, la chaîne générée obtenue est en langage machine M2 capable de lire L et de générer du langage machine M2. C'est le compilateur C cherché. Moyennant certaines opérations pratiques (modifications de code, de supports ...) on peut implanter ce compilateur sur l'ordinateur M2.

Les différentes étapes ci-dessus peuvent être schématisées comme suit :



I - 3.3. Application du bootstrapping à l'écriture d'un compilateur pour un nouveau langage

Cette seconde application, bien que procédant du même principe, a des résultats différents de la première méthode qui peut être considérée comme plus "artisanale". Le principe en est simple : pour écrire un compilateur d'un langage L en langage évolué (avec tous les avantages que cela comporte) il suffit de posséder un compilateur pour un sous-ensemble SL du langage L.

$$C_1(SL \rightarrow M)_M$$

Les concepts sémantiques de L devant être contenus dans SL afin de pouvoir écrire le compilateur traitant L (cf. III-1).

Le processus peut être résumé dans les étapes suivantes :

a) On désire obtenir le compilateur d'un langage L opérant sur une machine M. La chaîne résultante étant en langage machine M. Soit

$$C(L \rightarrow M)_M$$

ce compilateur.

b) Supposons que l'on possède, sur la machine considérée M, un compilateur pour un langage SL sous-ensemble de L. Soit

$$C1(SL \rightarrow M)_M$$

ce compilateur.

c) Supposons également que SL possède une puissance sémantique suffisante pour écrire le compilateur traitant L, alors on peut écrire en SL un compilateur de L pour M :

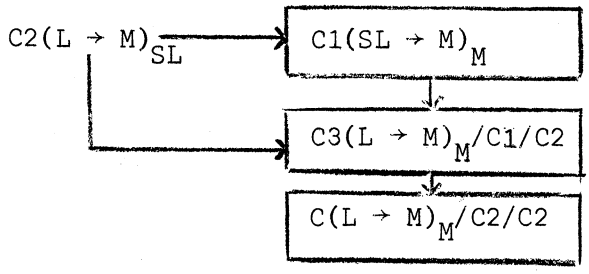
$$C2(L \rightarrow M)_{SL}$$

d) Donnons C2 à compiler à C1 (compilateur de SL), on obtient une chaîne capable de lire le langage L et de générer au langage M soit

$$C3(L \rightarrow M)_M$$

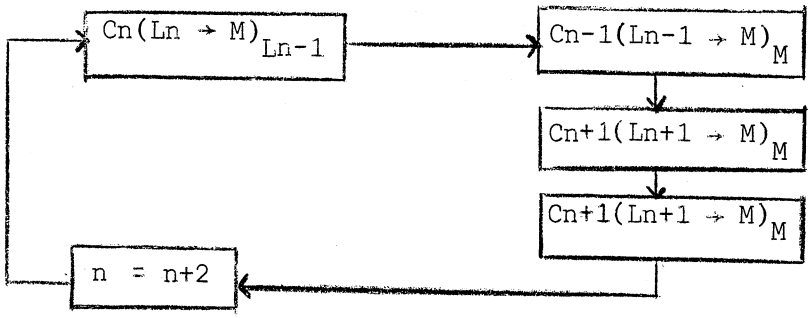
On peut penser que l'on a obtenu le compilateur désiré, on a en effet un compilateur pour le langage L, mais en fait C3 n'est qu'une étape et afin d'obtenir un compilateur C indépendant du compilateur source C1, il faut redonner C2 à compiler par C3.

Là aussi nous résumerons la méthode ci-dessus par le schéma suivant :



On remarque en comparant les deux schémas que les deux applications partent du même principe à savoir que la version compilée d'un compilateur écrit en langage évolué et compilant celui-ci donnera un compilateur indépendant du compilateur ayant compilé la version écrite en langage évolué.

Ces deux versions peuvent facilement être appliquées ensemble puisque le langage généré peut être changé à chaque instant. On a ainsi une troisième application possible, possédant un compilateur noyau sur une machine M1 pour un langage L1, désirer un compilateur pour une extension L2 de L1 sur M1, le désirer sur M2, étendre L2 en L3 ... On en déduit le schéma suivant :



I - 4. REMARQUES SUR LE BOOTSTRAPPING

La définition que nous venons de donner du bootstrapping pourrait faire croire à une grande simplicité de la méthode, en fait il ne faut pas oublier qu'à chaque boucle du troisième schéma il faut écrire un compilateur et qu'il est donc nécessaire d'optimiser ce nombre de boucles pour avoir une méthode justifiant son utilisation.

I - 4.1. Indépendance relative

Dans le paragraphe précédent, nous avons parlé de la dépendance relative à la compilation et à l'exécution des différents compilateurs, il peut être en effet important de se libérer des performances du compilateur d'origine.

Nous parlerons de la qualité d'un compilateur en parlant des temps de compilation et du temps d'exécution de la chaîne générée à la compilation.

En reprenant les notations du deuxième schéma, on a :

étape d : En faisant compiler C2 par C1, on obtient une chaîne C3 qui dépend de C1 aussi bien pour le temps d'obtention de cette chaîne que de sa qualité à l'exécution qui dépend du code que génère C1.

étape c : Dans la compilation de C2 par C3, la chaîne C obtenue dépend :
Pour le temps de compilation : de la qualité du code de C3, donc dépend de C1.
Pour la qualité du code généré, uniquement de la qualité du code que génère C3 donc de C2.

Cette compilation ne se produit en principe qu'une seule fois, l'utilisation du compilateur C doit par contre donner des résultats indépendants du compilateur C1.

Donc, dans la compilation d'un programme P par C, le temps d'obtention de la chaîne codée dépend du code de C donc de C2. La qualité du code généré - qui entraîne une performance plus ou moins grande au niveau exécution - dépend de la qualité du code que génère C, donc de C2.

Cette indépendance vis-à-vis du compilateur source peut être intéressante en ce sens que l'on obtient un compilateur dont les qualités ne proviennent que de la méthode de compilation utilisée pour ce travail et que l'on peut pour cela utiliser un compilateur dont les qualités soient médiocres.

I - 4.2. Simplification des processus d'écriture

La première remarque que nous ferons est que le bootstrapping apporte une facilité évidente d'écriture puisqu'il est plus facile d'écrire en langage évolué qu'en langage machine, cette écriture en langage clair entraîne une lecture facilitée d'où des avantages sur la documentation, la maintenance et la correction des erreurs. D'autre part étant donné que le temps d'exécution du code généré par le compilateur se doit d'être minimum, l'optimisation de ce compilateur est nécessaire, et la méthode du bootstrapping simplifie et accélère cette optimisation.

1 - Parce que l'on se réfère mieux dans un programme écrit en langage évolué.

2 - Parce que l'avantage est double, l'optimisation se faisant à la fois sur le programme généré et sur le compilateur lui-même.

Par exemple, supposons qu'à chaque apparition de l'opérateur α le compilateur C2 du schéma général génère 4 instructions alors que deux suffiraient pour obtenir le même résultat à l'exécution. Dans C2 remplaçons le générateur de α qui générant quatre instructions par un autre qui n'en génère plus que deux.

La compilation de C2 par C3 fournit donc une chaîne C plus courte car α apparaît certainement dans la rédaction de C2. De plus à la compilation d'un programme par C, le code généré P sera également plus court. On a donc un gain de temps sur la compilation du fait que C soit moins longue et un temps d'exécution plus petit du fait que P soit moins long.

Signalons enfin un avantage relatif à ce que nous avons appelé la première application du bootstrapping. Le compilateur que l'on désire obtenir pour la machine M2 peut être fait dans sa plus grande partie sur la machine MI dont on possède déjà le compilateur pour le langage spécifié. Cela permet l'écriture d'un compilateur sans posséder la machine receptrice d'où gain de temps important dans l'élaboration du software de cet ordinateur.

I - 4.3. Restrictions

La difficulté première est que le bootstrapping ne s'applique pas aux entrées-sorties, du fait - évident - que celles-ci dépendent de l'ordinateur sur lequel est implanté le compilateur. Il ne peut être question de décrire en langage évolué des concepts qui ne peuvent être interprétés au sens de concepts sémantiques puisqu'ils dépendent exclusivement du hardware. On sera donc amené à prévoir des procédures en code de la machine résultante pour compiler les entrées-sorties.

Une autre difficulté est que si la différence de conception des deux machines est trop grande, il faut paramétrer les différentes données et variables de façon qu'à l'implémentation sur la nouvelle machine, on puisse remplacer certaines valeurs pour optimiser la place tenue en mémoire.

Il est bien évident cependant que ces difficultés sont mineures et n'enlèvent rien à l'efficacité du bootstrapping.

I - 5 CONCLUSION

Nous venons de voir que le bootstrapping apportait de nombreux avantages dans l'écriture des compilateurs ceci étant essentiellement dû au fait que l'on écrit en langage évolué. Cependant il nous a semblé

que dans le cas de l'extension d'un langage, cette méthode était plus satisfaisante sur le plan théorique. Nous n'avons dans ce cas pas seulement une méthode d'écriture des compilateurs mais aussi une théorie qui s'ajoute à celle des méthodes de compilation. L'extension peut être faite à partir d'un noyau commun et l'on peut supposer que l'on aura un jour un compilateur de base à partir duquel on pourra bootstrapper tous les langages de programmation. C'est dans cette optique que nous avons poursuivi notre étude en tentant la création d'un noyau conceptuel commun à tous les langages. Auparavant, nous allons donner dans la deuxième partie quelques exemples de bootstrapping.

DEUXIEME PARTIE

E X E M P L E S D E R E A L I S A T I O N

-:=-:=-:=-:=-:=-

II - 1. INTRODUCTION

Les divers exemples dont nous allons parler ont été pris parmi ceux d'une liste déjà longue de réalisations de compilateurs par la technique du bootstrapping.

Nous avons porté notre choix sur ceux qui nous ont semblé les plus significatifs.

La technique du bootstrapping n'étant pas nouvelle certains de ces exemples datent de plusieurs années.

Dans certains exemples, nous serons amenés à énoncer certaines généralités qui devront être considérées comme valables pour tous les exemples.

II - 2. LE "LANGAGE" PL 360
N. Wirth 1966-1970

(cf IV-13)
IV-15)
IV-16)

PL 360 est ce que l'on pourrait appeler un langage semi-évolué. Bien que possédant une apparence de langage évolué, il est dépendant du 360/IBM. En effet, les concepts sont ceux du langage d'assemblage que l'on déclare comme étant des fonctions du langage. De plus, on dispose de divers concepts syntaxiques ou compilatoires, structure de bloc, instructions itératives et de choix. En tant que langage,PL 360 est très intéressant puisque tout en possédant la puissance d'un langage d'assemblage, il offre toutes les facilités syntaxiques et sémantiques d'un langage évolué.

On dispose pour traiter ce langage d'un compilateur, qui étant donné la configuration très dépendante du langage vis-à-vis de la machine, donne des temps de compilation excellents et un code très performant.

La taille de ce compilateur est très restreinte : 38 K octets environ pour le module. Une caractéristique intéressante de ce compilateur c'est qu'il est lui-même écrit en PL 360.

A l'origine du projet PL 360, si la configuration générale de la machine IBM/360 était connue, il n'y en avait pas encore de disponible. Il fut donc décidé d'utiliser un ordinateur Burroughs B 5500 alors en service.

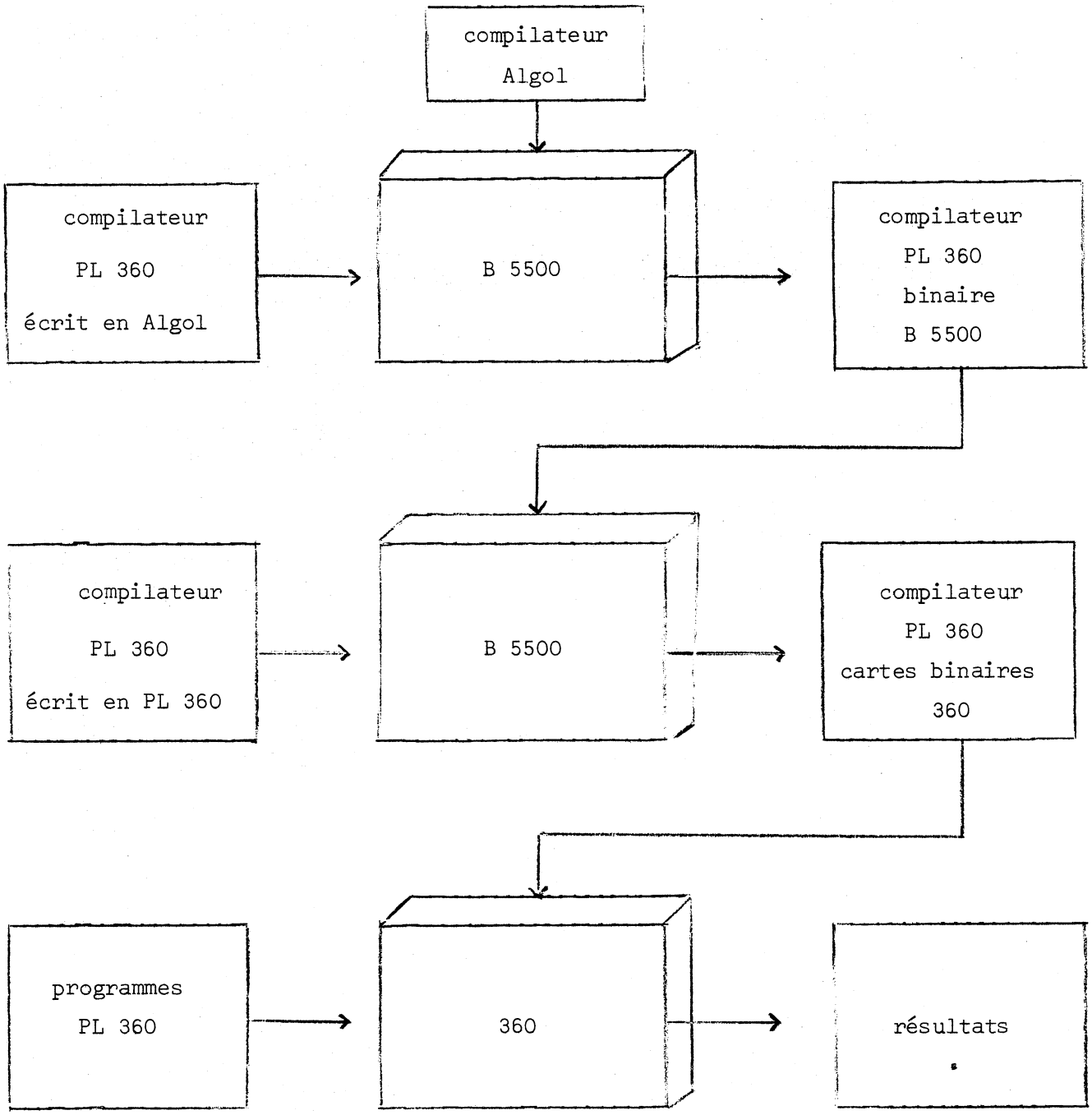
Un compilateur PL 360 fut écrit en Algol pour B 5500 qui compilait PL 360 tel qu'il était défini à l'époque. Le compilateur fut réécrit dans son propre langage. Par l'intermédiaire d'un chargeur et d'un superviseur le programme recompilé sur B 5500 donnait un module immédiatement utilisable sur 360.

Les avantages de cette méthode sont multiples.

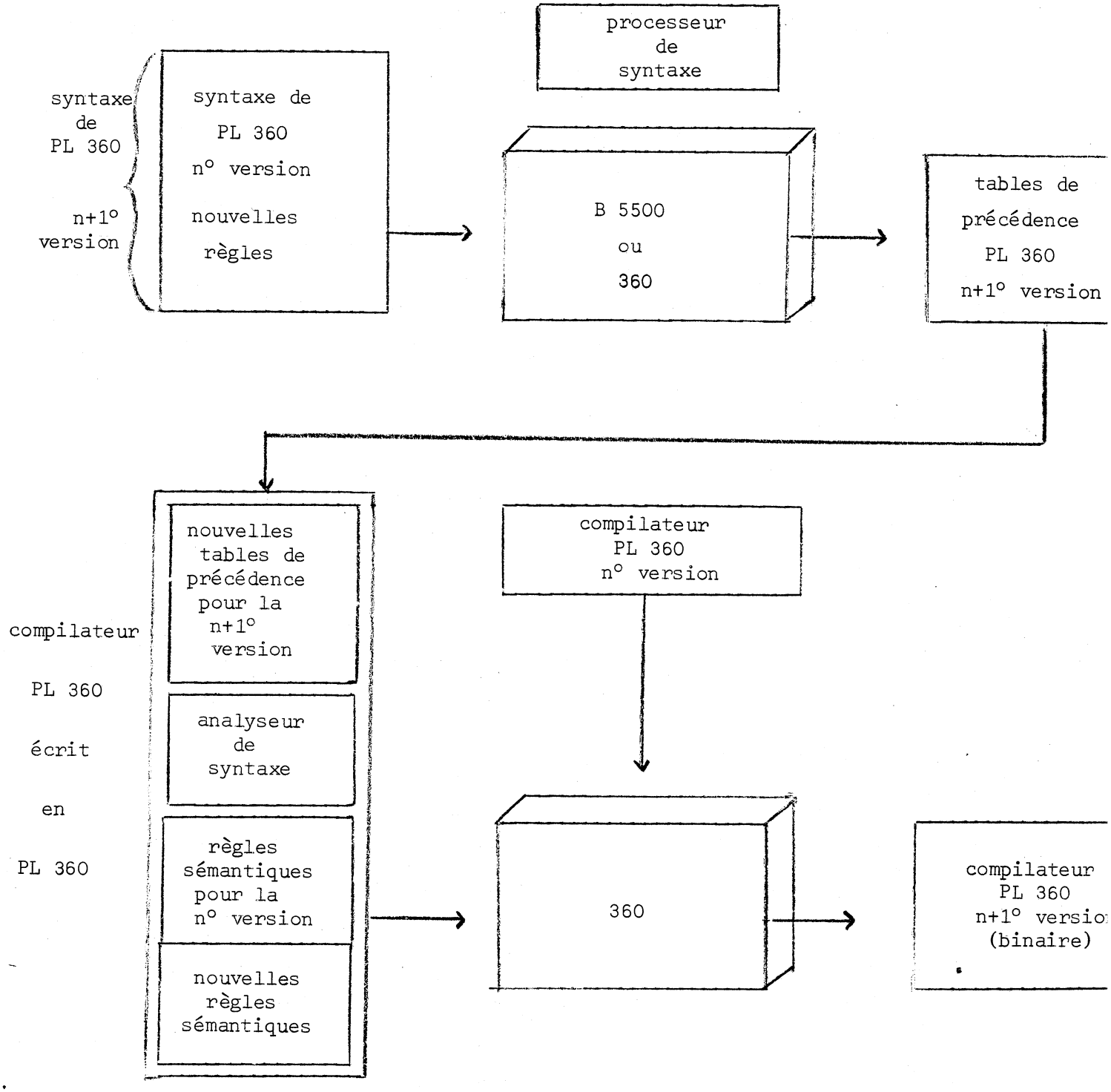
Tout d'abord le fait de compiler l'algorithme définissant le langage lui-même est un très bon test pour voir si le langage est bien défini, c'est ainsi qu'au cours de cette réécriture certains concepts qui semblaient désirables furent ajoutés alors que d'autres qui se révélaient d'emploi peu intéressant ou conduisaient à des erreurs de conception furent supprimés.

La combinaison de la méthode du bootstrapping et de l'organisation systématique du compilateur au moyen de règles syntaxiques strictes, a été appliquée avec succès.

Le processus pour ajouter de nouvelles constructions au langage consistant simplement à ajouter de nouvelles règles d'interprétation, ces règles devant bien sûr être décrites en termes de facilités définies précedemment. En fait, le bootstrapping est aisément applicable car le sous-ensemble PL 360 (n° version) contient déjà toutes les facilités de l'ordinateur 360 et que les nouvelles règles que l'on peut définir sont toujours exprimables à partir des concepts de la version précédente.



Bootstrapping pour la version initiale du langage avec changement de machine.



Bootstrapping applicable à un changement
de définition du langage.

II - 3. ECRITURE D'UN COMPILATEUR ALGOL EN ALGOL (M. Chemla 1964-1965)

Nous avons choisi ce second exemple car il illustre parfaitement le passage d'un compilateur d'une machine à une autre.

Ici, le but fixé était l'obtention d'un compilateur Algol pour le GE-635, celui-ci n'étant pas opérationnel à l'époque du démarrage du projet. L'équipe disposait de deux ordinateurs : un Gamma 60 et un IBM 7094 et, bien sûr, des compilateurs Algol sur ces deux machines.

Nous ne reviendrons pas sur les avantages qu'offre le Bootstrapping, nous les avons décrits dans le chapitre précédent et ils s'appliquent évidemment ici. Nous ne retiendrons que les problèmes spécifiques à cet exemple.

Tout d'abord le langage :

Bien sûr, il est séduisant d'écrire un compilateur Algol en Algol, Algol étant un sous-ensemble de lui-même, la technique de bootstrapping s'applique parfaitement. Reste à savoir si Algol est un langage de software. Le réalisateur déplore une perte de place de 10 % environ du fait qu'Algol ne comporte pas de variable binaire.

Rien que par ce fait, il est évident qu'Algol 60 ne se prête pas à l'écriture de software.

Or, il est impossible de trouver un sous-ensemble d'Algol 60 qui contienne les variables binaires.

On peut donc se demander si la technique du bootstrapping est applicable à tous les langages de programmation. Il nous semble que plus le langage est puissant, plus le sous-ensemble que l'on peut déterminer est performant et plus il se rapproche d'un bon langage de software.

A propos des machines, le fait d'utiliser deux ordinateurs 'source', s'il entraîne accélère le processus, entraîne certaines limites puisque les problèmes d'implémentation interviennent, Algol 60 est implémenté de deux façons différentes sur le Gamma 60 et IBM 7094, ceci étant dû aux différences de conception des deux ordinateurs.

Donc le sous-ensemble utilisé pour décrire Algol 60 n'est pas le même selon que l'on utilise l'un ou l'autre des 2 ordinateurs. Il faut donc prendre un sous-ensemble commun aux deux, ce qui limite encore la puissance de ce sous-ensemble.

Si l'on veut utiliser plusieurs ordinateurs sources il faut que ces ordinateurs soient à peu près de même puissance pour qu'ils acceptent le même langage source.

En effet, si l'un des ordinateurs ne pouvait accepter un concept important du langage du fait de sa conception même (ordinateur ne travaillant pas en virgule flottante par exemple) alors il faudrait supprimer ce concept pour décrire le langage lui-même, ce qui alourdirait considérablement cette description.

Enfin cette dualité des ordinateurs sources entraîne une phase d'initialisation des constantes qui ne peuvent apparaître que sous forme symbolique puisque la représentation interne varie.

Malgré toutes ces restrictions il semble que les avantages l'emportent et que la méthode du bootstrapping soit applicable avec succès à tout langage de programmation, chacun d'eux étant sous-ensemble de lui-même est capable de se décrire lui-même.

II - 4. COMPILATEUR FORTRAN H POUR 65/360 (Lowry & Medlock 1969)

(cf I-6)

Cet exemple a été choisi car il montre ce que peut apporter un bootstrapping en série.

Le problème à résoudre est l'écriture du compilateur de Fortran H pour IBM/360.

Le compilateur est écrit en Fortran et bootstrappé 3 fois.

La première fois consiste à bootstrapper le compilateur pour passer de IBM/7094 à IBM/360. On écrit donc en Fortran un compilateur Fortran H qui compilé par le compilateur Fortran de la 7094 génère un code opérationnel sur 360.

De l'avis même des auteurs cela fut un travail ardu. En effet, nous retombons sur les problèmes de l'exemple précédent.

(cf IV-8)

Fortran est-il un langage de software apte à décrire un compilateur générant du code pour une machine relativement compliquée (gestion des registres, adressage).

Le second bootstrapping fut fait pour optimiser la taille du compilateur, celle-ci passant de 550 K à 400 K. On a vu que si l'on savait optimiser le code généré, alors on réduisait aussi la taille du compilateur.

Enfin comme l'on désirait ajouter les facilités de traitement de chaîne de bits et des données organisées en fichiers (référence à des structures), le compilateur fut réécrit encore une fois et optimisé.

Le temps de compilation était alors réduit de 35 % et la taille réduite de moitié permettant au compilateur de travailler avec 256 K seulement.

On voit donc dans cet exemple que certains utilisateurs n'hésitent pas à faire toute une série de bootstrapping afin d'obtenir le meilleur compilateur possible.

Le bootstrapping est en effet la seule méthode qui optimise de façon relativement automatique un compilateur.

TROISIEME PARTIE

E S S A I D E F O N D E M E N T C O N C E P T U E L
D E S
L A N G A G E S D E P R O G R A M M A T I O N

-:=-:=-:=-:=-:=-

III - 1. RAPPORT AVEC LE BOOTSTRAPPING

Nous avons vu dans la première partie que pour écrire le compilateur d'un langage L, il fallait si l'on désirait appliquer le bootstrapping, posséder le compilateur d'un sous-ensemble de ce langage L; ce sous-ensemble formant un langage suffisamment puissant pour décrire le langage L lui-même.

Cela signifie que les concepts sémantiques contenus dans le langage L doivent être exprimables à partir de concepts plus simples, l'exemple de la boucle POUR en Algol 60 étant des plus significatifs :

'POUR' I := 1 'PAS' 1 'JUSQUA' 5 'DO' instruction ;

peut s'écrire sans utiliser la boucle POUR :

```
      I := 1
E1 :  <instruction>;
      I := I + 1;
      'IF' I <= 5 'GOTO' E1;
```

or, nous l'avons vu dans l'introduction, il est bien rare qu'un langage L1 existant soit sous-ensemble d'un langage nouvellement créé L2. Il faut prendre une partie du langage L1 en éliminant les concepts incompatibles avec ceux de L2.

Si l'on ne possède pas déjà le compilateur du sous-ensemble nécessaire, il faut l'écrire, ce qui crée plusieurs problèmes.

1) Etre capable de déterminer un sous-ensemble du langage.

2) Que le choix de ce sous-ensemble soit tel qu'il facilite l'écriture des deux compilateurs.

Ce choix du sous-ensemble est évidemment de la plus grande importance. La taille du premier compilateur dépend du nombre de concepts choisi dans le sous-ensemble. Plus ce nombre est petit, plus le langage ainsi défini est simple et plus le travail d'écriture du compilateur est simplifié, mais ceci entraîne que les concepts du langage lui-même seront plus difficilement exprimables à partir du sous-ensemble

et la description du compilateur complet en sera alourdie : ce à quoi il faudra arriver et ceci est un but lointain - est la définition d'un langage de base (ou langage noyau) dont on possèdera une fois pour toutes le compilateur - quitte à faire une opération de bootstrapping pour passer d'une machine sur une autre.

Les compilateurs de langages plus complets étant obtenus automatiquement par la définition des concepts à l'aide des concepts du noyau. Le langage sera donc un langage extensible et la production des compilateurs sera automatisée. Nous étendrons cette discussion à la fin du chapitre.

III - 2. DEFINITIONS

- Soit A un ensemble de base dont les éléments seront appelés primitifs.

- Soit $\mathfrak{A}$ l'ensemble A × A, on appelera $\mathfrak{A}$ un ensemble d'actions.

- On appelera langage conceptuel une famille de parties de $\mathfrak{A}$.

- On appelera concept de base d'un langage conceptuel un sous-ensemble réduit à un élément de $\mathfrak{A}$.

- Un langage noyau est défini comme l'ensemble des concepts de base.

- On appelera concept un sous-ensemble de $\mathfrak{A}$.

- On dira qu'un concept C2 contient un concept C1 si

$$C1 \subset C2$$

III - 3. SEMANTIQUE DES LANGAGES DE PROGRAMMATION

Nous avons déjà vu que lors de l'élaboration des langages de programmation, on s'était d'abord attaché à produire des langages qui répondaient aux besoins de l'époque et qu'afin de faciliter l'écriture des compilateurs et des programmes, on s'était efforcé de créér des langages à la syntaxe très rigoureuse comme Algol par exemple. Les principales recherches étaient effectuées sur l'analyse syntaxique des langages et l'on avait un peu laissé de côté la conceptualisation des langages.

Actuellement, on voit au contraire naître des langages qui possèdent une structure sémantique bien définie et plus rigoureuse, par exemple Algol 68.

Dans un langage on rencontre en général trois sortes de concepts :

III - 3.1. Les concepts sémantiques

Ce sont des concepts qui exprimables en langage évolué se traduisent, au moment de l'exécution du programme par le résultat espéré dans l'écriture de ce programme.

Ces concepts sont exprimables à partir de concepts de base qui sont donc au point de vue exécution sur un ordinateur donné, les concepts sémantiques de cet ordinateur.

III - 3.2. Les concepts syntaxiques

Ces concepts n'apportent rien de plus au point de vue exécution du programme, mais simplement une facilité d'écriture du langage évolué.

Par exemple en Fortran, on peut se dispenser de déclarer entières les variables dont l'identificateur commence par I, J, K, L, M, N. Il est bien évident que cet additif au concept de déclaration n'apporte rien de plus à la puissance du langage; de plus, dans ce cas particulier, la facilité apportée à l'écriture du programme est minime et plus dûe à l'habitude que l'on a de cet emploi qu'à une vraie réduction de l'effort d'écriture du programme; la compilation est d'autre part alourdie par la présence de tests supplémentaires.

Un autre exemple tout à fait contraire au précédent est celui des étiquettes numériques d'Algol. On est en présence ici d'une fausse généralisation de la syntaxe. Un identificateur est écrit à l'aide d'un ensemble de signes alors que les nombres sont écrits à l'aide d'un autre ensemble de signes dont la représentation graphique est identique à celle d'une partie du premier ensemble. Afin d'éviter une compilation trop lourde, on interdit en général que le premier signe d'un identificateur ait la même représentation graphique - donc le même code interne à la machine - qu'un chiffre.

III - 3.3. Les concepts compilatoires

Ces concepts sont ajoutés au langage afin de faciliter la tâche du compilateur. Nous sommes en effet actuellement en présence d'ordinateurs aux possibilités limitées et il est nécessaire de palier à ces défauts par une amélioration des capacités des compilateurs, autrement dit à optimiser les programmes.

On peut donc espérer que dans l'avenir, avec l'avènement de machines plus performantes, on verra ces concepts disparaître des langages de programmation.

Comme exemple de concept compilatoire on peut citer le COMMON dans Fortran qui sert à optimiser la place tenue en mémoire, mais qui n'apporte rien à la puissance sémantique du langage qui la contient.

Un autre exemple est la déclaration des variables ceci dans une certaine mesure, les déclarations étant en rapport avec la structure de bloc pour une question de place en mémoire (gestion dynamique des tableaux par exemple).

III - 4. CONCEPTUALISATION. ESSAI

Nous avons défini un langage conceptuel comme étant une famille de parties de $\mathcal{C} = A \times A$.

Nous allons faire une énumération de A et $\mathcal{A}$. Cette énumération nous donnera en plus des primitifs (A) les concepts de base nécessaires à l'élaboration d'un langage de programmation.

III- 4.1. Ensemble des primitifs : A

Considérons le partitionnement suivant de A

$$A = \mathcal{M} \cup I \cup V \cup \boxed{P}$$

$\mathcal{M}$: Ensemble, en général fini, de cases

I : Ensemble de noms pouvant être formé à partir d'un ensemble de caractères.

V : Ensemble de contenus des cases $\in \mathcal{M}$.

$\boxed{P}$: Ensemble de cases distinctes de celles $\in \mathcal{M}$

$\mathcal{M}$: est le support physique. Nous l'avons implicitement considéré comme une suite logique d'éléments de taille quelconque - mais tous égaux pour une mémoire donnée - c'est en fait la mémoire centrale d'un ordinateur ainsi que l'on a l'habitude de la considérer.

I : c'est l'ensemble des mots par lesquels on peut nommer un élément de la mémoire. C'est l'identificateur au sens langage évolué, c'est le terme symbolique au sens assembleur. Les restrictions que l'on retrouve dans presque tous les langages évolués font que l'on n'utilise qu'un sous-ensemble de I. Ainsi en Algol W par exemple :

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

La définition d'identificateur telle qu'elle est faite en Algol W est une restriction langage puisqu'on ne prend qu'un sous-ensemble de I. De plus le fait qu'un identificateur ne peut avoir plus de

256 caractères est une restriction d'implémentation dûe aux caractéristiques machines.

V : c'est l'ensemble des contenus des éléments de mémoire. Le contenu particulier d'un élément de mémoire est en général appelé la valeur de cet élément.
V est constitué de sous-ensembles où les valeurs ont le même type.
On considérera des valeurs arithmétiques (pouvant se décomposer en arithmétique à virgule fixe, à virgule flottante, en arithmétique décimale ...), des valeurs de type booléen (pouvant prendre des valeurs VRAI ou FAUX), des valeurs de type binaire, caractères, octales, hexadécimales, des constantes adresses, les ordres du programme, des valeurs physiques, notées [Ei] , c'est-à-dire représentables sur des éléments périphériques, etc ... .
Les valeurs plus évoluées (format, valeurs multiples, routines ...) ne sont pas des éléments de V mais sont soit des sous-ensembles de V soit plus généralement des sous-ensembles de A.

[P] : ce sont des éléments physiques qui prenant les valeurs dans des éléments mémoires, les transforment sous une forme physique de telle sorte qu'elles puissent être stockées dans des emplacements n'appartenant pas à $\mathcal{M}$, ou, au contraire, à partir de ces emplacements les amener en mémoire.
Ce sont donc les périphériques d'entrées/sorties ou encore les mémoires auxiliaires.

Considérons maintenant une énumération de A (qui n'est peut-être que partielle).

Emplacement :

$$Si\ m \in \mathcal{M}\ alors\ m = \bigcup_{i=1}^{i=k} \xi_i\ et\ (\xi_k \cap \xi_j = \emptyset\ si\ k \neq j)$$

C'est le plus petit élément accessible de la mémoire m. Dans les langages de programmation plus anciens, cette notion n'existe pas. Elle a toujours été implicite du fait de la rigidité hardware des machines à mots.

On peut cependant imaginer une configuration dont l'accès soit variable selon les besoins ou les opérateurs appliqués. Dans les machines à octets actuelles, à quelques exceptions près pour des traitements numériques, il est impossible de travailler sur des éléments dont la taille soit inférieure à 8 positions binaires. On peut très bien imaginer une configuration hardware permettant de travailler au niveau du bit. En fait l'emplacement est l'élément de mémoire adressable.

Identificateur :

Il existe une application injective de $\mathfrak{M}' \subset \mathfrak{M}$ *dans I.*

A tout élément de mémoire peut être affecté un nom quelconque, c'est ce principe qui fut adopté dans les langages d'assemblage pour éviter les calculs d'adresse. A une translation près, la correspondance entre nom symbolique et emplacement est biunivoque. Le principe a été pris à l'envers pour les langages évolués, on prend un identificateur que l'on considère comme un emplacement virtuel, c'est au niveau compilation qu'est faite l'affectation à un emplacement physique avec calcul d'adresse correspondant. A un même identificateur correspondent un ou plusieurs emplacements, un seul étant "actif" à un instant donné ceci à cause de la structure de bloc principalement, un même identificateur pouvant indiquer dans des blocs différents deux éléments distincts.

Dans les langages de programmation l'identificateur désigne, indifféremment une variable, un tableau, une étiquette, une procédure ... . Dans tous ces cas, l'identificateur désigne un seul emplacement, les autres étant déduits soit par la configuration de la machine soit par le type de ce qu'il désigne.

Information :

Il est possible de définir une application de $\mathfrak{N}' \subset \mathfrak{N}$ *dans V.*

Un emplacement contient en général un enregistrement magnétique ou autre mis par un moyen mécanique quelconque. Cet enregistrement correspond - dans l'état actuel de la technique - à une chaîne binaire qui sera interprétée de diverses façons selon le type affecté à cette information.

Primaire de longueur ℓ :

$$P_i^{\ell} = \bigcup_{j=i}^{i+\ell} \xi_j$$

Remarque :

Sauf quand cela est nécessaire on indiquera un primaire par P_i. Le primaire est l'association de 1 ou plusieurs emplacements c'est en fait le primaire qui est à considérer comme le concept de variable dans les langages de programmation. Si dans un langage il est possible de définir la longueur d'une variable, celle-ci sera composée de plusieurs emplacements. Si cette notion n'existait pas dans les langages plus anciens (en Algol 60 il y a correspondance entre mot machine et variable) elle existe sous une forme relativement figée depuis la création des machines à octets.

Short integer, integer, real, long real, sont des types permis en PL 360.

En Algol 68 la longueur de la variable est quelconque , il y a simplement restriction au moment de l'implémentation du langage à quelques cas particuliers.

Si le hardware progresse de telle sorte que par exemple l'emplacement soit réduit à la position binaire, on pourra traiter des variables de longueur quelconque.

Les primaires possèdent une propriété :
l'accessibilité. Cette notion souvent laissée aux soins du superviseur peut être introduite dans un langage en vue du traitement en parallèle ou de la microprogrammation. L'inverse de l'accessibilité est la protection c'est-à-dire que l'on ne peut accéder à l'information que le primaire considéré contient.

On peut avoir plusieurs types de protection.

On peut avoir une protection mémoire, c'est-à-dire que telle partie du programme est inaccessible à partir d'une autre partie cela est vrai au niveau des données pour la structure de bloc, on peut prévoir un traitement parallèle de 2 morceaux de programme mutuellement inaccessibles, alors on peut y voir de la programmation en parallèle.

Dans les langages de programmation classique cette notion d'accessibilité mémoire n'est pas laissée à la portée du programmeur mais traitée par le compilateur, l'accessibilité aux données dans la structure de bloc étant peu utilisée ou ignorée.

```
begin integer A;
      begin integer A;
      end
end
```

Le primaire de nom A défini dans le premier bloc est inaccessible dans le bloc interne puisque remplacé par le 2^{e} primaire de même nom A, mais le premier est conservé, le cas n'est pas le même dans l'exemple suivant :

```
begin integer A ;
      begin integer B ;
      end
end
```

B n'est pas accessible dans le bloc externe car il n'a pas d'existence dans ce bloc.

Un autre genre de protection est la conversion de type.

Si on fait dans Algol W

```
begin reference R; integer I;
      I := R;
```

on aura une erreur lors de l'exécution de l'opération d'affectation car Algol W ne prévoit pas la transformation de type : Référence → entier.

On dira que R n'est pas accessible.

Dans PL/1 toutes les transformations de types étant possibles, il n'y a pas de protection par conversion, le programmeur décide seul de l'accessibilité aux primaires de son programme.

Enregistrement :

Soit un primaire P_i^ℓ, son contenu est une suite concaténée d'informations.

$$E_i^\ell = q_i \cdot q_{i+1} \cdot q_{i+2} \cdot \cdots \cdot q_{i+\ell-1}$$

$$E_i^\ell \in V$$

Remarque : On notera E_i un enregistrement quand la longueur ne sera pas nécessaire.

L'enregistrement est la valeur contenue dans le primaire, la concaténation étant un mécanisme purement hardware.

Dans toutes les machines actuelles, le primaire est constitué par un emplacement dont l'adresse donne la correspondance avec l'identificateur utilisé - et ceux qui le suivent, ceci à cause du traitement séquentiel des ordinateurs actuels. On peut supposer qu'il existera des ordinateurs travaillant avec des mémoires parallèles, mais cela ne changera rien au principe, seul le mécanisme de concaténation changera.

Secondaire :

$$S_{i_1}^L = P_{i_1}^\ell \ \alpha \ P_{i_2}^\ell \ \alpha \ldots \alpha \ P_{i_L}^\ell$$

Un secondaire est un ensemble de L primaires de longueur ℓ liés entre eux par une liaison de type α tel qu'elle définisse le type du secondaire.

On a un primaire privilégié dont l'identificateur est celui du secondaire.

Il existe un ordre de rangement pour un secondaire qui est actuellement naturel, c'est l'ordre séquentiel, il est implicite dans les langages de programmation et dans le fonctionnement des ordinateurs.

La longueur du secondaire permet de savoir combien de primaires sont contenus dans le secondaire.

III - 4.2. Ensemble $\mathcal{A}$ des actions

Dans l'énumération de cet ensemble nous allons examiner successivement - en les rapprochant des idées existantes - les concepts de base qui sont nécessaires en totalité ou en partie à la construction des langages de programmation.

Spécification :

$i \in I$ spécifie ξ_j si on a

$$i = f(j)$$

f est appelé fonction adresse
si $i = f(j) = f(j')$ alors $j = j'$

L'application inverse est multivoque (concept d'équivalence).

Remarque : Dans certains langages possédant la structure de blocs, il est possible de donner le même nom à plusieurs variables; ceci est un concept compilatoire, le compilateur traite cet identificateur commun de telle sorte que cela revient à avoir des noms différents (name liste).

Type d'enregistrement

Il existe un ensemble fini de fonctions t_i, appelées fonctions types, telles que E_j étant l'information d'un primaire P_j^{ℓ} on a :

$$V_k = t_i(E_j) \qquad V_k \in V$$

La fonction t_i est le mode de décodage de l'enregistrement considéré.

Toute spécification de primaire entraîne en général le désir de s'intéresser à son contenu. Selon l'emploi que l'on désire en faire il peut être intéressant que ce contenu soit décodifiable de façons différentes. La représentation d'une valeur pourra par exemple être définie au moment de la déclaration en indiquant le type choisi. C'est le cas

d'Algol 68 où l'on peut déclarer des modes. En fait, on pourrait se contenter d'un type binaire. Les autres types étant définis au moyen de modes analysant la chaîne binaire d'une façon ou d'une autre.

```
[(Type entier longueur L) A :
     I = 2; A = 0;
          Si Bit(1) = 0 alors SIGN = + sinon
                                    begin SIGN = -;
                                          A = COMP(A)
                                    end;
          Calcul : Si bit(I)  = 0 alors A = A + 2^(L - I);
                   I = I + 1;
                   Si I    = L + 1 alors aller à CALCUL];
```

La syntaxe de ce "langage" est imaginaire ... . La fonction complément qui redonne la représentation positive est supposée écrite.

On pourrait donc imaginer un seul type de base qui définirait tous les autres types. On remarque que l'emplacement - encore que l'on pourrait le simuler au moyen des opérateurs logiques - est le bit, or nous n'en sommes pas encore là dans le hardware machine. De plus, les ordinateurs ont pour la plupart des circuits permettant de traiter de façons différentes les chaînes binaires, disposant ainsi de plusieurs types de base.

Logiquement ce sont ces types de base que l'on retrouve dans les langages évolués car ils permettent de traiter la plupart des problèmes. Les autres types s'en déduisent.

On a donc :

deux types pour le traitement arithmétique

entier
réel

1 type pour le traitement de décisions logiques

Booléen

1 type pour le traitement des chaînes caractères

Caractère

1 type pour référencer à d'autres données du problème

Référence

A remarquer que l'on pourrait remplacer le type logique par le type binaire. Les opérations booléennes étant incluses dans les opérations logiques. Il faudrait encore une fois que l'emplacement soit le bit.

De plus,on peut considérer un "type multiple" un primaire ayant une longueur quelconque on peut admettre que l'enregistrement contenu qu'il contient est la concaténation de plusieurs valeurs chacune ayant un type bien défini.

C'est donc le principe de record où chaque record contient différentes valeurs de types différents par exemple en Algol W :

record R(integer I; string C, reference(R) SUITE)

chaque primaire R est constitué d'une valeur entière, d'une chaîne de 16 caractères (valeur par défaut en Algol W) et d'une référence à une autre variable de la classe R et que l'on appelle SUITE).

Ce qui est à l'intérieur de la déclaration du record R doit être considérée comme une suite d'indications permettant de mettre des valeurs du type indiqué, les noms indiqués permettant de localiser les différentes valeurs qui sont dans le primaire.

Type de secondaire

On a vu qu'un secondaire était défini comme suit :

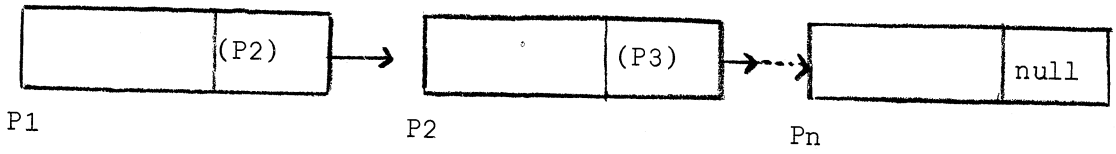
$$S^{L}_{i_1} = P^{\ell}_{i_1} \alpha\ P^{\ell}_{i_2} \alpha \ldots \alpha\ P^{\ell}_{i_L}$$

Il existe un ensemble fini de fonctions α, *appelées type de secondaire telles que*

$$i_{j+1} = \alpha(i_j) \quad j = 1, 2, \ldots, L-1$$

C'est le moyen d'indiquer comment les différents primaires sont liés entre eux de façon que l'on puisse toujours accéder à l'un quelconque des primaires.

Le type le plus général est celui défini en partie par LISP, chaque primaire est déclaré multiple, une partie de ce primaire est une référence au primaire suivant.



La partie gauche de chaque primaire est aussi compliquée qu'on le désire et peut contenir d'autres références à d'autres primaires, ceci afin d'obtenir des structures d'arbres ... .

Une simplification majeure de ce concept de secondaire est le concept de tableau. Les ordinateurs travaillent en séquentiel, c'est une référence interne à la machine (le compteur ordinal) qui permet de sélectionner le primaire suivant celui que l'on a déterminé.

En fait le concept tableau a été simplifié encore par le fait que les différents primaires ne peuvent contenir qu'une seule valeur.

integer array A[1 : 100]; en Algol 60

Alors que l'on pourrait aisément avoir dans un langage hypothétique :

array A(1 : 100) (integer, string(6), real);

indiquant ici que l'on désire avoir 100 primaires contenant chacun - et dans cet ordre évidemment - un entier, une chaîne de 6 caractères, un réel.

La longueur de chaque primaire est alors une question d'implantation (représentation interne de valeurs, alignement ...).

Donc comme concept de base nous prendrons celui de primaire pointant sur un autre primaire.

Néanmoins, nous considérerons comme à part le concept de tableau étant donné qu'il se trouve dans presque tous les langages et qu'étant donné la structure hardware des ordinateurs, il y a une référence implicite du primaire suivant du fait de la séquentialisation.

Mouvement

Il existe une fonction Mouvement m telle que P_i et P_j étant deux primaires on a

$$P_j = m(P_i) \Rightarrow E_j = E_i$$

C'est l'instruction d'affectation classique.

Le contenu d'un primaire P_i vient remplacer le contenu d'un primaire P_j. Le contenu de P_j étant modifié seul P_j doit être accessible.

Le mouvement travaille sur les enregistrements. En gardant la terminologie IBM/360 on doit aussi bien le considérer comme un Load que comme un MVC, c'est-à-dire que le mouvement travaille quelque soit le type des primaires.

Il doit y avoir compatibilité de type pour les affectations si on ne veut pas être conduit à une erreur. Le degré de compatibilité des opérandes étant à notre avis un concept compilatoire, le nombre de sous-programmes permettant les conversions étant toujours modifiable dans un compilateur.

Chargement

Il existe une fonction chargement c telle que P_i et P_j étant deux primaires on a

$$P_j = c(P_i) \Rightarrow E_j = i$$

Le chargement est un mouvement permettant d'identifier un primaire particulier.

Il correspond au Load adress des langages assembleurs. Tel que nous le définissons il existe assez peu dans les langages évolués. Peu d'entre eux traitent des adresses elles-mêmes.

Par exemple en Algol W on peut montrer la différence entre Mouvement et Chargement

```
begin
    integer I, J; string S;
    reference(ENREG) REF;
    record ENREG(integer K, string S);
    I := J; comment ceci est un mouvement;
    REF := ENREG(I, S);
    comment ceci est un chargement;
end.
```

Dans REF on a véritablement l'adresse machine de l'enregistrement (au sens Algol W) créé.

De plus la notion de chargement permet d'expliquer la notion habituelle du allera. En effet si après chaque action, on effectue une action de chargement dans un primaire privilégié de l'adresse du primaire contenant sous la forme d'un ordre l'action suivante à effectuer, le moniteur saura alors exécuter une suite d'actions.

Comparaison

Il existe une fonction de comparaison R telle que P_k *étant un primaire privilégié,* P_i *et* P_j *deux primaires quelconques on a*

$$P_j = R(P_i) \Rightarrow (E_k = vrai \text{ si } E_i = E_j) \text{ ou } (E_k = faux \text{ si } E_i < E_j)$$

L'action de comparaison permet donc, comme son nom l'indique de comparer les enregistrements contenus dans deux primaires P_i et P_j.

Le primaire P_k peut être un primaire privilégié servant toujours pour ce type d'action. En ce sens il est équivalent au code condition de l'IBM/360.

Nous avons choisi pour définir la comparaison des deux opérateurs de relation = et <, ces deux opérateurs permettant de retrouver toutes les autres relations. On aurait pu en choisir d'autres. En fait en vue d'optimiser le noyau minimum, les opérateurs de relation servant très souvent, on verra que l'on a intérêt à prendre tous les opérateurs de relation classique

Canal

Il existe un ensemble fini de fonctions K telles que étant donné un primaire P_i *et* $\boxed{P_j}$ *un élément de* $\boxed{P}$ *on a*

$$\boxed{E_j} = K(E_j)$$

La relation canal permet donc d'utiliser les périphériques comme éléments d'entrées ou de sorties d'enregistrement.

La relation canal peut, en fait, être décomposée en diverses relations élémentaires permettant différents transferts dans un sens ou dans un autre de données.

En fait cet ensemble est actuellement limité :

1) par le nombre des périphériques

2) par le nombre d'ordres permettant de traiter différemment une même donnée.

Du point de vue langage ces ordres sont limités actuellement dans la plupart des langages à des ordres d'écriture ou de lecture sur des supports magnétiques (disques, bandes, tambour ...) ou des supports physiques (cartes, imprimante ...).

On peut supposer que d'autres ordres pourront être créés, spécifiques à certains périphériques.

On a déjà dans certains langages des ordres graphiques qui ne s'adressent qu'à des périphériques spécifiques (display - traceur). Même si actuellement à cause du hardware, on peut enregistrer ces données sur des supports magnétiques.

On peut avoir par exemple

```
Cercle(centre, rayon);
Droite(A, B);
```

qui au niveau langage et pour le programmeur sont destinés à être visualisés et non écrits.

Nous n'aimons pas beaucoup les notions adoptées par Algol 68 (notions de canaux, ouverture, fermeture de fichiers ...) qui sont dans une certaine mesure restrictives et surtout obligent le programmeur à tenir compte de certaines contingences système, ce qu'il n'aime pas en général.

Opérateurs

Il existe un ensemble de fonctions o telles que P_i et P_j étant deux primaires quelconques on a

$$E_j = o(E_i)$$

Là encore la notion de relation opérateur peut se décomposer en relations différentes selon l'opérateur considéré. On a en fait un ensemble d'opérateurs.

Le choix de cet ensemble dans le calcul du sous-ensemble minimum d'un langage est très variable.

Disons que les opérateurs arithmétiques étant les plus utilisés sont à prendre dans presque tous les cas.

\+ - * / flottant ou entier

Les opérateurs Booléens

$\wedge$ $\neg$ les autres s'en déduisant

Les opérateurs "binaires"

Décalage

Nous avons [illegible] ce que nous entendions par opération [illegible] La notion d'opérateurs est presque intuitive (peut-être parce que très utilisée dans tous les langages antérieurs) c'est celle-ci que nous utilisons.

La notion de déclaration d'opérateur dans Algol 68 est très intéressante mais elle revient en fait - sous une forme plus souple - à une déclaration de procédure.

par exemple :

```
op min = (real a, b) real : (a > b/b/a);
x := y min P_i/2;
```

revient à écrire en Algol W par exemple :

```
real procedure MIN (real value A, B);
if A > B then B else A;
real X, Y;
X := MIN(Y, PI/2);
```

Test

Il existe une fonction test T telle que étant donnés deux primaires P_i et P_j quelconques et deux primaires privilégiés P_k et P_ℓ on a

$$P_j = T(P_i) \Longrightarrow (E_\ell = i \text{ si } E_k = vrai) \text{ ou } (E_\ell = j \text{ si } E_k = faux)$$

E_ℓ est généralement appelé le compteur ordinal
E_k est le code condition résultant d'une comparaison.

Le test est un changement variable selon le résultat d'une relation. Cela revient donc à :

```
if <Relation> then goto E1 else goto E2;
```

En effet en plus du chargement on voit que l'on considère l'action a_i c'est-à-dire celle dont l'adresse correspond au chargement.

Le goto inconditionnel revient à aller à la même adresse pour tous les cas de relations.

Néanmoins toujours pour optimiser le noyau minimum décomposer le concept test en 2 concepts

le goto inconditionnel
le goto conditionnel

ceci afin de faciliter l'écriture du programme source. Il est en effet plus simple d'écrire

goto E1

plutôt que

if <relation quelconque> then goto E1 else goto E1

III - 4.3. Addition de concepts compilatoires

Afin de faciliter l'écriture d'un programme dans le langage défini par le sous-ensemble, il peut être intéressant de rajouter certains concepts à puissance sémantique nulle.

Sans ces concepts, la taille du programme écrit grandi très vite et l'on tombe sur des problèmes de capacité mémoire ou autres.

A notre avis 2 concepts sont pratiquement obligatoires ce sont :

1) La structure de bloc

Nous ne considérerons ici que le côté compilatoire de la structure de bloc, c'est-à-dire celui lié aux déclarations et l'optimisation de la place tenue en mémoire. Les autres côtés positifs,-modularité,indépendance d'écriture de blocs disjoints etc - qui eux ne sont pas uniquement des concepts compilatoires sont ignorés dans ce problème.

2) Le sous-programme

Ce concept n'est pas seulement un concept compilatoire, en ce sens qu'il peut se déduire des concepts de base, mais il faut avant tout le considérer comme tel car il apporte une facilité à l'écriture et un gain de temps à la compilation malgré une perte de temps à l'exécution.

En prenant pour support la syntaxe de Fortran, on peut relier le concept de sous-programme aux concepts de base suivants :

```
subroutine SP
        :
        :
        :
        :
return
        :
        :
        :
        :
call SP
```

Le call entraîne les opérations de base suivantes :

1) chargement dans un primaire privilégié de l'adresse du primaire contenant l'action suivante à effectuer.

2) chargement du compteur ordinal de l'adresse du primaire correspondant à l'identificateur SP.

3) accessibilité aux primaires SP et à ceux qui le suivent et exécution des diverses actions qu'ils contiennent.

4) chargement lors de la rencontre de l'action return, du compteur ordinal de l'adresse contenue dans le primaire privilégié du 1).

5) accessibilité aux primaires correspondant au chargement ci-dessus et exécution de l'action correspondante.

III - 4.4. Relation avec les concepts classiques

Dans tous les langages de programmation, il existe des concepts implicites ou admis une fois pour toute, cela souvent à cause de la configuration des ordinateurs ou des habitudes.

Un programme est une suite d'instructions que l'on peut considérer comme des primaires et qui permettent d'appliquer des actions sur d'autres primaires appelés données. La différence entre instructions et

données est nulle a priori et ce n'est qu'une question d'interprétation qui permet de les distinguer. Disons seulement que l'information que l'on désire considérer comme instruction sera donnée à un superviseur - qui est l'ordinateur lui-même au dernier niveau - qui appliquera l'action indiquée sur le primaire spécifié.

Il est donc nécessaire que le superviseur possède des primaires privilégiés qui contiendront des informations lui permettant d'exécuter le programme. Le primaire privilégié principal, est celui qui est généralement appelé compteur ordinal, ce qui lui permet d'accéder au primaire correspondant à l'information contenue dans ce compteur et généralement appelée adresse. Il faut donc à chaque fois que l'on veut accéder à une instruction faire un chargement du compteur ordinal avec l'adresse du primaire contenant cette instruction.

Ceci est automatisé par la séquentialisation des mémoires, le primaire à considérer étant celui qui suit celui que l'on vient de considérer. Il peut cependant être nécessaire de ne pas considérer le primaire suivant, c'est le cas de ce que l'on appelle les ruptures de séquence qui sont une accessibilité au primaire spécifié, les étiquettes étant pour le programme ce que les identificateurs sont pour les données.

La notion de compteur ordinal est implicite dans la plupart des langages de programmation du fait de la séquentialisation de la mémoire, c'est pourquoi le concept de l'indexation est très courant. Ce n'est en fait qu'un chargement calculé en ajoutant à l'adresse du primaire d'ordre 0 - qui correspond au primaire spécifié - le numéro d'ordre du primaire dans le secondaire.

De plus, il existe certains concepts fondamentaux dont nous ne parlons pas dans notre définition de sous-ensemble et qui semblent fondamentaux à la plupart des informaticiens, ce sont :

Les procédures :

Il y a génération du corps de procédure à chaque appel. Ceci peut être facilement remplacé par un sous-programme.

```
integer V, W;
Procedure P(integer A, B);
begin integer X;
   X := A * B + X;
   A := X - A;
end;
 :
 P(V, W);
 :
 P(Z, Y);
```

peut être remplacé par :

```
Subroutine P
X = A * B + X
A = X - A
return
integer V, W, Z, Y

A = V
B = W
call = P

A = Z
B = Y
call P
```

L'emploi du sous-programme est moins souple et moins élégant que celui de la procédure mais on arrive au même résultat.

Récursivité

Considérons la procédure Algol W suivante :

```
begin integer procédure SOM(integer value X, Y);
    begin integer TEMPSOM;
      comment fait la somme des entiers de X à Y : X < Y;
if X = Y then TEMPSOM := X else
   TEMPSOM := X + SOM(X + 1, Y);
,Tempsom
```

pourra être écrite en ne considérant pas le concept de la récursivité mais en utilisant les concepts de procédure simple et d'instruction POUR :

```
   begin
integer procedure SOM(integer value X, Y);
   begin integer tempsom
        tempsom := X;
     for I := 1 until Y - X do
begin X := X + 1
        tempsom = tempsom + X;
end;
  tempsom
end;
```

On pourrait multiplier les exemples.

III - 5. EXEMPLES DE REALISATIONS IDENTIQUES

(cf 3-1) - Le premier que nous indiquerons est celui du Professeur Bauer qui à partir d'un système de bouées flottantes ou coulées tente de retrouver les concepts des langages de programmation en conservant un certain rapport (mais est-il possible de l'éviter ?) avec le hardware, ses bouées pouvant, de la même façon que notre concept "primaire" être considérées comme des mots machine.

(cf 3-2) - Le second est un exemple de réalisation plus complet. Il a été réalisé à Computer Associates par : MM. Cheatham, Fischer et Jorrand.

Après un essai de définition sémantique des langages, il a été défini syntaxiquement un langage - ELF. (Extensible Language Facility), qui est un langage noyau permettant la description de langages étendus à partir de ce noyau, c'est-à-dire, en fait, l'écriture de leur compilateur.

On peut, peut-être, regretter que la longueur modifiable des variables n'ait pas été retenue comme concept de base, car cela interdit l'emploi de ELF pour des langages de la classe d'Algol 68 où la longueur variable des éléments est autorisée.

Nous pensons que cette obtention de langage noyau extensible est très intéressante et peut apporter des avantages pour la compilation, la communication des programmes et la possibilité de création d'un langage universel pouvant être étendu selon les besoins spécifiques de chaque installation.

III - 6. RECHERCHE DU SOUS-ENSEMBLE OPTIMUM D'UN LANGAGE

Le premier travail que l'on doit faire est de dresser les listes des concepts contenus dans le langage à traiter.

On aura une liste de concepts sémantiques, une autre de concepts compilatoires, une autre de concepts syntaxiques et une de concepts hardware. Cette décomposition est délicate car c'est à partir de ces listes que sera construit le sous-langage utilisé pour décrire le langage complet.

Les listes étant obtenues, on commencera à construire le sous-langage à partir de la liste des concepts sémantiques uniquement.

Supposons connu le noyau conceptuel des langages, on aura les opérations suivantes :

1) Un concept du langage est équivalent à un concept du noyau, ce concept est pris pour faire partie du sous-langage.

2) A un concept du langage correspond plusieurs concepts du noyau - c'est le cas le plus fréquent - on les prend pour constituer le sous-langage.

3) Un concept du langage n'est pas constitué par l'union de concept de base, dans ce cas, le concept est mis dans SL mais de plus rajouté au noyau - c'est le cas le moins fréquent du moins nous l'espérons -.

4) Il y a des concepts du noyau qui ne correspondent à aucun concept du langage, ceux-ci ne sont bien sûr pas considérés. La liste des concepts étant obtenus on a alors le langage noyau minimum. Pour des raisons que nous avons indiquées dans le chapitre précédent, il est intéressant d'ajouter des concepts compilatoires de façon à obtenir un langage noyau optimum capable de décrire dans les meilleures conditions la totalité du langage.

III - 7. CONCLUSION

Nous avons tenté dans ce chapitre de définir un noyau conceptuel des langages de programmation. Nous espérons être arrivés près du langage noyau "idéal". Tout théorème devant posséder sa démonstration nous avons essayé cette méthode de détection de noyau optimum d'un langage pour écrire son compilateur, sur le langage Fortran. Nous allons décrire cette expérience dans le chapitre qui suit.

QUATRIEME PARTIE

E S S A I D ' A P P L I C A T I O N

A

F O R T R A N

-:=-:=-:=-:=-

IV - 1. INTRODUCTION

Cet essai comportera plusieurs phases, qui finalement seront les phases normales d'un bootstrapping dans le cas de l'écriture d'un compilateur quand on ne possède pas le compilateur de base.

- 1) Etude du langage Fortran d'un point de vue uniquement sémantique.
  Cela consiste à dresser la liste de tous les concepts Fortran, apparents ou non contenus dans le langage.

- 2) Création du noyau optimum pour le langage considéré, en l'occurence Fortran.
  Pour cela on compare les concepts sémantiques du langage avec ceux du noyau défini dans la troisième partie. La méthode de comparaison ayant été énoncée en III.6.

- 3) Ce langage noyau étant déterminé, écrire un compilateur pour celui-ci. Il faut pour cela choisir la méthode de compilation et le langage dans lequel sera écrit le compilateur de base.

- 4) Ecrire dans le langage que compile le compilateur de base défini précedemment, un compilateur pour le langage Fortran complet.

IV - 2. DETERMINATION DES CONCEPTS FORTRAN

(cf IV-4) Nous avons pris comme rapport de base le livre de Mr. Dreyfus : "Fortran IV".

IV - 2.1. Déclarations

Le langage Fortran accepte de traiter les variables dans les types suivants :

- Entier

  Avec déclaration implicite pour les identificateurs dont la première lettre est I, J, K, L, M, N.

- Réel

  Avec déclaration implicite pour les identificateurs dont la première lettre n'est pas I, J, K, L, M, ou N.

- Double précision

- Complexe

- Logique

- Alphanumérique.

IV - 2.2. Déclaration de variables indicées

Notion de représentation matricielle à n indices.
Deux représentations syntaxiques par exemple :

```
Integer TAB
Dimension TAB(5, 6)
```

est équivalent à

```
Integer TAB(5, 6)
```

IV - 2.3. Opérateurs

Fortran reconnaît les opérateurs suivants :

Arithmétiques : + - × / ↑
Logiques et Booléens : = ≠ < ≤ ≥ > ¬ ∧ ∨

IV - 2.4. Fonctions

Ce sont toutes les fonctions standards soit mathématiques (sin, cos ...) soit de servitude (conversion de type par exemple).

IV - 2.5. Formules arithmétiques

On affecte à une variable la valeur résultant du calcul d'une expression arithmétique.

IV - 2.6. Identités logiques

On affecte à une variable logique le résultat d'une expression logique.

IV - 2.7. Ruptures de séquence

Elles sont de deux sortes :

- Inconditionnelles

  . goto étiquette
  . goto d'affectation par l'intermédiaire de assign to
  . goto calculé par l'intermédiaire de l'affectation d'une valeur à la variable qui régit le goto
  . ordres de terminaison :
  stop, end, pause.

- Conditionnelles

  if arithmétique de la forme
  if (expression ou variable), N1, N2, N3 par exemple.
  if logique de la forme
  if (expression logique) instruction exécutable selon le résultat logique (vrai ou faux) de l'expression on exécute l'instruction ou on poursuit en séquence. Cette instruction est en général un goto étiquette qui permet d'exécuter une autre partie du programme.

do sert à répéter un nombre de fois déterminé la séquence de programme qui la suit immédiatement.

IV - 2.9. Les entrées-sorties

Nous les énoncerons simplement :

read, punch, print, rewind, end file, backspace

liés avec les notions de liste de format.

IV - 2.10. Fonctions et sous-programme

Ce sont des séquences de programme qui reviennent souvent dans le programme et qui sont utilisées de trois façons différentes :

- Fonction formule. On déclare une fonction ne comportant qu'une seule formule et qui peut être ensuite utilisée avec les paramètres déviés.

- Les fonctions qui s'emploient de la même façon que les formules mathématiques (IV.2.4) mais qui doivent être définies dans le programme.

- Les sous-programmes qui sont des fonctions généralement admettant plusieurs résultats et ne pouvant se trouver dans une expression. La définition se fait avec l'ordre subroutine, la fin du sous-programme étant indiquée par return l'appel au moment de l'exécution se faisant par call.

IV - 2.11. Il existe dans Fortran IV divers concepts compilatoires régissant l'implantation des variables en mémoire :

- Equivalence : même adresse en mémoire pour deux identificateurs différents.

- Common : une même variable peut être utilisée conjointement par plusieurs programmes.

IV - 2.12. Création d'information au niveau du programme symbolique

- Data : permet une affectation globale des éléments d'un tableau.

- Block Data : sous-programme permettant de fournir des tables de constantes pouvant être partagées (au moyen de common) par plusieurs programmes.

IV - 3. DETERMINATION DU SOUS-ENSEMBLE DE FORTRAN A L'AIDE DES REGLES ENONCEES EN III - 4.

IV - 3.1. Concepts sémantiques

Déclarations

Correspond à l'action type d'enregistrement.
On a donc les types de bases suivants :

- entier
- réel
- logique

Variables indicées

Ce concept est inclu dans type de secondaire puisque seul le concept de tableau existe dans Fortran.

Seul le concept de tableau à 1 dimension est à retenir. La pluralité des dimensions se simulant aisément.

$$A(I, J) \equiv A(I \times N + J)$$

Opérateurs

L'ensemble des opérateurs est relativement limité en Fortran; il correspond pratiquement à l'ensemble 0 du chapitre 3.

On aura donc + - * / opérateurs arithmétiques

$\wedge$ $\neg$ opérateurs booléens

Le concept d'opérateurs de relations est inclu dans le concept de Relation où l'on a

$< \leq > \geq = \neq$ opérateurs de relation

Fonctions, Formules arithmétiques

Ce sont des opérateurs pouvant être déduits à partir de l'ensemble O.

Les fonctions ne font donc pas partie du sous-ensemble.

Identités logiques

C'est le concept même de Relation.

Ruptures de séquence

Le seul concept contenant la rupture de séquence est celui de test

- inconditionnelles

On retiendra le goto étiquette dans le sous-ensemble le seul qui corresponde à :

if A = A then goto E_1 else goto E_1.

- conditionnelles

Là encore le choix parmi les diverses possibilités offertes par Fortran est évident et on retiendra dans le sous-ensemble la forme :

if <relation> goto <étiquette>

où l'instruction exécutable si la relation est vérifiée est seulement un goto.

Les boucles de programme

Rien dans le sous-ensemble ne permet de les retenir.

Entrées-sorties

Du fait de la vocation du sous-ensemble il n'est pas nécessaire d'avoir des entrées-sorties évoluées.

Il suffit que l'on puisse lire le compilateur, que l'on puisse imprimer le texte lu, d'éventuels messages d'erreur et enfin 3^{e} concept de sortie : un ordre Dump permettant de connaître le contenu de certaines mémoires afin de tester le compilateur.

Si donc les ordres de lecture et d'écriture peuvent être considérés comme des concepts de base à partir desquels on pourra construire des entrées-sorties plus évoluées (format), il est nécessaire de considérer un concept compilatoire permettant de sortir un état intermédiaire des résultats.

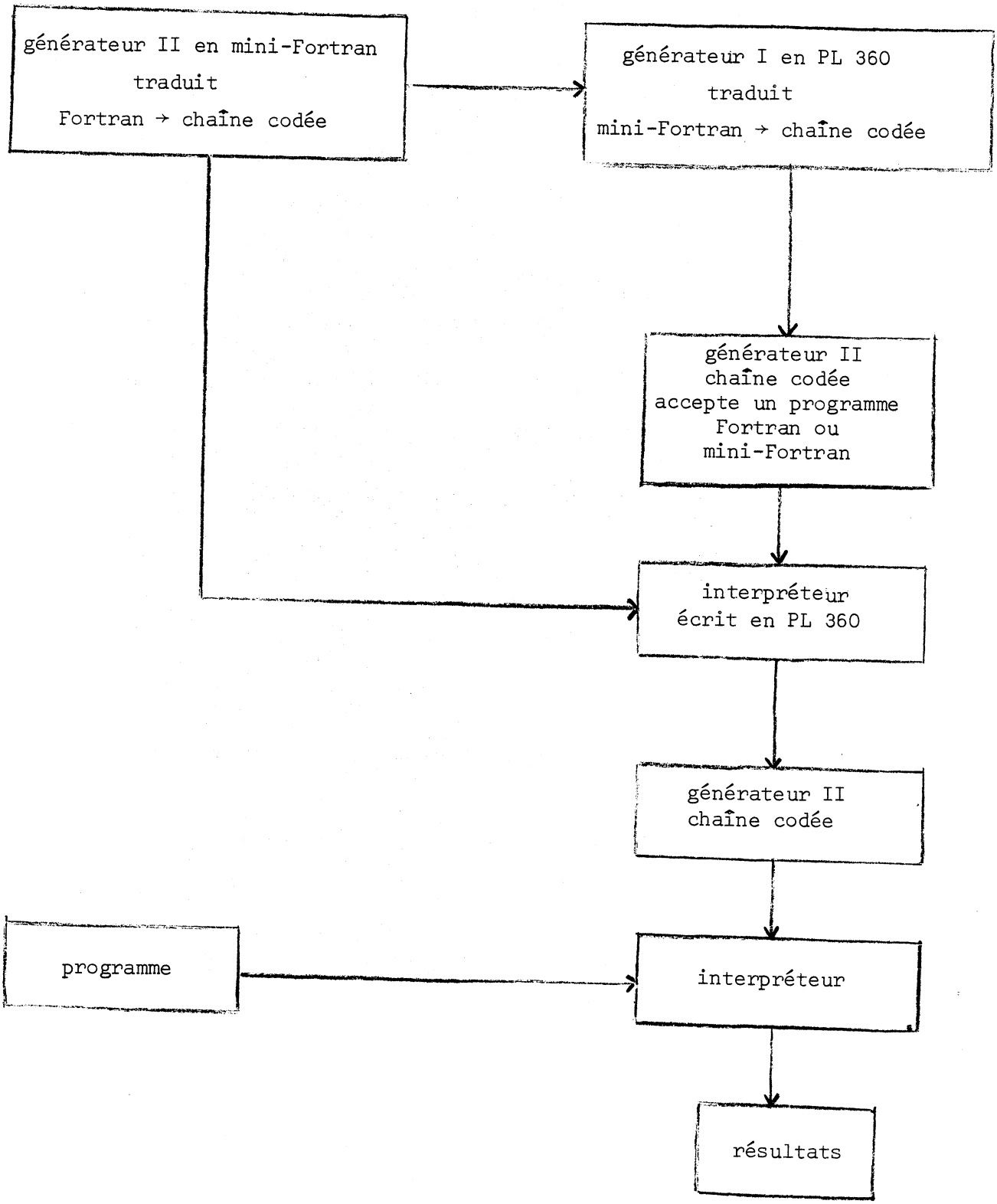
IV - 4. ECRITURE DU COMPILATEUR POUR LE SOUS-ENSEMBLE DE FORTRAN

IV - 4.1. Méthode de compilation

Notre compilateur n'étant pas à vocation opérationnelle mais ayant simplement une valeur de démonstration pour la théorie que nous avons développée au chapitre III, nous ne nous sommes pas souciés outre mesure d'écrire un compilateur très optimisé ni très performant.

La méthode choisie est celle de Génération-Interprétation.

(cf 2-10) L'avantage pour cette méthode en 2 passages c'est que comme il nous fallait écrire deux compilateurs, il nous suffisait d'écrire un seul interpréteur et deux générateurs.


générateur II en mini-Fortran
traduit
Fortran → chaîne codée
générateur I en PL 360
traduit
mini-Fortran → chaîne codée
générateur II
chaîne codée
accepte un programme
Fortran ou
mini-Fortran
interpréteur
écrit en PL 360
générateur II
chaîne codée
programme
interpréteur
résultats

Pour écrire cette première partie de l'exemple possédant divers traducteurs et n'ayant pas à nous soucier d'efficacité, nous aurions pu choisir n'importe quel langage d'assemblage ou évolué. Nous avons choisi la solution intermédiaire d'un langage s'écrivant facilement mais donnant un code plus performant que le même programme écrit en langage d'assemblage.

IV - 4.2. Ecriture du compilateur

Il y a peu de chose à dire sur cette écriture du premier compilateur.

Nous avons employé des algorithmes de compilation très simples et très connus.

Nous avons cependant dû faire attention à la gestion des registres, de la même façon que si nous avions écrit le programme en assembleur.

Nous avons dû paginer la chaîne générée par le générateur, cette chaîne, du fait de sa taille, ne pouvant être en mémoire.

Nous n'avons utilisé qu'une gestion statique de la mémoire. Si ceci simplifie la tâche au générateur 1 cela nous a limité dans le nombre de variables pour l'écriture du second générateur, nous obligeant à écrire des constantes peu explicites à la place d'identificateurs mnémoniques.

IV - 5. ECRITURE DU SECOND GENERATEUR

Nous avons du écrire en mini-Fortran un programme compilant Fortran.

Quelles sont donc les réponses que doit apporter l'écriture du 2° générateur ?

1) Y a-t-il avantage à écrire le compilateur dans le langage mini-Fortran, plutôt que dans un autre langage ?

2) Mini-Fortran est-il bien défini ? est-il suffisant pour décrire Fortran ?

3) Est-il suffisant pour décrire Fortran de façon efficace ?

I) Ecrire un programme, c'est utiliser la syntaxe de ce langage.

La définition du sous-langage, telle que nous l'avons décrite, entraîne l'adoption de la syntaxe du langage origine.

Dans notre exemple, nous avons donc été tributaires de la syntaxe de Fortran, ce qui a été gênant par rapport à PL 360, mais ceci ne peut se généraliser, cette remarque étant spécifique de chaque exemple.

Du point de vue concepts de programmation, nous estimons être gagnants dans l'écriture propre du second générateur. Nous donnons deux exemples écrits dans les 3 langages

Assembleur
PL 360
Mini-Fortran

Le second est évidemment catastrophique pour mini-Fortran. Celui-ci ne travaillant ni au niveau caractère, ni au niveau octet du 360. Mais une fois la subroutine CARSUI écrite, il n'y a plus tellement d'exemples ou mini-Fortran soit inférieur à ce point. Naturellement cela exclut l'emploi de mini-Fortran dans la résolution de problèmes ne travaillant que sur des chaînes, mais ceci vient de ce que Fortran IV n'est pas fait non plus pour cela.

1) <u>En Assembleur</u>

```
*
* CALCUL D'UNE CONSTANTE ENTIERE
*
CONSTINT DS    OH
         STM   R10, R15, CSTSAVE
         SR    R10, R10
         SR    R11, R11
         SR    R12, R12
CSTBLAN  CLI   0(R3), C' ' élimination des blancs
         BNE   CSTCAL
         LA    R15, CARSUI
         BALR  R14, R15
CSTCAL   IC    R12, 0(R3)  schéma de Horner
         S     R12, = F'240'  sur la suite de caractères
         M     R10, = F'10'  constituant la constante
         AR    R11, R12
```

```
            LA    R15, CARSUI
            BALR  R14, R15
            CLI   0(R3), C'0'
            BNL   CSTCAL
            LM    R10, R12, CSTSAVE  le résultat est dans R11
            BR    R14
CSTSAVE     DS    6F
```

2) En PL 360

```
        procédure CONSTANTE(R5);
        comment calculer d'une constante entière;

        begin.
            integer array 3 CSTSAVE;
            STM(R10, R12, CSTSAVE);
            R10 := R11 := R12 := 0;
        CSTBLAN : CLI(" ", B3); if = then begin CARSUI; goto CSTBLAN; end;
        CSTCAL : IC(R12, B3);
        comment schéma de Horner sur la suite de caractères constituant
        la constante;
        R11 := R11 * 10 + R12 - 240;
        CARSUI
        CLI("0", B3);
        if > = then goto CSTCAL;
        comment le résultat est dans R11;
        LM(R10, R12, CSTSAVE);
    end;
```

3) En mini-Fortran

```
        subroutine CSTINT

C   Calcul d'une constante entière
        CONSTINT = 0
01      CONSTINT = CONSTINT * 10 + CARACT - 240
        CALL CARSUI
        if CARACT > = zero then goto 01
        return
```

Comme second exemple nous allons écrire le sous-programme

CARSUI appelé dans le premier exemple. On va voir qu'ici Mini-Fortran ne soutient pas la comparaison avec PL 360 ni même l'Assembleur.

1) en Assembleur

```
CARSUI  DS  0H
        LA  R3, 1(R3)
        BR  14
```

2) en PL 360

```
procedure CARSUI(R6);
begin R3 := R3 + 1; end;
```

3) en mini-Fortran

```
      subroutine CARSUI

C POINCAR  numéro de l'octet dans l'analyse de l'instruction
C NUMOCT   numéro de l'octet dans le mot en cours d'analyse
C MEMSUI   mot en cours d'analyse
C CARACT   caractère recherché
      NUMOCT = NUMOCT + 1
      if NUMOCT > = 4 then goto 34
      if NUMOCT = 3 then goto 33
      if NUMOCT = 2 then goto 32
C NUMOCT = 1 on prend le second octet
      MEMT = MEMSUI . and . MASK 2
      CARACT = MEMT/65536
      GOTO 35
C NUMOCT = 2 : troisième octet
32    MEMT = MEMSUI . and . MASK 3
      CARACT = MEMT/256
      GOTO 35
C NUMOCT = 3 : quatrième octet
33    CARACT = MEMSUI . and . MASK 4
      GOTO 35
```

```
C NUMOCT = 4 : mot suivant
        NUMOCT = 0
        MEMSUI = BUFFER(POINCAR)
        POINCAR = POINCAR + 1
        MEMT = MEMSUI . and . MASK 1
        CARACT = MEMT/16277216
        RETURN.
```

Comme nous l'avons montré dans le chapitre II il y a un avantage à écrire un compilateur en langage évolué.

Le choix du sous-ensemble minimum du langage pour l'écriture propre du compilateur est une restriction; il est évident que l'on aurait mieux fait d'écrire le compilateur en langage évolué, Algol W par exemple, qui traite bien les chaînes et les caractères mais on perdrait les avantages du bootstrapping proprement dit.

II) Définition de mini-Fortran

Nous n'avons pas trouvé de problèmes pour définir entièrement Fortran à partir de mini-Fortran en ce sens que tous les algorithmes de compilation ont été écrits facilement.

Il semblerait donc, mais un seul exemple est insuffisant pour le prouver véritablement, que la recherche d'un langage noyau telle que nous l'avons définie au chapitre IV conduise effectivement à un sous-ensemble suffisant.

III) Efficacité du langage noyau

Le langage noyau tel que nous l'avons défini originellement n'est pas apte à décrire de façon efficace, le langage complet dont il fait partie. Il est nécessaire de lui adjoindre des concepts compilatoires ou syntaxiques.

Nous avons ajouté deux concepts compilatoires.

- Le sous-programme qui permet de restreindre la taille du programme et qui a l'avantage de permettre une écriture plus aisée

- Le Dump qui permet de tester le second compilateur en donnant l'état de la mémoire.

Il n'est qu'à regarder l'écriture du second compilateur pour se rendre compte que la forme de l'instruction conditionnelle if choisie est inefficace car elle entraîne un nombre effrayant de ruptures de séquence.

L'absence de concepts aussi simples que DO nous a considérablement gênés dans l'écriture du second générateur.

C O N C L U S I O N

-:=-:=-:=-

Dans notre étude nous avons soulevé plusieurs problèmes qui nous ont semblé intéressants notamment la définition conceptuelle des langages de programmation et la technique du bootstrapping.

Nous regrettons seulement que cette étude n'ait pas fait partie d'un projet plus vaste, ce qui aurait peut-être mieux montré l'importance de ces problèmes. Néanmoins, nous pensons être arrivés à quelques résultats qui pour ne pas avoir été suffisamment approfondis pourront peut-être aider à d'éventuels projets.

Les différentes conclusions que nous allons donner sont assez décevantes sur le plan de notre seul travail; mais elles dégagent un esprit relativement nouveau appuyé par des recherches récentes dans ce domaine.

Tout d'abord, il nous faut donner une conclusion sur le bootstrapping lui-même. Premier pas de notre étude, c'est lui qui nous a amené à essayer de dégager un noyau conceptuel commun aux langages de programmation. Si l'on considère le bootstrapping dans le cas du passage d'un compilateur d'une machine sur une autre, nous avons une méthode artisanale qui présente de nombreux avantages. Le principal étant celui d'éviter l'écriture fastidieuse en langage machine et ensuite de permettre une mise au point facilitée. Dans ce cas, il nous semble que le bootstrapping soit une méthode très intéressante que l'on peut recommander d'employer.

Dans le cas de l'extension d'un langage par contre, notre conclusion sera relativement négative vis-à-vis de notre étude. Certes les mêmes avantages persistent mais nous pensons que - notre exemple ainsi que d'autres plus poussés le montrent - l'on a avantage à utiliser un langage noyau propre à l'ordinateur utilisé - PL/360, LP 70 - plutôt que de créer un compilateur pour un langage noyau. En effet ces langages semi-évolués sont faits de façon à utiliser au maximum les performances de l'ordinateur considéré et d'après leur syntaxe permettent d'avoir certains avantages du Bootstrapping.

En effet, les avantages que l'on a trouvés au bootstrapping existent, mais il est à se demander si ces avantages ne disparaissent pas devant le fait d'avoir à écrire deux compilateurs.

L'écriture du second compilateur contient une réécriture dans un autre langage du premier compilateur et ce seul fait annihile les bénéfices : - efficacité, simplicité - du bootstrapping.

Par contre l'écriture en langage semi-évolué permet de n'écrire qu'un seul compilateur, de plus cette écriture est facilitée par la syntaxe du langage et sa puissance égale à celle d'un langage d'assembleur.

Il n'est qu'à regarder l'écriture du compilateur Algol W écrit en PL 360 pour voir ce que l'on peut obtenir comme compilateur avec ce langage.

L'écriture dans un tel langage empêche que le compilateur puisse être utilisable sur un autre ordinateur que l'IBM/360 puisque PL 360 n'est opérationnel que sur cette machine.

De tels langages ne sont actuellement pas fournis par les constructeurs, et, ne sont encore pas généralisés malgré leurs grands avantages.

Mais, comme si nous le pensons, cette généralisation se fait prochainement, il sera facile de traduire un compilateur en écrivant simplement un traducteur de ces langages. Ces traducteurs travaillant sur des langages de bas niveau par exemple : PL 360 → LP 70 donneront une traduction efficace, ce qui ne fera rien perdre à l'efficacité du compilateur traduit.

Nous pensons que cette méthode est plus à retenir, que celle du bootstrapping à partir d'un langage noyau forcément de puissance restreinte.

De plus si l'on se base sur PL 360, le code produit par de tels compilateurs est tel que les questions de dépendance relative des compilateurs initiaux et résultants est négligeable.

La deuxième conclusion porte évidemment sur la seconde partie de notre étude : le langage noyau. L'étude du bootstrapping nous a amené à cette recherche d'un langage noyau pour un langage donné, nous avons alors été amenés à dépasser le cadre strict du bootstrapping pour aller jusqu'à la conception sémantique des langages de programmation, cette étude fut bien sûr la plus intéressante pour nous. C'est en effet un problème passionnant et assez peu étudié jusqu'à ces dernières années, c'est peut-être pourquoi certains lecteurs pourront trouver cette étude incomplète ou même inutile, habitués qu'ils sont à voir les langages de programmation principalement du côté syntaxique.

Une étude plus complète comme celle faite dans ELF (cf III-2) présente des avantages nombreux puisque l'étude sémantique est complétée par celle d'une syntaxe permettant la création d'un langage noyau.

Mais même dans ce cas, que nous considérons comme optimum, les mêmes restrictions d'efficacité existent et le problème ne sera vraiment résolu que lorsqu'à partir d'un langage noyau unique, on saura créer des langages ayant la même syntaxe que le noyau; l'extension se faisant dans un sens ou un autre selon les besoins de l'installation considérée et cette expansion pouvant être reprise à tout instant.

La solution idéale que nous voyons, ne le sera que si l'on ne dispose que de quelques grands langages bien définis : Algol 68 est le premier exemple de ces grandes classes de langages qui, nous le pensons, se développeront; on peut même espérer qu'un langage à tout faire style : PL/1, se généralisera au point de supplanter tous les autres langages.

Ceci peut effectivement être fait par une très grande firme d'ordinateurs imposant peu ou prou sa volonté sur le marché.

Ce qui est nécessaire c'est que ce langage soit défini de façon très formelle. Si cela est, on peut espérer que la plupart des concepts ajoutés pour étendre le langage pourront être exprimés à partir des règles existantes dans le langage.

Voici donc comment nous voyons les choses :

1) définir un langage noyau pour le langage, par exemple à l'aide des règles que nous donnons.

2) optimiser ce langage par l'adjonction de concepts compilatoires et éventuellement de concepts syntaxiques si ceux-ci se révèlent être d'une grande utilité pour l'écriture dans le sous-langage.

3) écrire un compilateur C1 pour ce langage noyau; choisir pour ceci un langage semi-évolué si on en dispose d'un.

4) écrire un compilateur C2 pour le langage lui-même, ou pour une partie de ce langage qui intéresse une certaine classe d'utilisateurs; choisir pour ceci le langage noyau dont le compilateur existe.

5) faire compiler ce compilateur par lui-même pour appliquer les règles du bootstrapping.

6) Si on désire modifier le langage
2 cas se présentent :

- les additifs sont exprimables sémantiquement et syntaxiquement à partir des règles du noyau : modifier C2 et retourner en 5.

- les additifs sont totalement nouveaux sémantiquement parlant, créer des règles syntaxiques et sémantiques et modifier C1 et retourner en 4.

7) Si on désire obtenir le compilateur du langage sur une nouvelle machine, retourner en 3.

Ce processus est évidemment idéal si l'on ne dispose que d'un langage. Ceci ne pouvant pas être le cas, on peut espérer que ce processus ne s'applique qu'à quelques langages peu nombreux, le processus n'est alors à appliquer qu'à ces quelques langages.

De plus si le langage noyau est bien choisi, on peut écrire plusieurs compilateurs ne traitant qu'une partie du langage, ceci selon les besoins du client.

L'écriture de ces compilateurs se faisant dans un langage unique ne serait pas très difficile. L'emploi de la méthode des compilateurs par la syntaxe s'impose pratiquement de lui-même.

Cette méthode est en effet celle qui s'intègre le mieux à celle (cf 1-11) du bootstrapping ainsi qu'on peut le voir au chapitre II-1.

-:=-:=-:=-:=-:=-:=-

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

-:=-:=-:=-:=-

Nous avons décomposé la liste des ouvrages en 4 parties

I Bootstrapping

II Compilation

III Définition

IV Langages

Certains ouvrages pourront se retrouver dans plusieurs parties.

I - BOOTSTRAPPING

1) E. BOOK, H. BRATMAN.

"Using compilers to build compilers"
S D C Santa Monica . Août 1960.

2) C. CHEMLA.

"Ecriture d'un compilateur Algol en Algol"
AFIRO Lille 1966.

3) M.I. HALPERN.

"Machine Independance : its technology and economics"
CACM Décembre 1965.

4) M.H. HALSTEAD.

"Machine Independant Computer Programming"
Spartan Books 1962.

5) P.Z. INGERMAN.

"A syntax oriented translator"
Academic Press 1966.

6) E.S. LOWRY, C.W. MEDLOCK.

"Object code optimisation"
CACM Janvier 1969.

7) K.S. MASTERSON.

"Compilation for two compilers with Neliac"
CACM Novembre 1960.

8) D. SUTY.

"Exposé et Commentaires de méthodes connues d'auto-compilation"
I.M.A.G. Janvier 1968.

9) W.H. WATTENBURG.

"Techniques for automating the construction of translators for programming languages"
Berkeley, University of California Janvier 1964.

10) M.V. WILKES.

"An experiment with a self-compiling compiler for a simple list processing language : WISP.

11) N. WIRTH, J.W. WELLS, E.H. SATTERTHWAITE.

"The PL 360 System"
CS 91 Stanford University April 1968.

II - COMPILATION

1) H. BAUER, S. BECKER, S. GRAHAM.

"Algol W implementation"
CS 98 Stanford University May 1968.

2) F.L. BAUER, K. SAMELSON.

"Sequential formula Translation"
CACM Février 1960.

3) L. BOLLIET.

"Notation et processus de traduction des langages symboliques"
Thèse d'Etat I.M.A.G. 1967.

4) T.E. CHEATHAM.

"The architecture of compilers"
Computer Associates 1964.

5) T.E. CHEATHAM.

"The theory and construction of compilers"
Computer Associates 1967.

6) E.W. DIJKSTRA.

"On the design of machine independant programming languages"
Annual review in Automatic Programming 1963.

7) M. GRIFFITHS.

"Analyse déterministe et compilateurs"
Thèse d'Etat I.M.A.G. Octobre 1969.

8) P.Z. INGERMAN.

"A syntax oriented compiler"
Academic Press 1966.

9) E.S. LOWRY, E.W. MEDLOCK.

"Objet code optimisation"
CACM Janvier 1964.

10) B. RANDELL, RUSSELL.

"Algol 60 Implementation"
Academic Press 1960.

11) J.P. VERJUS.

"Etude et Réalisation d'un système Algol conversationnel"
Thèse Ingénieur-Docteur I.M.A.G. Juillet 1968

III - DEFINITION DES LANGAGES

1) F.L. BAUER.

"Fondation conceptuelle et description formelle des langages de programmation"
Institut de Mathématiques et Centre de Calcul
Rapport n° 6701. München.

2) T.E. CHEATHAM, A. FISCHER, P. JORRAND.

"On the basis for ELF, an extensible language facility"
Computer Associates 1968.

3) J. COHEN.

"Langages pour l'écriture de compilateurs"
Thèse I.M.A.G. Juin 1967.

4) J.V. GARWICK.

"The definition of Programming languages by their compilers"
Congrès IFIP 1966.

5) A. VAN WIJNGAARDEN.

"Recursive definition of syntax and semantics"
Congrès IFIP 1966.

6) B. WILLIS.

"PL/1 in an extensible language environment"
I.M.A.G. Septembre 1969.

7) B. WILLIS.

"A base language for PL/1"
I.M.A.G. Avril 1970.

IV - LANGAGES

1) H. BAUER.

"Introduction to Algol W programming"
Stanford University.

2) H. BAUER, S. BECKER, S. GRAHAM, E. SATTERTHWAITE.

"Algol W language description"
Stanford University CS 110 Septembre 1969.

3) L. BOLLIET, N. GASTINEL, P.J. LAURENT.

"Algol un nouveau langage scientifique"

4) M. DREYFUS.

"Fortran IV Langage"
Dunod 1968.

5) IBM System/360

6) Principles of operations S360-01 A22-6821-7

7) Assembler Language S360-21 GC28-6514-6

8) Fortran IV Language S360-25 C28-6515-7

9) Algol Language S360-25 C28-6615-1

10) PL/1 Language reference manual S360-29 GC28-8201-3

11) O. LECARME.

"Etude comparative des principaux langages de programmation"
Thèse 3° Cycle I.M.A.G. 1965.

12) D. SUTY.

"Le langage Algol W"
I.M.A.G. 1970.

13) D. SUTY.

"Le langage PL 360"
I.M.A.G. 1971.

14) A. VAN WIJNGAARDEN, B.J. MAILLOUX, J.E.L. PECK, C.H.A. KOSTER.

"Report on the algorithmic language Algol 68"
Mathematisch Centrum, Amsterdam Novembre 1969.

15) N. WIRTH.

"A programming language for the 360 computer"
Journal ACM Janvier 1968.

16) N. WIRHT, J.N. WELLS, E.H. SATTERTHWAITE.

"The PL 360 System"
Stanford University CS 91 April 1968.

17) N. WIRTH, WEBER.

"Euler, a generalization of Algol 60 and its formal definition"
CACM Janvier 1966.

18) G. GOOS, K. LAGALLY, G. SAPPER.

"PS 440, Eine niedere Programmiersprache"
Rechenzentrum der T.U. München.
Bericht 7002 1970.

VU,

Grenoble, le

Le Président de la Thèse

Vu, et permis d'imprimer

Grenoble, le

Le Président de l'Université Scientifique et Médicale