Analyse déterministe et compilateurs

Michael Griffiths

To cite this version:

Michael Griffiths. Analyse déterministe et compilateurs. Autre [cs.OH]. Université Joseph-Fourier - Grenoble I, 1969. Français. NNT : . tel-00008419

HAL Id: tel-00008419
https://theses.hal.science/tel-00008419

Submitted on 9 Feb 2005

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

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

N° d'ordre

présentée à

LA FACULTE DES SCIENCES DE GRENOBLE

pour obtenir

LE GRADE DE DOCTEUR D'ETAT

par

Michael GRIFFITHS

Master of Arts, University of Oxford

Analyse déterministe et compilateurs

Thèse soutenue le 25 octobre 1969, devant la commission d'examen :

Monsieur	J. KUNTZMANN	Président
Messieurs	N. GASTINEL	
	L. BOLLIET	Examinateurs
	C. PAIR	

FACULTÉ DES SCIENCES

de GRENOBLE

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

DOYEN HONORAIRE : Monsieur MORET

DOYEN : Monsieur BONNIET

PROFESSEURS TITULAIRES

MM.	NEEL Louis	Physique Expérimentale
	HEILMANN René	Chimie Organique
	KRAVTCHENKO Julien	Mécanique Rationnelle
	CHABAUTY Claude	Calcul différentiel et intégral
	BENOIT Jean	Radioélectricité
	CHENE Marcel	Chimie Papetière
	FELICI Noël	Electrostatique
	KUNTZMANN Jean	Mathématiques Appliquées
	BARBIER Reynold	Géologie Appliquées
	SANTON Lucien	Mécanique des Fluides
	OZENDA Paul	Botanique
	FALLOT Maurice	Physique Industrielle
	KOSZUL Jean-Louis	Mathématiques
	GALVANI O.	Mathématiques
	MOUSSA André	Chimie Nucléaire
	TRAYNARD Philippe	Chimie Générale
	SOUTIF Michel	Physique Générale
	CRAYA Antoine	Hydrodynamique
	REULOS René	Théorie des Champs
	BESSON Jean	Chimie Minérale
	AYANT Yves	Physique Approfondie
	GALLISSOT	Mathématiques

Melle	LUTZ Elisabeth	Mathématiques
	BLAMBERT Maurice	Mathématiques
	BOUCHEZ Robert	Physique Nucléaire
	LLIBOUTRY Louis	Géophysique
	MICHEL Robert	Minéralogie et Pétrographie
	BONNIER Etienne	Electrochimie et Electrométallurgie
	DESSAUX Georges	Physiologie Animale
	PILLET E.	Physique Industrielle & Electrotechnique
	YOCCOZ Jean	Physique Nucléaire Théorique
	DEBELMAS Jacques	Géologie Générale
	GERBER Robert	Mathématiques
	PAUTHENET R.	Electrotechnique
	VAUQUOIS Bernard	Calcul Electronique
	BARJON Robert	Physique Nucléaire
	BARBIER Jean-Claude	Physique
	SILBER R.	Mécanique des Fluides
	BUYLE-BODIN Maurice	Electronique
	DREYFUS Bernard	Thermodynamique
	KLEIN Joseph	Mathématiques
	VAILLANT François	Zoologie et Hydrobiologie
	ARNAUD Paul	Chimie
	SENGEL Philippe	Zoologie
	BARNOUD Fernand	Biosynthèse de la Cellulose
	BRISSONNEAU P.	Physique
	GAGNAIRE Didier	Chimie Physique
Mme	KOFLER Lucie	Botanique
	DEGRANGE Charles	Zoologie
	PEBAY-PEROULA Jean-Claude	Physique
	RASSAT André	Chimie Systématique
	DUCROS Pierre	Cristallographie Physique
	DODU Jacques	Mécanique Appliquée I.U.T.
	ANGLES D'AURIAC Paul	Mécanique des Fluides
	LACAZE Albert	Thermodynamique

PROFESSEURS SANS CHAIRE

MM.	GIDON P.	Géologie et Minéralogie
	GIRAUD P.	Géologie
	PERRET R.	Servomécanisme
Mme	BARBIER M. J.	Electrochimie
Mme	SOUTIF J.	Physique
	COHEN Joseph	Electrotechnique
	DEPASSEL R.	Mécanique des Fluides
	GASTINEL Noël	Mathématiques Appliquées
	GLENAT René	Chimie
	BARRA Jean	Mathématiques Appliquées
	COUMES André	Electrotechnique
	PERRIAUX Jacques	Géologie et Minéralogie
	ROBERT André	Chimie Papetière
	BIAREZ Jean	Mécanique Physique
	BONNET Georges	Electronique
	CAUQUIS Georges	Chimie Générale
	BONNETAIN Lucien	Chimie Minérale
	DEPOMMIER Pierre	Physique Nucléaire et Génie Atomique
	HACQUES Gérard	Calcul Numérique
	POLOUJADOFF Michel	Electrotechnique
Mme	KAHANE Josette	Physique
Mme	BONNIER J. M.	Chimie
	VALENTIN Jacques	Physique
	PAYAN Jean-Jacques	Mathématiques

PROFESSEURS ASSOCIES

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

MAITRES DE CONFERENCES

MM.	LANCIA Roland	Physique Atomique
	DEPORTES C.	Chimie
Mme	BOUCHE Liane	Mathématiques
	SARROT-REYNAULD J.	Géologie Propédeutique
	KAHANE André	Physique Générale
	DOLIQUE Jean-Michel	Electronique
	BRIERE Georges	Physique
	DESRE Georges	Chimie
	LAZEROWICZ	Physique
	BERTRANDIAS J. P.	Mathématiques Appliquées
	LAURENT P.	Mathématiques Appliquées
Mme	BERTRANDIAS Françoise	Mathématiques Pures
	LONGEQUEUE J. P.	Physique
	SOHM Jean-Claude	Electrochimie
	ZADWORNY François	Electrotechnique
	DURAND F.	Chimie Physique
	CARLIER Georges	Biologie Végétale
	AUBERT Georges	Physique
	DELPUECH Jean-Jacques	Chimie Organique
	PFISTER Jean-Claude	Physique
	CHIBON P.	Biologie Animale
	IDELMAN S.	Physiologie Animale
	BOUVARD Maurice	Hydrologie
	RICHARD Lucien	Botanique
	PELMONT Jean	Physiologie Animale
	BLOCH Daniel	Electrotechnique I. P.
	BOUSSARD Jean-Claude	Mathématiques Appliquées
	MOREAU René	Hydraulique I.P.
	BRUGEL L.	Energétique I.U.T.
	SIBILLE R.	Construction Mécanique I.U.T.

	ARMAND Yves	Chimie I.U.T.
	BOLLIET Louis	Information I.U.T.
	KUHN Gérard	Energétique I.U.T.
	GERMAIN Jean-Pierre	Construction Mécanique I.U.T.
	CONTE René	Thermodynamique
	JOLY Jean-René	Mathématiques Pures
Mlle	PIERY Yvette	Biologie Animale
	BERNARD Alain	Mathématiques Pures

MAITRES DE CONFERENCES ASSOCIES

MM.	SAWCZUK A.	Mécanique des Fluides
	CHEEKE John	Thermodynamique
	YAMADA O.	Physique du Solide
	NATR Lubomir	Biologie Végétale
	NAYLOR Arch	Physique Industrielle
	SILBER Léo	Radioélectricité
	NOZAKI Akihiro	Mathématiques Appliquées
	RUTLEDGE Joseph	Mathématiques Appliquées
	DONOHO Paul	Physique Générale
	EGGER Kurt	Biologie Végétale

Je tiens à remercier

Monsieur le Professeur J. KUNTZMANN, qui a bien voulu me faire l'honneur de présider le jury,

Monsieur le Professeur N. GASTINEL, pour l'intérêt sympathique qu'il a porté à mon travail,

Monsieur le Professeur C. PAIR, qui a bien voulu le juger,

et

Monsieur le Professeur L. BOLLIET, à qui je dois mon séjour en France et dont les encouragements m'ont été précieux.

Je voudrais aussi remercier

Messieurs P.M. WOODWARD, J.M. FOSTER, D.P. JENKINS, I.F. CURRIE, P.R. WETHERALL et A.J. FOX, avec qui j'ai passé cinq ans d'apprentissage dans une ambiance très agréable ;

Madame M. CHABRE, Messieurs M. PELTIER et M. BERTHAUD, des collègues de bureau qui sont la gentillesse même ; en particulier, Michel BERTHAUD a eu le vain espoir de supprimer toute anglicisme de ce texte ;

L'équipe "software" du Laboratoire de Mathématiques Appliquées pour sa constante coopération, et en particulier les membres de cette équipe qui ont participé aux projets décrits ici ; et

ceux qui ont contribué à la réalisation matérielle de cette thèse.

A ma Femme

TABLE DES MATIERES

Page

CHAPITRE I : Introduction 2

CHAPITRE II : Définition de Langages 7

2.1. Contrôle du Pseudo-Parallélisme en FROLIC 7
2.2. Contrôle des Tâches pour RTL 12
2.3. CORAL 66 14
2.4. Extensions des Langages par des Commandes Conversationnelles 17
2.5. Le Macro-Langage 22
2.6. Remarques Finales 27

CHAPITRE III: Théories Syntaxiques 29

3.1. Analyse Prédictive avec Génération d'un Arbre 29
3.2. Transformation d'une Grammaire et Déterminisme 33
3.3. Production d'un Analyseur 35
3.4. Théorie du Déterminisme 37
3.5. Algorithmes de Transformation et de Test 43
3.6. Génération Téléscopique 48

CHAPITRE IV : Outils Sémantiques 52

4.1. Synchronisation avec la Syntaxe 52
4.2. Gestion des Listes 53

Page

CHAPITRE V : Algorithmes Sémantiques 57

5.1. Allocation Dynamique Classique 57
5.2. Environnements 59
5.3. Allocation Statique de CORAL 66 62
5.4. Contrôle de l'Interprétation 63
5.5. Problèmes de Portée en Interprétation 64
5.6. La Table des Symboles 65
5.7. Communication entre Segments 66
5.8. Allocation pour Plusieurs Utilisateurs en Parallèle 67
5.9. Les Mots-Clés en PL/I 68
5.10. Sur les Types des Variables 69

CHAPITRE VI : Conclusion 73

Bibliographie 76

CHAPITRE I

INTRODUCTION

1. INTRODUCTION

Ce document présente un ensemble de travaux concernant la définition et l'implantation des langages de programmation. Bien que la plupart de ces travaux aient été réalisés en vue d'applications pratiques sur calculateurs électroniques, ils contiennent tous des aspects théoriques.

La seule méthode utilisable en vue de tester la validité d'un système de programmation est de le mettre à la disposition d'utilisateurs, de préférence professionnels, et de recueillir leurs opinions, après un délai d'acclimatation pendant lequel le produit est perfectionné. On pourra constater que dans la plupart des projets présentés, un produit a été, ou sera, mis sur le marché. La programmation se trouve actuellement dans le même état que les mathématiques d'il y a deux mille ans, nos théories et nos calculs étant jugés sur leur applicabilité aux problèmes qui se posent dans le monde. Le sens de la valeur en soi qui vient avec la maturité des mathématiques nous manque encore, ce qui offre en même temps des avantages et des inconvénients.

Nous allons décrire des travaux accomplis des deux côtés de la Manche, d'une part en Angleterre au **"Royal Radar Establishment"**, un laboratoire des services scientifiques gouvernementaux anglais (l'équivalent du C.N.R.S. français), et d'autre part en France à l'**Université de Grenoble**, pour la plupart en collaboration avec le **Centre Scientifique IBM-FRANCE**. Quand on considère la taille des problèmes abordés, il est évident qu'ils nécessitent presque toujours un travail d'équipe.

Les projets que nous avons réalisé en Angleterre se décomposent comme suit :

- un compilateur expérimental pour un sur-ensemble d'ALGOL destiné à des problèmes de type simulation
- un langage pour la description des processus en temps réel
- une série de compilateurs pour un autre langage en temps réel
- le développement de certains outils pour l'écriture de compilateurs.

Le compilateur pour le langage de simulation (FROLIC) a été écrit mais il n'a jamais été incorporé à un système. Sa valeur éducative et théorique est grande mais non sa valeur économique. Une courte note sur ce langage est disponible [1]

Le langage RTL (Real Time Language) n'a pas encore été implanté à ma connaissance. Il a été conçu à la demande de la "British Computer Society" dans l'espoir d'influencer les compagnies britanniques. Il est toujours difficile de savoir si un tel essai a réussi. L'équipe qui a fait le travail comprenait outre des fonctionnaires, des industriels et des universitaires. Ma contribution personnelle a porté sur le contrôle des tâches, des priorités et des interruptions. Ces idées étaient fortement inspirées par FROLIC. Un rapport [2] a été publié par la British Computer Society.

RTL est probablement trop complet et trop lourd. En tout cas ce langage n'est pas applicable à la gamme de calculateurs souvent employés pour contrôler les processus chimiques (ou industriels), la circulation urbaine ou aérienne. Pour ces raisons, le langage CORAL 66 [3] a été conçu pour les calculateurs sans arithmétique en virgule flottante. CORAL 66 est un successeur de CORAL [4] basé sur ALGOL 60 [5]. Les raisons de ce choix sont données par P.M. WOODWARD [6]. Plusieurs compilateurs ont été produits à l'aide d'un "générateur téléscopique" (bootstrapping) [7] en utilisant des méthodes basées sur un travail de J.M. FOSTE [8]. Ces compilateurs ont été réalisés en collaboration avec I.F. CURRIE.

Cette génération téléscopique est un des outils qui sera décrit d'une manière plus détaillée. Ces outils ont été améliorés par le travail en France [29] qui comprend également un langage de macros utilisé pour la définition de la partie sémantique d'un compilateur [9] , [30] . Ce langage, comme tous les projets français, a été créé en collaboration avec M. PELTIER.

Notre première étude en France a porté sur la compilation incrémentielle d'ALGOL 60 [10] . Cette étude a dérivé ensuite vers PL/I et la première version d'un compilateur est en cours d'achèvement. Les spécifications externes [11] et la technologie utilisée [12] ont été le sujet de publications. M. BERTHAUD écrit l'interpréteur de PL/I pour la Compagnie IBM et Madame CHABRE continue le développement du compilateur ALGOL pour l'Université.

La syntaxe de PL/I est d'une taille telle qu'il ne nous semblait pas réaliste de la traiter à la main. Une nouvelle version du programme de J.M. FOSTER [8] a été écrite, en tenant compte des idées de D.E. KNUTH [13] . Deux parties composent ce programme : transformation de la grammaire en une nouvelle forme, puis examen de cette forme pour en vérifier le déterminisme. Plus récemment, la génération d'un analyseur compatible avec le langage de macros [9] a été ajoutée. J. BORDIER examine maintenant les améliorations pouvant être apportées à ce travail.

Utilisant le transformateur de grammaire, en collaboration avec G. TASSART, une étude a été menée sur les propriétés de la grammaire de PL/I du fait de l'absence de discrimination typographique des mots-clés [14] . La grammaire utilisée pour ces études a été écrite à partir du manuel officiel PL/I [15] . Il existe maintenant une grammaire de PL/I diffusée par IBM [16] qu'il serait bon d'utiliser. Malheureusement, cette grammaire n'est pas exprimée dans la forme normale de BACKUS mais dans la forme SRL ; M. ASSABGUI travaille actuellement sur cette forme de la grammaire pour la rendre utilisable dans notre système.

Par ailleurs, IBM nous a demandé de faire un analyseur de syntaxe pour PL/I, incluant quelques tests sémantiques. Ce programme est au point et il est en cours de distribution. Il est à noter que PL/I est si vaste et si complexe que cet analyseur contient autant d'instructions qu'un compilateur ALGOL complet. Toute la documentation à ce sujet n'est pas encore prête mais un papier préliminaire décrit certains algorithmes [17] .

Dans le but de montrer les liens entre tous ces projets, cet exposé est divisé suivant la nature des idées utilisées plutôt que suivant la nature des applications. La première partie traitera de la définition des langages et la deuxième partie de l'analyse de la syntaxe en utilisant les outils développés dans ce domaine.

En ce qui concerne la sémantique, une partie sera consacrée aux outils, et l'autre à des algorithmes importants, en particulier l'allocation de mémoire pour des variables, et la recherche de leurs valeurs.

CHAPITRE II

DEFINITION DE LANGAGES

2. DEFINITION DE LANGAGES

De nouveaux langages, ou des extensions des langages existants, se trouvent da FROLIC, les deux projets pour le temps réel, le langage de macros et le langage de commande pour les compilateurs incrémentiels. Dans FROLIC, RTL et le langage de c mande on trouve un essai de définition de fonctions linguistiques destinées à trai- ter de nouvelles applications des calculateurs. CORAL 66 et le langage de macros ont été créés pour des raisons techniques car l'utilisation des langages existants n'était pas économiquement valable. Cette question économique devient actuellemen de plus en plus importante. Les langages de haut niveau sont très puissants et tr généraux mais, en contrepartie, ils sont coûteux en temps et en place ; quand ce coût devient trop important il faut définir un langage de bas niveau particulier a problème considéré. C'est pour cette raison que CORAL 66 et le langage de macros sont de nouveaux langages (basés néanmoins sur des idées d'ALGOL) et que les autre langages en sont des extensions.

2.1. Contrôle du Pseudo-Parallélisme en FROLIC

Les extensions à ALGOL qui apparaissent en FROLIC concernent la simulati d'opérations qui dans le monde réel se déroulent en parallèle. Il n'y a aucun cha gement à la syntaxe d'ALGOL, mais des "procédures standards" ont été ajoutées qui modifient éventuellement le déroulement du programme. Cette méthode d'extension présente des avantages du point du vue de la compatibilité et de l'implantation, mais il se trouve qu'elle n'est ni assez claire ni assez sûre. Avant de préciser ces points, il faut résumer les facilités demandées et réalisées.

L'utilisateur veut simuler une situation dans laquelle des tâches sont activées par des commandes et se déroulent en parallèle. Pour créer un modèle, il lui faut un moyen pour simuler le déroulement du temps et l'apparition des signaux extérieurs. Il doit être aussi possible de communiquer certaines informations entre les tâches et en particulier d'avoir un moyen de continuer ou de suspendre l'exécution d'une tâche sur une condition fournie par une autre tâche.

Pour accomplir ces fonctions, les procédures standard

```
activate (f)
delay (n)
waitfor (e)
```

sont disponibles, ainsi que d'autres routines, par exemple pour créer des séquences de nombres aléatoires. Une tâche pseudo-parallèle est une procédure qui est initialisée par un appel de la procédure "activate" ayant comme paramètre le nom de la tâche. La tâche appelée donne, dès ce moment, l'impression de se dérouler en parallèle avec toutes les autres tâches actives. Une tâche est terminée à la fin de la procédure correspondante. En vue de commencer l'exécution du programme, le système simule une activation du bloc le plus extérieur. A la fin de ce bloc, le programme est terminé et les tâches encore actives sont arrêtées.

Les tâches sont contrôlées par une pseudo-horloge qui peut être interrogée. Vis-à-vis de cette horloge les instructions machine ne prennent aucun temps et tout incrément de temps est commandé par le programmeur. Lorsqu'il estime qu'une opération dans le monde réel aurait pris n unités de temps, l'instruction

```
delay (n)
```

suspend la tâche pendant n intervalles de pseudo-temps.

Si le déroulement d'une tâche ne dépend pas seulement du temps, l'instruction

waitfor (expression booléenne)

suspend cette tâche jusqu'à ce que l'expression booléenne devienne vrai. La communication entre les tâches est établie à l'aide de variables communes aux procédures selon les règles normales de portée d'ALGOL.

Il est à noter que la procédure "waitfor" donne la possibilité de synchroniser deux processus en parallèle. A la même époque, DIJKSTRA a montré comment faire ce travail plus systématiquement [18] .

L'exemple ci-dessous simule la circulation des automobiles à des feux tricolores.

```
        begin integer n, temps, i, nbdevoitures ;
              read(n, total) ;
  comment n est le nombre de routes, total est le nombre total de voitures qui
            passent pendant la simulation ;
              begin boolean array feux [1:n] ;
                    integer array queue [1:n] ;
  comment queue [i]  contient le nombre de voitures attendant feux [i] ;
                    procédure changerfeux ;
  comment changerfeux  change la couleur des feux toutes les 30 secondes ;
                    begin integer i ;
                      p :   for i := 1 step 1 until n do
                            begin feux [ if i=1
                                         then n
                                         else (i-1)] := false ;
                                  feux [i] := true ;
                                  delay (30)
                            end ;
```

```
                    goto p
                end changerfeux ;
                procédure passer ;
comment passer permet à une voiture de passer le feu vert toutes les trois
        secondes ;
                begin
                  p :   for i := 0, i+1 while not lights [i]  do ;
                        if queue [i] ≠ 0
                        then begin queue [i] := queue [i] - 1 ;
                                   nbdevoitures := nbdevoitures + 1
                             end ;
                        delay (3) ;
                        goto p
                end passer ;
                procédure arriver ;
comment arriver simule l'arrivée des voitures ;
                begin integer a ;
                  p :   a := aléatoire entre (1, n) ;
comment a contient un nombre aléatoire entre 1 et n ;
                        queue [a] := queue [a] + 1 ;
                        delay (aléatoire) ;
comment aléatoire est une procédure retournant un nombre aléatoire ;
                        goto p
                end arriver ;
                for i := 1 step 1 until n do
                begin queue [i] := 0 ;
                          feux [i] := faux
                end ;
```

```
                  nbdevoitures := 0 ;
                  activate (changerfeux) ;
                  activate (arriver) ;
                  activate (passer) ;
                  waitfor (nbdevoitures = total) ;
comment imprimer ici les résultats ;
     end
end
```

La définition du langage FROLIC et son implantation contiennent en germe les idées et les méthodes qui seront utilisées dans les projets futurs. Il est néanmoins évident que FROLIC possède des défauts techniques qui le rendent imparfait. Pour les tâches, il y a deux inconvénients. L'un concerne la sortie d'une procédure par GOTO, car il est difficile de déterminer si la tâche doit être terminée ou non. L'autre ennui est qu'une tâche ne peut être une procédure avec des paramètres, car on ne peut pas donner les paramètres effectifs d'un nom de procédure, correspondant à un paramètre formel spécifié <u>procédure</u>, au moment de l'appel de la procédure activate. Ainsi, la forme suivante est illégale en ALGOL :

activate (g(x))

avec la déclaration

<u>procédure</u> activate (f) ; <u>procédure</u> f ;

Pour éviter cet inconvénient, une solution consiste à disposer d'un nouveau mot-clé

<u>activate</u>

ce qui ajoute à la syntaxe d'ALGOL la règle suivante :

<Statement> ::= <u>activate</u> <procédure identifier> |
<u>activate</u> <procédure identifier> (<actual parameter list>)

FROLIC aurait été utilisable malgré ses défauts, mais il est à noter que d'autres systèmes ont vu le jour en même temps ou peu après. On peut mentionner GPSS produit par IBM [19] , SIMULA en Norvège [20] et SIMSCRIPT de la RAND Corporation [21] .

2.2. Contrôle des Tâches pour RTL

Nous ne considérons ici que la partie de RTL développée à partir des idées de FROLIC. Le langage complet est vaste et le travail des autres sections a été fait par d'autres membres de notre groupe.

FROLIC n'est pas suffisant pour le traitement des problèmes en temps réel. L'exécution du code machine prend réellement du temps et donc une opération n'est pas obligatoirement terminée avant l'arrivée d'un signal extérieur demandant l'exécution d'une autre opération. Il faut aussi associer les signaux extérieurs avec les programmes devant être exécutés. Il existe, donc, en RTL une instruction "ON" qui permet de prendre en compte ces signaux avec un système de priorités. Les dispositifs spéciaux de FROLIC sont conservés, mais toutes les instructions sont de vraies instructions et non des appels à des procédures. La coopération entre des processus parallèles est assurée par la méthode de DIJKSTRA.

On a des instructions comme :

> activate procédure withpriority expression arithmétique
> on identificateur réservé do instruction
> secure variable
> release variable

où secure et release correspondent aux P et V de DIJKSTRA [18], l'identificateur

après on est le nom d'un signal extérieur possible et la procédure après activate est une "program procédure". Ce dernier concept mérite un peu plus d'attention. Une telle procédure ne peut pas terminer par un goto à une étiquette extérieure à la procédure, cette restriction pouvant être examinée à la compilation. En plus, pour être sûr qu'une sortie "cachée" n'existe pas, une program procédure ne peut appeler une procédure non-locale que si la procédure appelée est soit une program procédure, soit une routine de la bibliothèque. Donc, toute program procédure se termine à l'instruction end correspondante et comme toute tâche est une program procédure, le point de terminaison d'une tâche est bien défini.

L'exemple artificiel qui suit montre une utilisation possible des instructions on, secure et release. Considérons le système d'exploitation d'une calculatrice dans lequel, à un moment donné, on veut demander le montage d'une bande par l'opérateur. On suppose que l'unité d'affichage des messages est un écran de télévision auquel plusieurs processeurs ont accès.

```
secure écran ;
bandemontée := faux ;
afficher ('veuillez monter bande 1234 sur unité 56') ;
waitfor bandemontée ;
release écran ;
:
```

Ailleurs dans le même programme, on trouve les instructions suivantes :

```
on interrupt56 do begin if nbdebande(56) = 1234
                        then bandemontée := vrai
                        else afficher ('mauvaise bande sur 56')
                  end
```

La variable écran est un entier qui a la valeur 1 si l'écran est libre et 0 autrement. secure écran attend que écran ait la valeur 1, remet cette variable à zéro et prend le contrôle de l'écran de télévision, jusqu'à nouvel ordre. release remet écran à la valeur 1. L'interruption provoquée par le montage de l'unité de

bandes est récupérée par l'instruction on, qui teste que la bande est effectivement celle demandée et affecte la valeur vrai à la variable booléenne bandemontée. Ceci permet au programme principal de continuer, car waitfor l'avait suspendu tant que la condition restait faux.

La définition du langage traitant du contrôle des tâches en RTL est supérieure à celle de FROLIC pour l'utilisateur et pour le réalisateur. En comparaison avec les facilités analogues de langages comme PL/I, c'est un travail de base utilisable et éducatif.

2.3. CORAL 66

Ce langage a été conçu pour des ordinateurs de taille réduite sans arithmétique en virgule flottante, pour les applications en temps réel. Plusieurs projets civils et militaires ont été considérés, entre autres, le contrôle des feux tricolores de circulation à LONDRES et le contrôle des radars donnant de l'information sur la circulation aérienne. Ces deux installations sont aujourd'hui en état de marche et le langage a été implanté sur les machines suivantes : RREAC PLESSEY XL9, ELLIOTT 920, MARCONI MYRIAD, ICT 1902 et une machine de FERRANTI.

CORAL 66 est un langage de haut niveau analogue à ALGOL 60 dont l'un des objectifs est d'être en même temps efficace. Les types des données sont inspirés de JOVIAL [22] et plus précisément d'un sous-ensemble de JOVIAL appelé lui-même CORAL [4] . Nous ne donnons ici que les grandes lignes de la définition, qui se trouve dans une version complète en [3] .

En général, la structure d'un programme est identique à celle d'un programme ALGOL. Les expressions simples et les affectations ne sont pas changées. Les boucles pour ainsi que les instructions ou expressions conditionnelles ont la même syntaxe mais ont un contenu sémantique différent. Dans l'instruction

pour variable := expression pas expression jusqu'à expression

l'évaluation du pas et de la valeur finale n'est faite qu'une seule fois au début de la boucle. Après if la condition peut n'être évaluée que partiellement. Par exemple, si a est faux dans

a et b

la condition complète est nécessairement faux et b n'a pas besoin d'être évalué. Donc, les effets de bord d'ALGOL sont délibérément supprimés, d'où un gain en efficacité.

Les blocs et les procédures sont identiques à ceux d'ALGOL, excepté quelques changements dans les détails de la syntaxe des procédures. Les spécifications apparaissent dans la liste de paramètres ; par exemple

```
PROCEDURE quad (VALUE INTEGER a,b) ;
```

La valeur d'une procédure n'est pas affectée à l'identificateur de la procédure mais est fournie par l'instruction ANSWER ; par exemple

```
INTEGER PROCEDURE f ;
BEGIN
.
.
.
          ANSWER x + 1
END
```

Dans ce cas, la valeur de f est x + 1. La première idée se retrouve en ALGOL 68[2] et la deuxième en PL/I avec l'instruction RETURN. Ces changements ont été faits dans le but d'augmenter la simplicité et la clarté du programme.

L'arithmétique nonentière dans une machine sans virgule flottante pose de nombreux problèmes qui ne sont pas encore résolus. Dans CORAL 66 on essaie de donner des règles simples et claires, même si l'utilisateur doit faire certaines opérations lui-même. La déclaration d'un nombre de ce type est de la forme

FIXED (n_1, n_2) identificateur, identificateur ...

où n_1 est le nombre total de bits utilisés pour la représentation, y inclus le bit signe, et n_2 est le nombre de bits après le point (n_2 peut être négatif). La paire (n_1, n_2) est appelée l'échelle de la variable correspondante. Toute expression est évaluée en utilisant l'échelle soit de la variable en partie gauche dans une instruction d'affectation, soit de la première variable dans une condition. Ce système de conversion peut donner une "mauvaise" réponse en certains cas.

Dans ce langage il n'est pas suffisant de traiter des données qui remplissent un mot. Par la suite, on trouvera en CORAL 66 le concept de table, qui est un tableau d'enregistrements de longueur fixe, chaque enregistrement étant divisé en champs nommés d'une longueur mesurée en bits. Par exemple, si pour une machine ayant des mots de 24 bits on déclare

```
TABLE soldat [2,100]
             [numéro (24) 0, 0 ;
              âge (12 UNSIGNED 4) 1, 0 ;
              hauteur (7) 1, 12 ]
```

on a une table de cent enregistrements de deux mots. Chaque enregistrement comporte trois champs, appelés numéro, âge et hauteur. Le champ numéro comporte 24 bits et commence au mot 0 bit 0. Le champ âge contient un nombre sans signe de 12 bits (dont 4 bits après le point) et commence au mot 1 bit 0. Le champ hauteur est de 7 bits commençant au mot 1 bit 12. La table est référencée soit par le nom général soldat, soit par un membre particulier. Par exemple

```
âge [17]
```

est l'âge du dix-septième soldat. Cette facilité répond à des besoins exprimés par des utilisateurs, mais doit, évidemment, être utilisée avec attention.

On voit qu'en CORAL il n'est pas fait mention de la gestion des tâches, qui est faite par un superviseur séparé du compilateur. Ce choix nous était imposé dans la première implantation car le système était déjà spécifié. Néanmoins, nous pensons

que cette gestion des tâches serait mieux faite dans le compilateur aussi bien pou
l'utilisateur que pour le réalisateur. L'idée d'introduire la communication avec
monde extérieur dans le texte du programme à l'aide d'un langage compatible et de
même niveau se retrouve dans les projets de compilation incrémentielle. N. WIRTH a
exprimé la même opinion sur ce sujet [24] .

2.4. Extension des Langages par des Commandes Conversationnelles

L'étude et la réalisation de compilateurs incrémentiels nous ont amenés à définir de nouvelles instructions, qui sont incorporés dans le langage hôte, ceci e vue d'offrir à l'utilisateur des facilités conversationnelles qui n'étaient pas envisagées dans la définition originale. Les langages considérés sont ALGOL, PL/I e FORTRAN et les extensions sont les mêmes dans les trois cas. Une vue d'ensemble de trois projets apparaît en [25] .

Les compilateurs incrémentiels seront utilisés pour l'enseignement des la gages et pour la mise au point de programmes. L'organe d'entrée du programme est u machine à écrire et le système gère des dizaines de machines en parallèle. Le programme est entré, analysé, executé et changé ligne par ligne, ou plus exactement, incrément par incrément, l'incrément étant généralement une instruction.

Un programme consiste en un ou plusieurs segments formés d'incréments, do le nombre est limité à 256, chaque segment contenant au maximum 128 identificateurs Ces limitations sont évidemment des limitations d'impiémentation. La commande

<u>segment</u> segmentname

initialise l'analyseur/générateur et le système accepte les incréments qui suivent comme appartenant au segment nommé.

La commande

endsegment

termine le segment en cours, qui est protégé sur disque - sous une forme adéquate - en vue d'une exécution ultérieure. Dans un système de ce type le passage d'information n'est pas à sens unique, le système écrivant ses commentaires au fur et à mesure. De plus, le système numérote les lignes du segment pour pouvoir s'y référer plus tard. L'impression du numéro de la ligne suivante se fait après l'analyse de chaque ligne et indique que cette ligne est correcte. Dans le cas d'une erreur syntaxique, un message est imprimé et le numéro de la ligne est répété. On pourrait concevoir le dialogue suivant en ALGOL :

```
           segment p ;
    1      begin
    2      rel x, y ;
ERREUR. LE SYMBOLE rel N'EXISTE PAS
    2      real x, y ;
    3      .
    :      :
   10      endsegment
```

Il est déjà évident que le langage mis à la disposition de l'utilisateur n'est pas le même que le langage utilisé par le système. Le langage du système n'est qu'une collection de messages sans structures syntaxiques, qui signale certains évènements au programmeur.

Si le programmeur veut exécuter un segment, il peut utiliser la commande

exécute segmentname

qui initialise l'interprétation du segment indiqué. L'exécution est interrompue par

l'apparition de l'une des quatre conditions suivantes :

- une instruction wait dans le programme original
- une erreur sémantique dans le programme
- la fin du programme
- une interruption créée à la console par l'utilisateur.

Si le programme n'est pas fini, il reste dans un état suspendu où l'utilisateur a l contrôle. Il peut alors frapper n'importe quelle instruction en langage source qui est légale au point d'arrêt dans le texte original. Sauf pour certaines erreurs, c point d'arrêt est entre deux incréments. L'instruction est interprétée dans le con texte du programme à son point d'arrêt mais n'est pas insérée dans le programme. Quand l'instruction est terminée, le programme est remis dans l'état suspendu. L'ex cution du programme est reprise à partir de ce point à la réception de la commande

continue

et se déroule normalement jusq'au prochain arrêt. Cette facilité peut être utilis entre autre, pour l'impression, la modification ou l'initialisation des valeurs des variables.

Il est normal, dans un système conversationnel, de vouloir changer ou ajo ter des instructions à un programme. Dans le système proposé, ceci est possible da l'état suspendu ou pendant l'entrée d'un programme. La commande

edit nomdesegment

initialise de tels changements. La commande edit n'est pas nécessaire si le segmen à éditer est le segment en cours. La commande

after numérodeligne

insère les incréments qui suivent dans le segment comme si elles figuraient dans le texte original immédiatement après la ligne spécifiée. Cependant, les nouvelles in structions prennent les prochains numéros disponibles et non les numéros qu'elles auraient eu si l'utilisateur les avait frappées en séquence au point choisi. Par

exemple, on pourrait avoir le dialogue suivant concernant le segment ci-dessus :

```
         edit p ;
   *     after 3 ;
  11     x := y * 2 ;
  12     endedit ;
```

L'étoile accuse réception de la commande édit et la ligne numéro 11 suivra logiquement la ligne 3. L'édition est terminée par la commande

endedit

qui remet le système dans l'état précédant la commande edit correspondante. D'autres commandes d'édition sont

replace numérodeligne

qui remplace une ligne nommée par la ligne frappée, et

omit numéro1, numéro2

qui élimine les lignes à partir de numéro1 jusqu'à numéro2 inclus.

Il est regrettable, dans ce cas, qu'une machine à écrire ne soit pas un écran de télévision. L'édition ne peut, sans recopier le texte complet, placer les nouvelles instructions physiquement au bon endroit sur le papier. L'utilisateur se trouve avec trois séquences différentes dans son programme : l'ordre écrit, l'ordre qui serait exposé sur un écran de télévision et l'ordre d'exécution (dans lequel il y a toutes les déclarations en tête, et qui change avec les goto). Il est parfois difficile pour lui de s'y retrouver entre les trois.

L'utilisateur devrait s'efforcer d'écrire son programme en segments de taille limitée. Le système lui donne des facilités d'exécution d'un segment appelé comme un sous-programme. Comme il n'y a aucune association entre les identificateurs des segments, tout identificateur non-déclaré dans un segment doit être spécifié non-local à ce segment et traité comme un paramètre. L'instruction nonlocal est toujours

placée à la fin d'un segment. Au moment de l'appel d'un segment il y a donc une l
d'identificateurs qui doivent être associés avec les identificateurs non locaux du
segment appelé. Par exemple

```
         segment seg1 ;
   1     begin
   2     real a, b ;
   3     a := c * 2 ;
   :     :
  10     end
  11     nonlocal c ;
  12     endsegment
   *     segment seg2 ;
   1     begin
   2     real a ;
   :     :
   5     include seg1 using a ;
   :     :
  10     end
  11     endsegment
   *     exécute seg2
   *
```

L'instruction include dans seg2 provoque l'exécution de seg1 à ce point. Toute ré
rence à c dans seg1 (c étant déclaré nonlocal) est prise comme une référence à a da
seg2 (a étant le paramètre dans le using). La présence d'un a déclaré dans seg1 ne
provoque pas de confusion, car les caractères d'un identificateur ne sont pas connu
hors de son segment.

Les langages traditionnels ne sont évidemment pas faits pour ce genre d'exercice, et il est probable qu'un langage comme APL [26] ou POP2 [27] serait plus indiqué. Néanmoins, l'extension d'un langage existant est valable pour des raisons de compatibilité. Comme ce système sera utilisé en premier lieu pour l'enseignement et pour la mise au point de programmes en vue d'une exploitation ultérieure, il faut traiter les langages que l'étudiant apprend et pour lesquels il existe un compilateur assez efficace. Il sera nécessaire de créer, plus tard, des langages spéciaux pour le mode conversationnel qui seront plus directs et moins compliqués.

2.5. Le Macro-Langage

En vue de l'écriture d'un compilateur, la partie syntaxique a été beaucoup étudiée et il existe plusieurs formalismes qui seront mentionnées ci-dessous (Chapitre 3). Par contre, la partie sémantique est souvent réalisée en code machine, ou, dans certains cas (par exemple [28] et CORAL 66) en un langage évolué. La première méthode a le désavantage d'être lente du point de vue de la programmation, et d'autre part d'être lente à l'exécution. En plus, un compilateur doit garder le contrôle d'un programme au plus bas niveau. Dans l'écriture d'un interpréteur, par exemple, la conversion automatique de type faite par le langage évolué est souvent différente de la conversion requise.

Nous avons donc créé un langage de bas niveau analogue à PL/360 [60] qui utilise les macros-instructions de l'assembleur 360.

Un autre critère qui joue dans la définition du langage de macros est celui de la compatibilité avec la sortie du transformateur de grammaire [29] . Cette sortie est un ensemble de routines qui s'appelent dans un ordre indéterminé et qui sont fortement récursives.

La base du langage est définie par les deux macros :

ROUTINE nomderoutine

et

RETURN

qui délimitent une routine. On peut appeler une routine par

CALL nom-de-routine

La macro

EXIT

est équivalent à un saut à la fin de la routine. Le nom d'une routine est associé avec un entier par

SETINDEX nom-de-routine, entier

Chaque nom de routine doit être associé avec un entier différent.

Les routines d'un programme sont éventuellement distribuées parmi plusieurs modules, délimités par

MODULE entier

et

ENDROUTS

Un module peut être assemblé indépendamment des autres modules d'un programme. Les modules sont associés chacun à un entier différent.

A l'intérieur d'une routine on peut trouver les instructions normales de l'assembleur 360 ou des macros du langage. Celles-ci permettent de réaliser des tests de conditions, des boucles, des expressions et des affectations. Une condition a la forme

longueur,opérande1,opérateurderelation,opérande2

où la longueur est B(ou C), H, F (byte, halfword, fullword), les deux opérandes

peuvent représenter :

- le nom d'un registre
- une constante explicite
- le nom d'une constante
- le nom d'un espace de travail
- un déplacement avec base et éventuellement index

et les opérateurs admis sont :

GT (supérieur à)
LT (inférieur à)
GE (supérieur ou égal à)
LE (inférieur ou égal à)
EQ (égal à)
NE (différent de)

La condition est évaluée avec la longueur spécifiée et possède la valeur vrai ou faux.

Les conditions sont utilisées directement dans la macro

IF Condition

qui est toujours suivie par

ENDIF

ou par

ENDIF ELSE

et dans ce deuxième cas encore suivie par

ENDELSE

L'ensemble d'instructions entre IF et ENDIF est exécuté si la condition est vrai, et celui entre ELSE et ENDELSE si la condition est faux.

Les boucles utilisent aussi les conditions, avec

WHILE condition

qui répète les instructions jusqu'à

ENDWHILE

tant que la condition reste vrai, le test étant fait avant chaque exécution de la

boucle. Alternativement

DO

suivi par

ENDO WHILE,Condition

réalise une boucle en faisant le test après chaque exécution. La partie WHILE est optionnelle en ce cas, permettant d'obtenir une boucle terminée par le programmeur. Des conditions et des boucles peuvent être imbriquées les unes dans les autres.

Les affectations sont un cas spécial des expressions, qui sont évaluées par

ASSIGN longueur,expression

L'expression est une liste d'opérandes séparés par des opérateurs choisi parmi :

—> <— (affectations)

'—>''<—' (décalages)

\+ - *

L'évaluation est de gauche à droite sans priorité. Il y a des restrictions importantes. Dans une ASSIGN, l'affectation est soit à gauche, soit à droite. Seule l'affectation à droite peut être multiple. Un signe d'affectation à gauche est toujours le premier opérateur. Si l'affectation est à gauche vers un registre,ce registre est utilisé comme registre de travail, c'est-à-dire que l'expression est évaluée dans ce registre. Dans tous les autres cas, un registre du système est utilisé. Donc, si A est un registre,

ASSIGN H,A<—1+A

et

ASSIGN H,1+A—>A

ne donnent pas le même résultat car dans le premier cas le registre A contient 2

après les opérations

A := 1
A := A + A

et dans le deuxième, A est incrémenté de 1 après les opérations

RT := 1
RT := RT + A
A := RT

où RT est le registre de travail du système. En plus, comme il n'existe pas d'opérations machine sur des octets en 360, ces opérations sont interdites dans le système.

Un autre type de restriction est dû aux limitations de l'assembleur 360. Dans certaines conditions, il n'est pas possible de trouver le type d'un opérande. Donc, la convention suivante a été adoptée :

- les opérandes de la forme

NOM(BASE)

ou

NOM(BASE,INDEX)

sont des mémoires ;

- les opérandes sans parenthèses commençant avec I sont des constantes immédiates (déclarées avec EQU) ;
- les opérandes commençant par K sont des constantes dans le programme (déclarées avec DC ou DS) ;
- les opérandes commençant par d'autres lettres sont des registres.

Ces conventions ne sont pas idéales et il serait peut-être préférable de les remplacer par des macros de déclaration.

2.6. Remarques Finales

Ces projets de définition de langages ou d'extensions de langages ont des buts extrêmement différents. Mais ils sont similaires en ce qu'ils répondent tous un besoin précis et sont tous basés sur l'expérience. Notre opinion est que les changements et les améliorations dans ce domaine ne doivent se faire que lentement. Il est malheureusement courant que des essais de définition d'énormes structures linguistiques sur des bases nouvelles conduisent à utiliser des langages ne possédant pas une marge suffisante de sécurité et dans lesquels de belles idées sont souvent perdues par manque de clarté dans les principes. On progresse mieux en amé liorant les idées courantes, en supprimant des parties mauvaises, en les remplaçant par de meilleures et en introduisant lentement les nouvelles facilités lorsque nécessaire.

Il est également important qu'un nouveau langage se présente comme un "triplet" formé d'une définition rigoureuse faite à l'aide d'une syntaxe, d'une défi nition "populaire" avec des exemples, et d'une implémentation complète. RTL est un exception parmi nos projets en ce qui concerne ces critères, mais il est possible q s'il existait des utilisateurs et des compilateurs, on y trouverait des erreurs et des imprécisions. Une certaine expérience dans l'écriture de la documentation des langages [3], [31] nous a convaincu de la difficulté d'obtenir la précision requis Le rapport ALGOL 60 [5], qui était considéré un chef d'oeuvre lors de sa publicatio contient néanmoins une démonstration de la vérité de ces remarques (voir [32] qui donne aussi d'autres références). Il faut dire qu'en dépit de nos ambitions limité dans ces projets, ils n'échappent pas à ces critiques.

CHAPITRE III

THEORIES SYNTAXIQUES

3. THEORIES SYNTAXIQUES

La compilation moderne est généralement inspirée d'idées syntaxiques plus ou moins rigoureuses. L'utilisation d'un formalisme exact et général augmente la sécu rité d'un compilateur et le rend plus facile à écrire, à comprendre et à transférer d'une machine à une autre. Malheureusement, il n'est pas possible, actuellement, d généraliser ce formalisme de sorte que tout compilateur dérive directement des défi nitions du langage et de la machine. Parmi les raisons empêchant ce formalisme com plet, on trouve les limitations de nos connaissances actuelles du sujet et une pert d'efficacité dans les produits résultants. On verra plus tard que l'effort nécessa pour rendre plus efficace les théories existantes peut mener à d'autres théories pl puissantes.

Il existe plusieurs définitions du mot "syntaxe". Celle que nous utilisons ici est la suivante : la syntaxe est la partie formelle de la définition d'un langage. Il est souvent commode de traiter sémantiquement certaines notions d'un langage alo qu'elles pourraient être traitées syntaxiquement ; par suite, la séparation de la s taxe et de la sémantique est parfois aléatoire. De toute façon, notre choix est limité aux concepts pouvant être exprimés sous la forme normale de BACKUS présentée dans le rapport ALGOL 60 [5] .

3.1. Analyse Prédictive avec Génération d'un Arbre

L'organisation de FROLIC ressemble à celle des travaux grenoblois de L. BOLLIET et son équipe [33] , [34] , bien qu'elle ait été développée indépendémme à la même époque. Dans ces compilateurs, l'analyse syntaxique comprend un passage du texte et est utilisée pour créer un arbre syntaxique du programme source. Cet arbre est transformé, puis utilisé pour générer du code machine. Les détails des

algorithmes d'analyse varient avec les projets. A Grenoble on pratiquait plutôt une analyse de précédence qui est décrite par A. COLMERAUER [35] tandis que FROLIC utilisait l'analyse prédictive décrite par S. GREIBACH [36] .

L'analyse prédictive est peut-être la plus simple des méthodes d'analyse syntaxique. C'est une méthode descendante dans laquelle toutes les possibilités futures sont retenues en même temps. Un exemple le montre plus simplement qu'une description formelle. Considérons la grammaire d'ALGOL 60 :

```
1    Bloc  —> début LD ; LI fin
2    LD    —> D
3             D ; LD
4    LI    —> I
5             I ; LI
```

où LD, LI représentent respectivement liste de déclarations et liste d'instructions. Une chaîne d'entrée possible est

début D ; D ; S ; S fin

On considère, pour le moment, que déclaration et instruction sont des symboles de base (symboles terminaux) dans le langage. Puisque tout programme ALGOL est un bloc, on commence l'analyse avec le but

Bloc

sur une pile. Dans l'état normal, la tête de pile est comparée au symbole courant. Ici, Bloc et début ne sont pas comparables, car Bloc est un nom de classe (symbole non-terminal). Bloc est développé et la pile devient

début LD ; LI fin

Maintenant, le début dans la chaîne d'entrée est éliminé avec le début sur la pile pour laisser

LD ; LI fin

comme cible, et

D ; D ; I ; I fin

comme chaîne d'entrée. On a encore un nom de classe en tête de pile, qui se développe en :

D ; LI fin

et

D ; LD ; LI fin

Ces deux piles proviennent du fait que LD a deux expansions possibles. Maintenant D et ; peuvent être éliminés, laissant les piles

LI fin

et

LD ; LI fin

L'expansion de LI et de LD donnent quatre piles :

I fin
I ; LI fin
D ; LI fin
D ; LD ; LI fin

mais les deux premières sont éliminées car le caractère d'entrée est D et non I. Sur les deux autres piles D ; peut encore disparaître, laissant,

LI fin
LD ; LI fin

comme avant, mais avec la chaîne d'entrée raccourcie à

I ; I fin

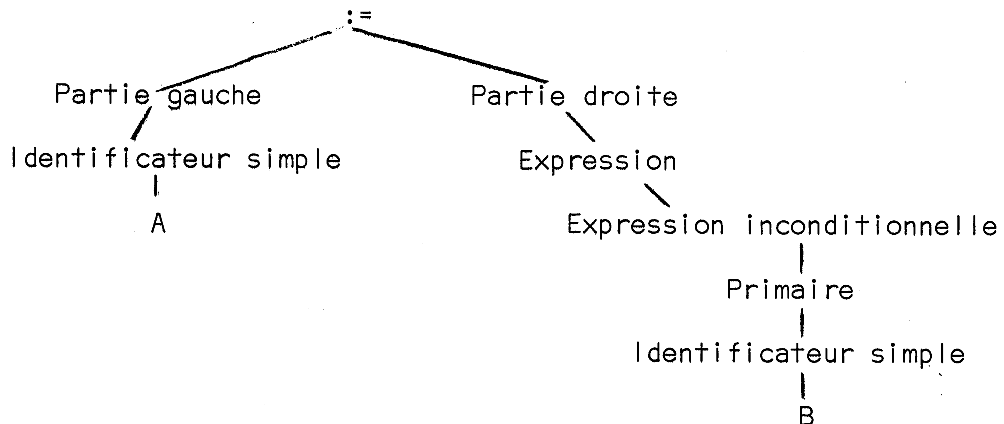
Ce processus continue jusqu'au point où la chaine d'entrée est vide. Si le dernier symbole élimine une seule pile cible, l'analyse qui a mené à cette pile représente l'analyse correcte du programme. Si le dernier symbole élimine plus d'une pile cible, une analyse ambiguë a été trouvée. Si toutes les piles sont éliminées avant la fin de la chaine d'entrée, une erreur existe dans cette chaîne.

Pendant cette analyse, on garde, pour chaque pile cible, les numéros des règles de la grammaire qui sont appliquées pour obtenir cette pile. Un arbre syntaxique du programme est créé à partir de la liste des numéros des règles correspondant à l'analyse du programme.

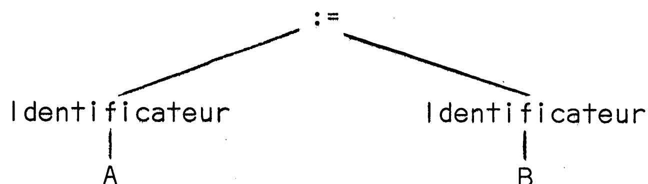
En général, l'arbre résultant de ce processus est très étendu et contient de l'information redondante. Par exemple, l'affectation

A := B

devient en arbre



On aurait préféré l'arbre



En FROLIC, la conversion des numéros des règles en arbre est faite directement dans cette forme condensée, qui n'est pas un vrai arbre syntaxique du programme.

3.2. Transformation d'une Grammaire et Déterminisme

L'analyse prédictive est extrêmement coûteuse en temps et en place mémoire quand elle est faite à partir d'une grammaire écrite normalement. Une des raisons de ce coût est apparu ci-dessus : après la première expansion de LD, on avait deux piles cible possibles

D ; LI fin

et

D ; LD ; LI fin

La déclaration est analysée une fois sur chaque pile, avec une perte correspondante d'efficacité. Ce phénomène est particulièrement frappant dans les expressions, où les huit niveaux différents d'opérateurs peuvent mener à 256 piles différentes avant de montrer qu'un identificateur est une expression.

Des expériences faites à l'aide de l'analyseur de FROLIC montraient qu'une réécriture de la grammaire peut améliorer d'un facteur de dix les temps d'analyse d'ALGOL 60 et aussi fournir un gain sensible de place. Nous donnons comme exemple la réécriture de la grammaire d'ALGOL 60 déjà donnée.

```
Bloc —> début LD ; LI fin
LD   —> D  X
X    —> Ø
         ; LD
LI   —> I  Y
Y    —> Ø
         ; LI
```

Le symbole ∅ représente l'expansion vide. Avec cette grammaire les piles successives dans l'analyse de notre chaîne d'entrée

début D ; D ; I ; I fin

sont

Bloc
début LD ; LI fin
LD ; LI fin
D X ; LI fin

et maintenant la déclaration n'est analysée qu'une fois avant de décider si l'élément suivant est une déclaration ou non.

La grammaire d'ALGOL 60 a été réécrite à la main pour FROLIC. Parallèlement à ce travail, J.M. FOSTER, qui dirigeait le projet, découvrait qu'il est possible d'écrire une grammaire d'ALGOL 60 pour laquelle il n'y a jamais plus d'une pile cible entre la lecture de deux caractères pendant l'analyse. Une grammaire de cette forme s'appelle une grammaire déterministe ou LL(1). On a démontré depuis que l'on peut trouver des grammaires LL(1) pour la plupart des langages de programmation utilisés, bien que, en certains cas, il faille trouver des algorithmes particuliers [14] . Deux autres avantages du déterminisme sont à noter : il est possible de créer une autre forme d'analyseur et il devient plus simple de lier la partie sémantique d'un compilateur à sa partie syntaxique.

FOSTER a écrit un programme [8] qui transforme une grammaire en une nouvelle forme, vérifie que cette nouvelle forme répond aux critères de déterminisme et produit un analyseur. Ce programme a montré son utilité pour la production de compilateurs pour CORAL 66. Il était donc naturel de continuer à développer la méthode pour travailler sur PL/I. Une nouvelle version de ce programme a été écrite [29] , [37], qui utilise, en plus des idées de FOSTER, des tests de déterminisme indiqués par KNUTH [13] et qui crée un analyseur en macros [9] .

3.3. Production d'un analyseur

Avec une grammaire déterministe, il est possible de produire un analyseur sous la forme d'un ensemble de procédures, chaque procédure étant l'expansion d'un nom de classe. La définition du déterminisme que nous utiliserons est que, pour chaque nom de classe, le caractère d'entrée courant suffit pour choisir l'expansion de ce nom de classe qui doit s'appliquer. Donc, chaque procédure est de la forme :

```
Décision entre les différentes expansions selon le caractère courant
        L1 :  Première expansion
              .
              .
              .
              EXIT
        L2 :  Deuxième expansion
              .
              .
              .
```

Dans le programme réalisé à Grenoble, les procédures sont des routines écrites dans le langage de macros. Une macro spéciale est utilisée pour les décisions. Elle a la forme :

```
DECIDE symbole-de-base,étiquette
```

DECIDE provoque un saut à l'étiquette indiquée si le caractère courant est le symbole de base indiqué. Chaque alternative consiste en une série de CALL, pour les symboles qui sont des noms de classe, et de CHECK, pour les symboles qui sont des symboles de base. CHECK est une macro qui confirme la présence du symbole indiqué et place le prochain caractère comme caractère courant. Comme exemple, prenons la forme déterministe de la grammaire déjà utilisée.

```
Bloc —> début LD LI fi
LD   —> D ; X
X    —> Ø
        LD
LI   —> I Y
Y    —> Ø
        ; LI
```

L'analyseur correspondant est

```
          ROUTINE BLOC
               CHECK 'DEBUT'
               CALL LD
               CALL LI
               CHECK 'FIN'
          RETURN
          ROUTINE LD
               CALL D
               CHECK ';'
               CALL X
          RETURN
          ROUTINE X
               DECIDE 'REEL',L1
               DECIDE 'ENTIER,L1
               :
               EXIT
L1             CALL LD
          RETURN
          ROUTINE LI
               CALL I
               CALL Y
          RETURN
```

```
          ROUTINE Y
               DECIDE ';',L2
               EXIT
L2             CHECK ';'
               CALL L1
          RETURN
```

Les routines sont appelées récursivement pendant l'analyse.

Une autre solution diminuant d'un facteur de deux la place en mémoire au prix d'une légère perte de temps est d'interpréter une table qui contient la même information sous forme codée [38] .

3.4. Théorie du Déterminisme

Les critères de FOSTER sont donnés sous une présentation formalisée dans KNUTH [13] , qui les a retrouvés de manière indépendante. Nous reprenons ici ces critères, mais avec une formalisation nouvelle qui dépend de la définition du déterminisme donnée au paragraphe précédent.

On a vu qu'il doit être possible de choisir l'expansion d'un nom de classe en considérant seulement le caractère courant. Nous voulons montrer que si cette condition s'applique à une grammaire donné l'analyseur produit à partir de celle-ci déterminera, dans un temps fini, si une chaîne de longueur finie appartient ou non au langage décrit par cette grammaire.

Une grammaire G est un quadruplet $\{A, R, V_T, V_N\}$ dans lequel V_T (les symboles de base) et V_N (les noms de classe) sont des ensembles de symboles, A est l'axiome et R un ensemble de règles. Les interactions entre les membres du quadruplet sont définies par :

1. $A \in V_N$
2. $V_T \cap V_N$ est nulle
3. Tout $v \in R$ a la forme
   $B \longrightarrow S_1 \mid S_2 \mid \ldots \mid S_n$ (entier $n \geq 1$)

où $B \in V_N$

$S_1, S_2 \ldots S_n$ sont définis comme les <u>dérivations immédiates</u> de B

$\forall i$ pour lequel S_i, $S_i = C_{i_1} C_{i_2} \ldots C_{i_{p_i}}$ (entier $p_i \geq 0$)

$\forall i,j$ pour lesquels C_{i_j}, $C_{i_j} \in V_N \cup V_T$

4. $\forall p \in V_N \; \exists$ une et seulement une règle $v \in R$ avec p en partie gauche.
5. On définit une <u>dérivation</u> d d'un $B \in V_N$ comme suit :

   Toute dérivation immédiate de B est une dérivation de B.

   Soit

   $C_1 C_2 \ldots C_n$

une dérivation de B (entier $n \geq 0$). Toute chaîne obtenue par remplacement de $C_i \in V_N$ par une dérivation de C_i est une dérivation de B. On appliquera à une grammaire la restriction suivante :

$\forall B \in V_N$,

les dérivations de B doivent finalement donner une chaîne d telle que

$d = C_1 C_2 \ldots C_n$

avec soit $n = 0$

soit $C_i \in V_T \; \forall i, \; i = 1 \ldots n$.

Etant donné une grammaire de cette forme, l'analyseur est produit comme suit :

Chaque règle correspond à une ROUTINE de la forme :

ROUTINE B

Décision entre les S_i

$f(C_{11})$

$f(C_{12})$

⋮

$f(C_{1_{p_1}})$

EXIT

$f(C_{2_1})$

⋮

$f(C_{2_{p_2}})$

EXIT

⋮

$f(C_{N_{p_n}})$

RETURN

où $f(C)$ = CALL C si $C \in V_N$

ou CHECK C si $C \in V_T$

et les C_i sont définis comme ci-dessus. Si la grammaire est déterministe, la décision entre les S_i pour une règle B quelconque ne dépend que du caractère courant. Les caractères qui provoquent le choix d'un S_i sont appelés les symboles directeurs pour cet S_i. Plus précisément, un symbole directeur de S_i est un caractère qui pourrait être le paramètre du premier CHECK rencontré si S_i est l'expansion de B qui est appliquée. Une grammaire est déterministe

si, pour chaque B, aucun caractère n'est un symbole directeur pour plus d'un S_i. A partir de cette définition, nous allons prouver que toute chaîne finie est analysée dans un temps fini, ce qui revient à démontrer qu'il n'est pas possible d'entrer une routine deux fois sans lire un caractère de la chaîne.

Considérons l'hypothèse contraire :

Si, commençant avec un caractère courant au début d'une routine, on revient à ce point sans avoir lu un caractère, un des deux cas s'est présenté :

- aucun des noms de classe par lesquels l'analyseur est passé n'avait plus d'une expansion. La séquence de règles correspondantes est donc

$$A_0 \longrightarrow A_1B_1$$
$$A_1 \longrightarrow A_2B_2$$
$$\vdots$$
$$A_n \longrightarrow A_0B_n$$

où B_i est une chaîne quelconque sur $V_T \cup V_N$ et aucun des A_i n'a une autre expansion. Une telle série de règles ne remplit pas la condition 5 ci-dessus, car les A_i n'ont pas de dérivation sur $V_T \cup \emptyset$.

- un des noms de classe A_p avait au moins deux expansions. La règle correspondant à A_p est

$$A_p \longrightarrow A_{p+1}\ B_{p+1}\ |\ C$$

Or, dans ce cas, tout symbole directeur pour l'expansion

$$A_p \longrightarrow C$$

qui n'a pas été appliquée, est un symbole directeur pour l'expansion

$$A_p \longrightarrow A_{p+1}\ B_{p+1}$$

car avec ce symbole comme caractère courant, l'analyseur peut toujours appeler les A_i jusqu'au point où il revient à A_p, et puis appliquer l'expansion

$$A_p \longrightarrow C$$

Donc, les symboles directeurs pour les alternatives de A_p ne sont pas disjoints, et la grammaire ne remplit pas la définition du déterminisme.

Comme le nombre des routines est fini, le nombre d'opérations entre deux lectures de caractères est fini. Donc, une chaîne d'entrée qui est de longueur finie est entièrement lue à l'aide d'un nombre fini d'opérations. Par suite, l'analyse se termine toujours.

On a montré ci-dessus que l'analyseur donne toujours un résultat quand la grammaire est déterministe. Le fait que l'analyseur ne boucle pas est une démonstration de la remarque de KNUTH : si les trois dernières conditions du déterminisme sont vérifiées, il n'y a pas de récursivité à gauche.

On a établi ci-dessus la condition à ajouter à la définition d'une grammaire "context-free" pour que la grammaire soit déterministe. Une telle grammaire déterministe est appelée LL(1) par KNUTH. En général, si k caractères sont nécessaires pour prendre une décision en analyse descendante, on dit que la grammaire et le langage décrit sont LL(k). Inversément, si k caractères sont nécessaires pour prendre une décision en analyse ascendante, on dit que la grammaire et le langage sont LR(k). Nous utiliserons ces définitions pour déterminer les limites des méthodes décrites ici. On dispose des résultats suivants :

- tout langage LL(k) est un langage LR(k)
- tout langage LR(k) est un langage LR(1)
- il existe des langages LR(1) qui ne sont pas LL(k) pour un k fini.
- il existe des langages "context-free" qui ne sont pas LR(k) pour un k fini.

Le premier résultat ci-dessus est intuitif et KNUTH [13] suggère que la démonstration de ce théorème soit le sujet d'une thèse de Troisième Cycle. Nous nous limiterons donc à l'énoncé de ce principe. Le deuxième résultat est prouvé par KNUTH en [55] . Un exemple d'un langage LR(1) qui n'est pas LL(k) est donné en [13] :

B —> E = E | (B)
E —> a | (E + E)

Cette grammaire représente un sous-ensemble des expressions booléennes en ALGOL 60. Nous avons rencontré un cas analogue dans la grammaire de PL/I, où on a la syntaxe suivante pour une liste de données dans une instruction PUT :

Dataoutlist —> Dataoutélément
Dataoutélément,Dataoutlist
Dataoutélément —> Expression
(Dataoutlist)
(Dataoutlist DO Variable = Speclist)

Cette grammaire est ambiguë car une expression entre parenthèses est toujours une expression mais le langage n'est pas ambigu et il est LR(1) mais sans être LL(k). Dans les deux exemples, la difficulté provient de la double utilisation des parenthèses. Un exemple de langage "context-free" qui n'est pas LR(k) est donné par HOPCROFT et ULLMAN [58] :

A —> aAb | ab | Bc
B —> aBbb | abb

On ne sait pas encore si tout langage LL(k) est LL(1), mais les résultats ci-dessus entrainent certaines conclusions. Les langages LL(1) forment le plus petit ensemble considéré. En fait, nous ne connaissons aucun langage de programmation dont la grammaire ne soit pas transformable sous une forme LR(1). Par contre, ni ALGOL ni PL/I ne sont exprimés par une grammaire décrivant un langage LL(1). Il est à noter qu'on ne considère pas les propriétés des langages ALGOL et PL/I eux-mêmes, car ni l'un ni l'autre

ne sont des langages "context-free", mais seulement les sur-ensembles de ces langages décrits par les grammaires "context-free" qui en constituent les descriptions formelles.

Il est bon de poser la question : "Pourquoi utilise-t-on des grammaires LL(1)?". Intuitivement, une analyse déterministe est plus efficace qu'une analyse non déterministe, de même qu'une analyse descendante est plus rapide qu'une analyse ascendante, ayant le même nombre d'opérations, car il est plus simple de trouver la partie gauche d'une règle que la partie droite. L'analyse à l'aide d'une grammaire LL(1) offre donc une grande efficacité au prix d'une réduction de l'ensemble des langages pouvant être traités. Or, dans la pratique, la restriction n'est pas grave, car les difficultés peuvent être résolues par d'autres méthodes. De plus, comme nous le verrons plus tard, l'analyse par une grammaire LL(1) permet un traitement correct de la sémantique et la grammaire est en général plus simple à écrire qu'une grammaire LR(1) pour le même langage [59] . Notons néanmoins que le choix de la méthode n'est pas évident, et que d'autres personnes ont choisi une grammaire LR(1) [61] .

Ce travail est poursuivi par J. BORDIER [40] et M. ASSABGUI [39] . La prochaine section va montrer comment la définition du déterminisme est interprétée en pratique.

3.5. Algorithme de Transformation et Test

Le transformateur de grammaire est un programme conséquent dont la partie la plus importante est celle qui teste le déterminisme d'une grammaire. Au préalable, le transformateur a lu les données, construit en parallèle le dictionnaire des identificateurs et écrit les règles sous forme de listes ; puis, des transformations triviales

réalisent les factorisations évidentes (avec contexte limité à un nom de classe). Par exemple :

A —> bCD | bCE | bF | G

est transformée en

A —> bX | G
X —> CY | F
Y —> D | E

Par contre

A —> bC | D
D —> bE | F

ne sont pas transformées.

Cette nouvelle grammaire est soumise à une série de tests pour vérifier les restrictions 1 à 5 et le critère de déterminisme. En fait, les conditions 1 à 4 sont vérifiées pendant la lecture des règles par de simples tests sur le dictionnaire des noms.

Ensuite, on détermine si un nom de classe peut ou non avoir une dérivation vide. L'algorithme est le suivant : un vecteur V contient une entrée pour chaque nom de classe. Les entrées peuvent prendre les valeurs

OUI (le nom de classe peut avoir une dérivation vide)
NON (le nom de classe ne peut pas avoir de dérivation vide)
INDECIDE

V est initialisé à INDECIDE. Puis on considère les dérivations immédiates pour chaque nom de classe. Il y a trois cas :

- une des dérivations immédiates est vide. L'entrée correspondante dans V est marquée OUI, et ce nom de classe n'est plus examiné.

- toutes les dérivations immédiates contiennent au moins un symbole de base. L'entrée dans B est marquée NON, et le traitement de ce nom de classe est terminé.

- autrement, on garde seulement les dérivations immédiates ne contenant que des noms de classe.

Le processus suivant est alors appliqué - autant de fois que nécessaire - en considérant les noms de classe INDECIDE. Pour chaque dérivation immédiate, il y a trois cas :

- un des noms de classe dans cette dérivation immédiate contient déjà la réponse NON : la dérivation immédiate est supprimée. Si toutes les dérivations d'un nom de classe sont supprimées de cette façon, ce nom de classe prend la valeur NON.

- un des noms de classe est déjà à OUI. Ce nom de classe est supprimé de la dérivation. Si une dérivation devient vide de cette façon, l'entrée dans V prend la valeur OUI.

- si rien ne peut être fait on passe au nom de classe suivant.

Si, au cours d'un passage dans la boucle, rien n'est changé dans V, la grammaire ne remplit pas la condition 5, car il y a au moins un nom de classe avec une dérivation qui ne donne jamais une chaîne terminale.

Il est à noter que la condition 5 n'est pas complètement testée par le programme. Si on s'assure qu'un caractère d'entrée est toujours lu dans un temps fini et que l'analyseur peut toujours décider quelle action prendre, une grammaire comme

A —> aB
B —> bA

qui ne produit aucune chaîne finie, est acceptable, car elle donne une réponse négative en un temps fini pour toute chaîne de longueur finie. Il est possible d'améliorer ce programme en vue de détecter de telles erreurs, ainsi que les symboles inaccessibles et parasites (nomenclature de MARTIN [41]) .

Les tests qui trouvent les symboles directeurs pour la macro DECIDE sont faits à l'aide de deux matrices de bits. La première matrice a une ligne par nom de classe avec deux champs, l'un pour les symboles de base et l'autre pour les noms de classe. La deuxième matrice a une entrée par nom de classe, avec trois champs, le premier pour les symboles de base et les deux autres (identiques) pour les noms de classe. Elle est utilisée pour trouver les successeurs (c'est-à-dire les caractères qui peuvent suivre un nom de classe dans une dérivation quelconque). Ces matrices sont initialisées comme suit :

Les règles sont balayées et les cas suivants sont considérés :

- les caractères débutant chaque dérivation immédiate d'un nom de classe sont marqués dans la matrice des débutants. Si un débutant est un nom de classe qui peut avoir une expansion vide, le caractère suivant est aussi marqué, et ainsi de suite. Les noms de classe qui peuvent donner une expansion vide sont déjà marqués en V.

- si, dans une dérivation, un nom de classe est suivi d'un caractère, ce caractère est marqué dans la matrice des successeurs.

- si un nom de classe A est le dernier caractère dans une dérivation immédiate de B, tous les successeurs de B sont aussi successeurs de A. Donc, ce nom de classe est marqué dans le troisième champ de la matrice des successeurs.

Ces deux matrices sont maintenant transformées pour calculer les symboles de base qui peuvent être les débutants et les successeurs d'une dérivation éventuelle. On décrit en premier lieu la transformation sur les débutants. Si un nom de classe A est un débutant d'un nom de classe B, tout débutant de A est un débutant de B. Donc, la ligne B est remplacée par l'union des lignes A et B. Ce processus est fait pour tout nom de classe qui est un débutant de B, y inclus ceux produits par la promotion des débutants d'un autre nom de classe, mais il n'est pas répété pour un nom de classe qui a déjà été traité dans cette ligne. Après un nombre fini d'opérations,

on obtient tous les symboles de base débutants. Si au cours de ce processus on retrouve A comme débutant de A, l'analyseur contient une boucle (récursivité à gauche) ce qui est interdit. Dans le chapitre précédent, il a été prouvé qu'une telle boucl contredit soit la condition 5, soit le critère de déterminisme.

Si un nom de classe A est un successeur de B, tout débutant de A est un successeur de B. La promotion décrite ci-dessus est faite à partir de la matrice de débutants vers la matrice des successeurs, et ceci d'une manière directe puisque l'o dispose de tous les symboles de base débutants. Pour le troisième champ dans la mat rice des successeurs, cette promotion est faite à partir de la matrice des successeu vers la matrice des successeurs.

Maintenant, seuls les symboles de base débutants et successeurs nous intéressent. Ils sont utilisés pour calculer les symboles directeurs pour chaque dérivation immédiate comme suit :

Considérons une expansion

$$A \longrightarrow B$$

où B est une chaîne sur $V_N \cup V_T \cup \emptyset$

- si pour B il n'existe aucune dérivation vide, les symboles directeurs pour B sont les débutants de B, trouvés dans la matrice.

- s'il existe une dérivation vide de B, les symboles directeurs pour B sont les débutants de B plus les successeurs de A.

Avec les ensembles de symboles directeurs, on peut tester que les ensembles correspondants aux dérivations immédiates d'un nom de classe sont disjoints et donc produire un analyseur déterministe.

Le transformateur accepte une grammaire sous une forme quelconque et tou exception aux spécifications sera signalée par un message d'erreur.

3.6. Génération Téléscopique

La méthode utilisée pour transférer CORAL 66 d'une machine à une autre ("bootstrapping") présente un certain intérêt. Sur notre machine de base R, il y avait un compilateur ALGOL, écrit évidemment dans le code machine de R. On représente ce compilateur par un triplet

$$ARR$$

En général, le triplet

$$PQR$$

représente un compilateur traduisant le langage source P dans le langage objet Q et écrit lui-même en langage R. Le compilateur CORAL 66 produit du code de la nouvelle machine X et est écrit dans un sous-ensemble commun de CORAL 66 et d'ALGOL. Appelons ce sous-ensemble C_A ; on aura donc un

$$CXC_A$$

Comme C_A est un sous-ensemble de A, le CXC_A peut être compilé par le compilateur ARR sur la machine R. On représente ce processus de la manière suivante :

$$CXC_A \;.\; ARR \;.\; R$$

La simplification de cette formule donne (en se rappelant que $C_A \subset A$)

$$CX\not{C}_A \;.\; \not{A}R\not{R} \;.\; \not{R} \longrightarrow CXR$$

Le CXR est un compilateur écrit en code machine R qui traduit CORAL 66 en code machine X. Ce nouveau CXR est utilisé pour recompiler le CXC_A et on obtient

$$CXC_A \;.\; CXR \;.\; R$$

qui donne par simplification $(C_A \not\subset C)$

$$CX\not{C}_A \;.\; \not{C}X\not{R} \;.\; \not{R} \longrightarrow CXX$$

Le CXX est un compilateur de CORAL 66 en X écrit en X. Donc, on a un compilateur pour la machine X sans jamais l'utiliser.

En fait, ce formalisme, dû à P.M. WOODWARD, peut représenter n'importe quel programme qui est compilé. On a un programme P qui traduit les données D en un résultat R, et qui est écrit en langage L. Donc,

$$P = DRL$$

Un compilateur pour L est écrit en langage machine M et traduit vers M. Le compilateur est un

$$LMM$$

La compilation est faite sur M

$$DRL \circ LMM \circ M$$

La simplification donne

$$DR\not{L} \circ \not{L}M\not{M} \circ \not{M} \longrightarrow DRM$$

qui est un programme transformant les données en résultats, écrit en M et qui marche sur M par présentation des données.

$$\not{D} \circ \not{D}R\not{M} \circ \not{M} \longrightarrow R$$

On n'obtient que les résultats.

Ce formalisme nous a permis de transférer aisément des compilateurs. Evidemment, entre les deux compilateurs de CXC_A, les macros d'analyse produites par le transformateur de grammaire devaient être changées du langage R en langage X. Ce changement à la main ne prend qu'une journée de travail. En fait, le compilateur pour CORAL 66 fait deux passages du texte source. Le code intermédiaire a une syntaxe et les deux passages ont été produits à l'aide de ce processus. Donc, on a écrit un traducteur de CORAL 66 en code intermédiaire (un CC_IC_A) et un traducteur du code intermédiaire en code machine X (un C_IXC_A). Les équations utilisées sont

$CC_IC_A \; . \; ARR \; . \; R \longrightarrow CC_IR$

$C_IXC_A \; . \; ARR \; . \; R \longrightarrow C_IXR$

$CC_IC_A \; . \; CC_IR \; . \; R \longrightarrow CC_IC_I$

$CC_IC_I \; . \; C_IXR \; . \; R \longrightarrow CC_IX$

$C_IXC_A \; . \; CC_IR \; . \; R \longrightarrow C_IXC_I$

$C_IXC_I \; . \; C_IXR \; . \; R \longrightarrow C_IXX$

et on a les deux passages, qui sont le CC_IX et le C_IXX.

Ce travail a été continué sur d'autres machines en vue d'autres applications. L'état actuel des travaux est décrit en [49] .

CHAPITRE IV

OUTILS SEMANTIQUES

4. Outils Sémantiques

Si la partie syntaxique d'un compilateur est la plus intéressante, et par définition la plus formalisée, la partie sémantique est la plus difficile et peut-être la plus importante. Quand la syntaxe est formalisée, il est possible de lui apporter des modifications et de vérifier que celles-ci n'entraînent pas d'erreurs par ailleurs. Ceci n'est absolument pas vrai vis-à-vis de la sémantique.

Jusqu'à présent, il existe peu d'outils généraux pour la sémantique. La plupart des algorithmes sont des "recettes de cuisine", transmises de bouche à oreille comme un folklore ancien. Le sujet est difficile à traiter d'une façon concise et nous ne donnerons ici qu'une description des plus importants algorithmes.

4.1. Synchronisation avec la Syntaxe

Il y a quelques années, les compilateurs comportaient une phase sémantique distincte de la phase d'analyse syntaxique. L'information était transmise entre ces phases à l'aide d'un arbre. Dans les analyseurs produits automatiquement, de la façon décrite ci-dessus, on utilise une méthode qui est plus souple et plus efficace.

Dans la grammaire originale, présentée au transformateur, il est permis d'inclure des noms qui ne sont ni des symboles de base ni des noms de classe mais des appels à des fonctions sémantiques, chacune étant une ROUTINE écrite à la main. Quand l'analyse rencontre un de ces noms, la ROUTINE correspondante est appelée. Donc, la phase sémantique est imbriquée dans l'analyse syntaxique. Cette facilité est un sous-produit du déterminisme, car l'analyse réalisée est nécessairement la vraie (il n'y a aucun retour en arrière). Les fonctions sémantiques sont

- transformées comme des symboles de base
- transparentes pour les tests
- appelées comme des noms de classe.

Le langage de macros est utilisé pour l'écriture de ces fonctions sémantiques. Nous donnons comme exemple la lecture d'une entier par la grammaire

```
Entier —> Digit f1 X
X      —> Ø
          Digit f2 X
```

Supposons que la valeur d'un digit se trouve en D(R1) après sa lecture. On a les ROUTINES suivantes :

```
ROUTINE F1
        ASSIGN H,RES(R1)<—D(R1)
RETURN
ROUTINE F2
        ASSIGN H,RES(R1)<—RES(R1)*10+D(R1)
RETURN
```

La valeur de l'entier se trouve en RES(R1). Ces facilités nous ont permis d'écrire des compilateurs dans un temps réduit. En particulier, les deux premiers compilateur pour CORAL 66 ont été écrits en six mois avec deux personnes.

4.2. Gestion de Listes

Il est souvent nécessaire, dans l'écriture des compilateurs, de manipuler des données structurées. Depuis le papier de McCARTHY [42] , on a toujours utilisé LISP ou une dérivation de LISP. La dérivation particulière utilisée pour l'écriture de FROLIC est décrite en [43] et [44] . Dans ce système, les fonctions de base

HD, TL et CONS

sont des appels à des fonctions de la bibliothèque de procédures ALGOL. Il n'y a aucune déclaration du type list, les variables ayant le sens de pointeur étant déclarées de type integer. Elles sont indiquées au système par leur présence comme paramètre de la fonction SETUP. Le système garde cette information pour la récupération

de la place mémoire non utilisée ("garbage collection"). Ceci est fait par la procédure CONS quand il n'y a plus de place disponible.

Un système de listes de ce type serait avantageusement complété par les déclarations <u>list</u> et <u>table</u>. Pour faire ceci, il est normalement nécessaire de l'avoir prévu au moment de l'écriture du compilateur. Le langage FROLIC contient ces declarations, qui augmentent considérablement la sécurité des programmes utilisant des listes.

Un autre inconvénient du système utilisé originalement est que pour écrire des procédures de bibliothèque, il fallait quand même connaître comment marche le compilateur. Ceci est évité en CORAL 66. Dans ce langage les listes ne se trouvent pas dans les spécifications pour l'utilisateur, mais elles étaient nécessaires pour le processus de génération téléscopique. Nous avons utilisé un système de listes écrites complètement en ALGOL dans lequel les pointeurs sont toujours des entiers mais les références aux éléments se font par des tables. Donc, au lieu d'écrire

HD(LIST) ou TL(LIST)

on écrit

HD[LIST] ou TL[LIST]

CONS est toujours une procédure qui récupère la place libérée, mais qui est écrite en ALGOL. Les listes sont distinguées par leur apparition à l'intérieur d'une procédure spéciale comme suit :

```
integer      LIST1, LIST2, ... LISTN ;
procédure    DECLARELISTS (P) ;
procédure    P ;
begin        P(LIST1) ;
             P(LIST2) ;
             .
             .
             .
             P(LISTN)
end DECLARELISTS ;
```

Ce système a aussi été utilisé à Grenoble et une description se trouve en [45]. Il est analogue à la réalisation de J. COHEN [46] , qui le présente à l'utilisateur d'une autre façon.

Dans le langage de macros, l'implantation des listes est laissée aux bons soins de l'utilisateur, qui peut néanmoins suivre la méthode des compilateurs incrémentiels. Dans ces compilateurs, aucune liste ne peut être gardée entre deux incréments. Donc, l'espace liste est réinitialisée à chaque fois et il n'y a pas besoin de récupérer de l'espace. Les macros ont la forme évidente :

```
HD   OP1,OP2           (OP2 := HD [OP1])
TL   OP1,OP2
CONS OP1,OP2,OP3       (OP3 := CONS(OP1,OP2))
```

La méthode utilisée est celle des vecteurs HD, TL et les opérandes OP1, OP2, OP3 sont les opérandes normaux du langage de macros.

CHAPITRE V

ALGORITHMES SEMANTIQUES

5. Algorithmes Sémantiques

La partie sémantique ayant le plus d'importance dans un compilateur est celle qui traite de l'allocation d'espace dans la mémoire pour les variables et la recherche des valeurs correspondantes. D'autres problèmes y sont associés, comme ceux du type des variables et de la portée des noms des variables. Ces problèmes sont moins évidents à résoudre quand l'exécution du programme est interprétative. On décrit ici des idées, en commençant par les conceptions classiques, sur les algorithmes qui traitent ces sujets dans les différents compilateurs.

5.1. Allocation dynamique classique

La structure de blocs introduite par ALGOL 60 a pour corollaire une allocation dynamique de la mémoire. La méthode la plus souvent utilisée est celle de la pile, qui a été présentée par DIJKSTRA [47] .

A l'entrée d'un bloc, l'espace nécessaire pour les déclarations simples est alloué, c'est-à-dire que le pointeur vers la dernière entrée du stack est augmenté par la quantité de mémoire correspondante. Cette quantité est calculée directement à la compilation. Une référence à une des variables simples est alors obtenue à l'aide du déplacement de la mémoire correspondante par rapport au début de la zone allouée pour le bloc. Ce déplacement est une constante qui peut apparaître dans le code objet. D'habitude, un registre index pointe vers le début de la zone de mémoire du bloc courant. Donc, une référence à une variable simple dans le bloc local peut se faire en une instruction. Par exemple, en 360, la partie adresse aurait une des deux formes :

Déplacement (Base)
Déplacement (Base,Index)

Les références aux variables non-locales sont plus difficiles. Une technique souvent utilisée est celle de positionner un registre sur le bloc adressé et d'utiliser ce registre de la même façon que le registre local. Le positionnement de ce registre peut prendre plusieurs instructions, car il faut trouver le bon niveau dans le stack. D'habitude, les entrées de blocs sont chaînées dans le stack, c'est-à-dire que chacune contient un pointeur vers la précédente. Dans le cas d'un nouveau bloc ouvert pour l'évaluation d'un paramètre effectif correspondant à un paramètre formel appelé par nom, un double chaînage est nécessaire, car cette évaluation se fait dans le contexte de l'appel de la procédure et non de sa déclaration. Le chaînage sert à trouver l'occurrence correcte du bloc recherché. La récursivité permise en ALGOL implique qu'il est possible d'avoir plusieurs entrées dans le stack pour un même bloc. Une solution, en vue d'économiser du temps, est de garder dans un tableau la plus récente entrée pour chaque bloc. Une méthode analogue a été utilisée par RANDELL et RUSSELL [48] avec DISPLAY. Une autre technique est d'avoir un registre qui pointe sur le bloc le plus externe, qui ne peut avoir qu'une entrée dans le stack. Ceci permet d'adresser les variables de ce bloc avec une instruction, ce qui représente un gain sensible, car les variables non-locales sont presque toujours déclarées globalement.

Pour les tableaux, il existe un autre problème, car leurs tailles ne sont pas nécessairement connues avant l'exécution. On réserve, comme pour une variable simple, l'emplacement pour un pointeur sur le stack pour chaque tableau. A la fin des déclarations des variables simples, les bornes de chaque tableau sont calculées et son encombrement est réservé. L'adresse du début du tableau est mise dans le pointeur, et toute référence à un élément est faite par un adressage indirect à l'aide de ce pointeur.

La place utilisée dans le stack pour les variables déclarées dans un bloc est récupérée en sortant du bloc. Les pointeurs au niveau actuel du stack et au début de la zone du bloc courant sont donc conservés à l'entrée de chaque bloc et leurs valeurs sont restaurées à la sortie du bloc. La zone dite "anonyme" est prise sur le stack et récupérée au fur et à mesure. La zone anonyme est celle qui est prise par le compilateur pour calculer, par exemple, les expressions et n'est pas declarée dans le programme.

5.2. Environnements

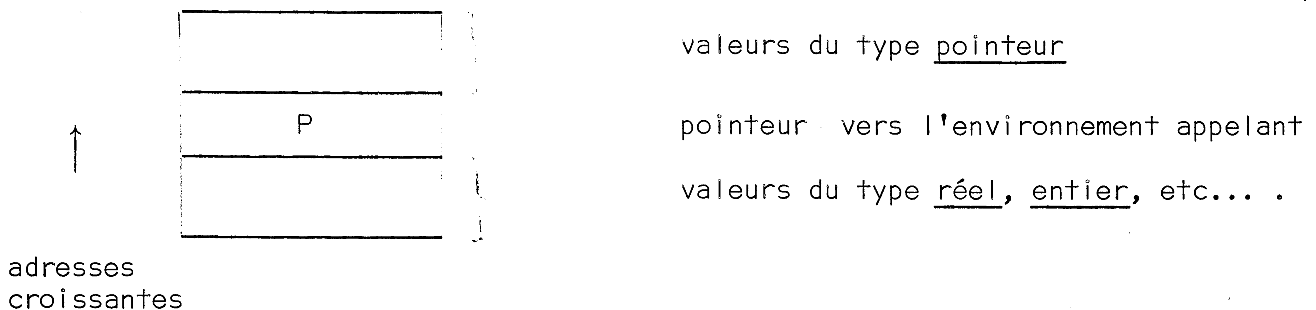
L'allocation de mémoire classique décrite ci-dessus est la base de l'allocation pour FROLIC, CORAL 66 et les compilateurs incrémentiels. Néanmoins, dans les trois cas, des changements importants ont dû être faits. L'allocation est particulièrement difficile pour FROLIC dans lequel le stack n'est utilisé que dans les programmes les plus simples. Une propriété de tout langage de simulation et de contrôle en temps réel est en effet que le stack ne suffit pas pour l'allocation, car dans de tels langages, le bloc dont on sort n'est pas nécessairement le dernier bloc entré. Considérons le programme :

```
begin procédure f ;
      begin procédure g ;
            begin
                 :
            end g ;
            real x ;
            :
            activate (g) ;
            :
      end f ;
      :
      activate (f) ;
      :
end
```

La procédure g est activée par la procédure f. Si f se termine avant g, ce qui est permis par le langage, la mémoire x serait perdue sur le stack avec la méthode traitionnelle. Or, on peut bien avoir des références à x dans g et un stack n'est donc pas utilisable dans ce cas.

On voit en effet que la récupération de l'espace mémoire qui n'est plus utilisée demande un algorithme plus sophistiqué que le simple stack. Elle est faite par un récupérateur (garbage collector) du type utilisé dans la programmation de listes. Dans le cas d'un programme sans tâches en parallèle ce récupérateur devient une pile.

Une zone de mémoire pour un bloc s'appelle l'environnement de ce bloc. Un environnement a la forme



Un pointeur vers un environnement pointe vers la mémoire P, qui contient l'information de chaînage. On appelle P l'adresse de l'environnement. Les valeurs simples ont des déplacements négatifs par rapport à l'environnement, les pointeurs ayant des déplacements positifs.

Pour chaque tâche active on garde l'adresse de l'environnement correspondant au bloc courant pour cette tâche. La liste des adresses de ces environnements est utiliséepour la récupération de la mémoire libérée de la même façon que sont utilisées les listes indiquées par SETUP dans le list-pack original.

Une valeur de type pointeur est une paire correspondant à la base et au déplacement d'une référence classique. La base est l'adresse de l'environnement concerné et le déplacement est le déplacement dans cet environnement. Chaque tableau déclaré a son propre environnement, les éléments étant tous d'un côté ou l'autre de P selon qu'ils sont du type pointeur ou non. Les listes utilisent un environnement particulier, pour lequel il y a une routine spéciale de récupération de place.

Les pointeurs sont utilisés pour récupérer l'information qui se trouve dans un autre environnement. Cette information peut être un élément d'un tableau, une variable non-locale, ou un élément d'une liste. Il est à noter que l'allocation à l'intérieur d'un bloc est la même que dans le système classique, c'est-à-dire par déplacement relatif à la base de la zone correspondante, ainsi que la méthode de référence. La différence réside dans la manière de distribuer les zones dans la mémoire, et de récupérer la place qui n'est pas utilisée.

L'algorithme de récupération de place prise par les environnements est bien connu. Tout environnement qui est l'environnement courant d'une tâche active est marqué (un bit de P est libre pour ceci). Puis tout environnement vers lequel il y a un pointeur dans un environnement marqué est aussi marqué, et ce processus est continué jusqu'à sa fin. La place de tout environnement qui n'a pas été marqué peut alors être récupérée, les environnements marqués étant retassés dans la mémoire. Pour éviter la mise à jour de plusieurs pointeurs vers chaque environnement retassé, il est possible de mettre un niveau d'adressage indirect dans les pointeurs, en gardant une table d'adresses d'environnements. Les adresses dans cette table sont mises à jour et les pointeurs dans les environnements réfèrent toujours à la table et restent donc inchangés. Ceci est encore équivalent au DISPLAY de RANDELL et RUSSELL.

Le système d'environnements est lourd à l'exécution en raison non seulement de la nécessité d'un algorithme de récupération de la place en mémoire, mais aussi de la complexité des références aux valeurs autres que les valeurs des variables dans l'environnement local. En particulier, le récupérateur de place serait inacceptable dans la plupart des ordinateurs traitant des problèmes en temps réel, car il immobilise la machine pendant un temps plus long que le temps de réponse nécessaire.

On aurait besoin d'améliorer la récupération de place si on voulait faire un compilateur efficace. Les techniques de pagination développées depuis éviteraient - au moins - le retassement des données en mémoire. Il est concevable que la méthode pourrait résoudre certains problèmes tel que celui du "tas" en ALGOL 68 et nous croyons fermement qu'elle mérite d'être mieux connue. Les premiers essais en vue d'implémenter ALGOL 68 [50], [51], [62] ont démontré le manque d'études sérieuses dans le domaine de l'allocation de mémoire, le papier de ROSS [52] faisant exception.

5.3. Allocation Statique pour CORAL 66

Le problème de l'allocation de mémoire en CORAL 66 est l'inverse du problème en FROLIC. CORAL 66 avait besoin, avant tout, d'être un langage efficace. Or, on a vu ci-dessus qu'une allocation dynamique de mémoire coûte toujours plus qu'une allocation statique. La comparaison de FROLIC avec ALGOL montre clairement le dilemme entre l'efficacité et la sophistication. Le choix d'un langage ne peut pas être absolu, mais reste un résultat des conditions dans lequelles le langage va être utilisé.

La définition de CORAL 66 est délibérément réduite aux facilités qui ne demandent pas d'allocation dynamique de mémoire. En particulier, les bornes des tableaux doivent être des constantes, la récursivité n'est pas admise et le nombre et le type des paramètres doivent être spécifiés pour tout paramètre formel de type procédure. L'intérêt n'est donc ni dans l'algorithme d'allocation, ni dans la méthode

de référence aux variables, qui sont tous les deux triviaux, mais dans les limitations qu'il faut mettre sur un langage pour éliminer ces problèmes. Pour les applications auxquelles CORAL 66 était destiné, il n'y avait pas de choix. Mais on verra que les compilateurs conversationnels pour des langages actuels sont d'une telle complexité qu'on ferait bien de considérer si on n'a pas intérêt à définir des langages avec ces limitations, pour des utilisateurs "amateurs" qui vont écrire leurs programmes en mode conversationnel. Le gros succès de BASIC [53] aux Etats-Unis confirme cette thèse, bien que BASIC soit probablement allé trop loin dans la voie de la simplicité. CORAL 66 provoque un choc aux puristes d'ALGOL, et il est indiscutable qu'il n'est pas adapté à la résolution de certains problèmes. Mais il est aisément utilisable pour la plupart des problèmes réels, pour lesquels les raffinements d'un langage trop sophistiqué ne sont qu'une charge improductive. Nous sommes convaincus que c'est une erreur de chercher à définir un langage universel de programmation.

5.4. Contrôle de l'Interprétation

L'interpréteur dans un système incrémentiel exécute dynamiquement des tâches faites par le compilateur dans un système batch. Le contrôle de l'interpréteur est donc important, car il faut bien séparer les fonctions variées du programme. Nous parlerons ici du compilateur ALGOL, mais les principes restent valables pour PL/I et FORTRAN. Supposons pour le moment qu'il n'y ait qu'un utilisateur du système.

A chaque segment correspond une table, appelée le dictionnaire de commande du segment, comprenant une entrée par incrément. Chaque entrée contient le type de l'incrément (BEGIN, Affectation, etc...), le numéro du prochain incrément et un pointeur vers le pseudo-code de l'incrément. Le dictionnaire est créé en même temps que le pseudo-code.

Avec ce dictionnaire de commande, il est toujours possible de décider quelle instruction doit être interprétée sans consulter le pseudo-code. En particulier, la

structure du programme y est implicitement contenue. L'édition d'un segment est facilitée car le pseudo-code pour les nouveaux incréments est ajouté à la fin du pseudo-code existant et il est repéré par le pointeur dans le dictionnaire. Quand l'édition modifie l'ordre des instructions, les changements sont reflétés dans les champs contenant le numéro du prochain incrément. L'ordre donné par le chaînage des numéros des incréments est appelé l'ordre lexicographique. Dans l'avenir, on pourrait envisager la présence d'écrans de télévision qui permettent de "visualiser" les incréments dans l'ordre lexicographique.

5.5. Problèmes de portée en interprétation

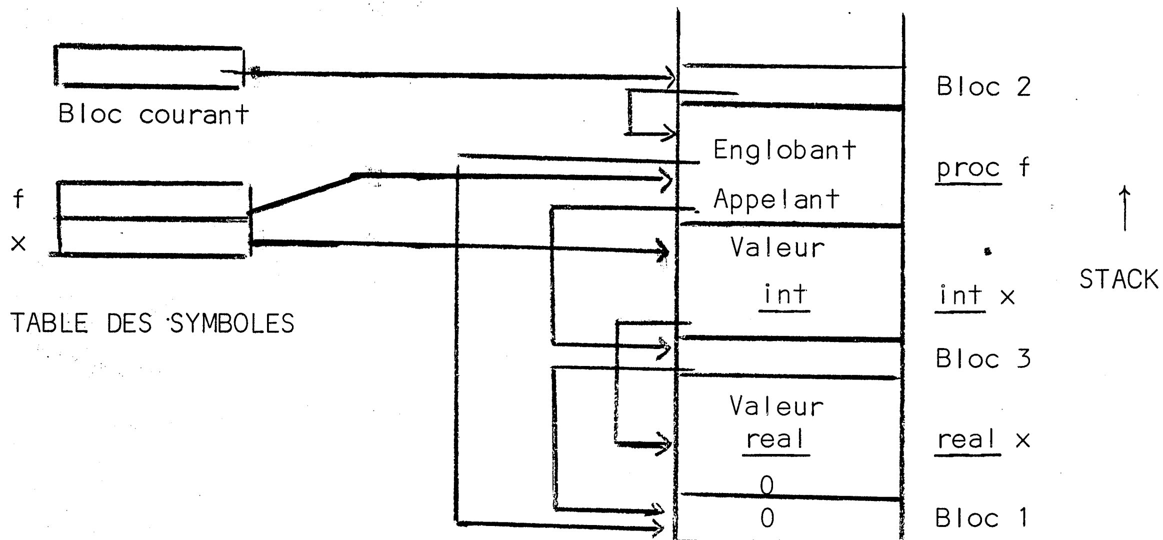
Comme les incréments d'un segment sont traités indépendemment les uns des autres, la relation entre l'utilisation d'un identificateur et sa déclaration n'est faite qu'à l'interprétation. Pour un identificateur local, cette relation est facile à établir, mais pour un identificateur non-local elle dépend de l'imbrication statique des blocs. De plus, pendant l'exécution les blocs sont imbriqués d'une façon dynamique, car une procédure peut être appelée n'importe où dans la portée de sa déclaration. Ceci peut provoquer des erreurs si l'interpréteur n'est pas bien construit,par exemple dans le programme ALGOL :

```
1       begin réel x ;
              procédure f ;
2             begin x := 1
              end ;
3             begin integer x ;
                        f
              end
        end
```

Au moment où la procédure f est appelée, le plus récent x est celui du bloc 3, mais la référence à x en f est au x du bloc 1. Pour résoudre ce problème, le chaînage dans le stack de l'interprétation est doublé, et au début de chaque procédure il y a un pointeur vers le bloc lexicographiquement englobant (pointeur statique) et un pointeur vers le bloc contenant l'appel de la procédure (pointeur dynamique). Au retour de la procédure le stack est mis à jour à l'aide du pointeur dynamique. Le chaînage des identificateurs est établi à partir de la table des symboles décrite ci-dessous.

5.6. La table des symboles

Pour retrouver les valeurs des variables dans un compilateur conversationnel, on utilise une "table des symboles" contenant une entrée pour chaque nom apparaissant dans le segment. Cette entrée contient un pointeur vers la plus récente entrée sur le stack correspondant à ce nom. Cette technique couvre en même temps la récursivité (la même déclaration existe alors plusieurs fois dans le stack) et l'utilisation du même nom déclaré dans des blocs différents. Pour retrouver les différentes entrées d'un nom sur le stack, chaque entrée pointe vers l'entrée précédente correspondant au même nom. Considérons le programme donné précédemment. Au moment de l'affectation à x dans la procédure f le stack et la table des symboles sont comme suit :



Le problème est de retrouver la valeur de x de l'intérieur du bloc 2. On note que le pointeur vers x dans la table des symboles a une valeur inférieure à la valeur du pointeur au bloc courant. On en déduit que x est non-local. Maintenant on suit la chaîne statique pour trouver le premier bloc dont la valeur du pointeur de début est inférieure à la valeur du pointeur vers x. Dans l'exemple on s'arrête au bloc 1. Puis on suit la chaîne des x pour trouver le dernier x dont le pointeur est toujours supérieur au pointeur du bloc 1, dans ce cas le deuxième x. On vérifie que cet x est déclaré dans le bloc 1 en suivant la chaîne dynamique jusqu'à la dernière valeur encore supérieure au pointeur du bloc 1. Comme cette valeur est supérieure au x considéré, celui-ci est donc déclaré dans le bloc 1 et est le x recherché.

Avec la table des symboles et le stack, il est possible de récupérer la valeur de toute variable déclarée dans le segment en cours. Les variables locales sont trouvées par un simple adressage indirect sur la table des symboles et les variables non-locales de la façon montrée ci-dessus. Il reste à décrire la méthode de recherche des valeurs déclarées dans un segment extérieur.

5.7. Communication entre segments

La communication entre deux segments est compliquée par l'absence d'un dictionnaire commun. L'identificateur x dans un segment n'a aucun lien avec l'identificateur x d'un autre segment. On a vu en section 2.4. que l'instruction spéciale nonlocal a été créée en vue d'associer les noms déclarés dans un segment à ceux déclarés dans un autre. Cette instruction est considérée comme une déclaration spéciale qui introduit un adressage indirect dans la référence.

Considérons le programme :

```
segment seg1 ;
begin
      :
end
nonlocal x ;
endsegment
segment seg2 ;
begin réel y ;
      :
      include seg1 using y ;
end
endsegment
```

Quand seg1 est entré, la déclaration de x est mise sur le stack. Cette déclaration est de type nonlocal et a comme valeur un pointeur vers la déclaration de y. Par ailleurs, les identificateurs de chaque segment sont numérotés de un à n. On pense intuitivement qu'il y aurait besoin d'une table des symboles pour chaque segment. En fait, il est possible d'utiliser la même table car il n'y a aucune référence dans un segment à une quantité extérieure, non définie comme nonlocal. Donc, il n'y a pas de confusion possible entre le fait que, par exemple, l'identificateur 1 puisse correspondre à deux noms différents car une référence à l'identificateur 1 est toujours locale.

5.8. Allocation pour plusieurs utilisateurs

Dans les quatre dernières sections nous avons supposé qu'il n'y avait qu'un utilisateur du système à un moment donné. Cette hypothèse est évidemment fausse et il faut considérer l'influence de l'existence de quelques dizaines de consoles pouvant être utilisées simultanément.

Comme idée de base on séparera ce qui est commun à tous les utilisateurs de ce qui est particulier à un d'entre eux. Le compilateur (c'est-à-dire générateur, interpréteur et éditeur) est commun et il n'en existe qu'une copie, conservée en mémoire; par contre, les données du compilateur (segments, pseudo-code, dictionnaire, stacks, etc...) sont particulières à leurs utilisateurs.

Pour chaque console on réserve une zone de mémoire de 4096 octets, dans laquelle l'information nécessaire pour la génération ou l'édition est conservée. Le pseudo-code est rangé dans la mémoire secondaire au fur et à mesure qu'il est produit. Le pseudo-code est défini de sorte que le pseudo-texte pour un incrément forme un tout. Au moment de l'interprétation d'un programme, la mémoire est demandée par pages.

La séparation entre programme et données implique l'obéissance à certaines règles. Le programme doit être ré-entrant, c'est-à-dire qu'il ne se modifie pas pendant son exécution. Pour garder la possibilité de ne pas mettre tout le programme en mémoire, il est souhaitable que le programme soit translatable. Toute référence aux données utilise un registre de base, qui indique l'adresse du début de la zone de mémoire réservée pour la console active. Ce registre est mis à jour à chaque changement d'utilisateur. L'adresse absolue d'une donnée n'est jamais conservée.

Dès que ces règles sont établies, on peut écrire le compilateur comme s'il n'y avait qu'un utilisateur, en laissant au système le soin de faire les transferts entre les utilisateurs. En pratique, il est simple de programmer de cette manière, à l'exception de quelques difficultés mineures dues à la définition du code machine du 360.

5.9. Les mots-clés de PL/I

En PL/I les mots-clés ne sont pas typographiquement distingués des identificateurs. Cette facilité du langage pose deux problèmes au réalisateur : un problème pratique qui est d'écrire un algorithme de discrimination et un problème théorique qui est d'être sûr que la définition du langage ne permet aucune ambiguité

syntaxique. En particulier, on veut prouver qu'il n'existe aucune chaîne du langage représentée par deux structures syntaxiques valables, en considérant successivement un nom comme mot-clé puis comme identificateur. Un projet réalisé en collaboration avec G. TASSART indique une solution à ces deux problèmes [14]. Le même problème en FORTRAN a été resolu par D. CLAUZEL et V. BAJAR [63] .

La grammaire pour le compilateur incrémentiel suppose que les mots-clés sont réservés. On a ajouté à cette grammaire la règle suivante :

```
Identifier —> BEGIN
              END
              IF
              :
              Terminalidentifier
```

Terminalidentifier consiste en tout nom qui n'a pas la forme d'un mot-clé, tous les mots-clés étant eux-mêmes des expansions possibles du nom de classe Identifier. La nouvelle grammaire n'est pas déterministe dans la mesure où les mots-clés posent des difficultés, et tout point ambigu sera indiqué par le transformateur de grammaire. En ces points seront insérées des fonctions sémantiques qui exécuteront les algorithmes de discrimination. Ceci est décrit dans le papier [14], mais cette solution est susceptible d'améliorations. Il est possible d'écrire une grammaire déterministe de PL/I qui fait la discrimination sans utiliser des fonctions sémantiques. Cette grammaire présente l'inconvénient d'avoir beaucoup plus de règles que l'originale, de ne pas représenter les idées intuitives du programmeur et d'être difficile à écrire. Néanmoins, elle est un moyen plus propre d'arriver à nos buts. Cette dernière étape est le projet de D.E.A. de G. TERRINE [54] .

5.10. Sur les types des variables

Dans les langages modernes de programmation il est parfois difficile de connaître le type d'une variable, l'information n'étant pas disponible au moment

convenable. En particulier, les déclarations en PL/I sont faites de trois façons (explicitement, contextuellement et par défaut) et la déclaration d'une variable peut aussi apparaître après une utilisation. Il est donc normal de faire le rapprochement des utilisations et les déclarations au cours d'un deuxième passage du texte, le premier ayant déterminé les portées des déclarations. Malheureusement, dans le "syntax checker", un deuxième passage n'est pas possible pour des raisons économiques ; des algorithmes spéciaux sont utilisés qui confirment l'existence de certaines des variables référencées et qui vérifient la compatibilité de type entre les utilisations et les déclarations. Il n'est pas nécessaire d'attacher les utilisations aux déclarations particulières car le syntax checker ne produit pas de code objet et n'a pas besoin de faire une allocation de mémoire. Les algorithmes sont décrits en [17] et nous ne donnons ici que les grandes lignes.

On garde l'information sur les identificateurs d'un programme PL/I sur un stack. L'entrée sur le stack correspondant à un nom est trouvée à l'aide d'une table des symboles. Une entrée contient les champs suivants :

- un pointeur vers la table des symboles qui est utilisé pour reconstituer à la fin d'un bloc la situation du début du bloc,
- un pointeur vers l'entrée précédente pour ce nom,
- le type d'une éventuelle déclaration explicite au niveau 1,
- le type d'une éventuelle déclaration contextuelle,
- l'utilisation ou la déclaration dans une structure,
- l'utilisation ou la déclaration comme variable indicée,
- l'utilisation ou la déclaration comme étiquette ou point d'entrée.

L'information est mise dans ces champs au fur et à mesure qu'elle devient disponible en analysant le programme. Quand un nom est rencontré, le pointeur dans la table des symboles indique si le nom a déjà une entrée dans le bloc courant. Sinon, une nouvelle entrée est créée. Les tests à l'intérieur d'un champ sont faits à l'arrivée de l'information. A la fin du bloc les entrées de ce bloc sont considérées une par

une. Pour chaque entrée, la compatibilité entre les différents champs est contrô
avant la destruction de l'entrée. Le pointeur dans la table des symboles est rem
à sa valeur à l'entrée du bloc. Quand une entrée peut avoir un effet dans un blo
extérieur, cette information est mise dans l'entrée pour le même nom dans le bloc
englobant.

Avec cette dépense raisonnable de place en mémoire, une bonne proportion d'erreurs est détectée sans nécessiter un appel au compilateur. Il semble que l'effort nécessaire pour trouver toutes les erreur, c'est-à-dire fournir les diag-nostics équivalents à ceux du compilateur, ne soit pas économiquement valable.

CHAPITRE VI

CONCLUSION

6. Conclusion

Cette thèse est présentée en accord avec la nouvelle règlementation qui accepte la combinaison d'un rapport de travail et des références à des papiers publiés antérieurement sur le même sujet. Les références à considérer sont 1, 2, 3, 7, 9, 10 11, 12, 14, 17, 25, 29, 31, 37 et 45.

Un lecteur éventuel de cette documentation aurait peut-être l'impression que no aimons la complexité et que les algorithmes proposés et réalisés correspondent à ce goût. Il aurait souvent raison en ce sens que les problèmes soulevés par l'implant tion des lan ages sophistiqués de programmation sont d'un intérêt intellectuel considérable. Malheureusement, cet intérêt intellectuel ne correspond pas aux intérêt économiques et pratiques de la plupart des utilisateurs des calculateurs électroniq On s'oriente actuellement vers des langages de plus en plus complexes utilisés par nombre croissant de programmeurs. Nous espérons voir à l'avenir des langages simpl et clairs pour le débutant, accompagnés d'une gamme de langages ou d'extensions spéc lisés pour les professionnels.

La clarté n'est pas seulement importante pour l'utilisateur d'un compilateur, m elle est aussi primordiale pour le réalisateur. Les difficultés rencontrées en PL/ par exemple, sont dans une grande mesure dûes à l'imprécision de ses spécifications Il est infiniement regrettable qu'un langage qui est si fonctionnel et qui devient répandu soit si difficile à mettre en oeuvre sur une machine. Il y a ici une quest de formalisme qui préoccupe bon nombre d'informaticiens, et avec juste raison. Le formalisme de cette science n'est actuellement pas équilibré entre la syntaxe et la sémantique des langages, ce qui apparaît dans ce document en comparant la longueur d la discussion syntaxique et le manque presque total d'outils sémantiques. On risqu d'attendre longtemps avant que l'équilibre soit rétabli.

Le formalisme syntaxique existant ne représent aussi qu'un commencement, mais il semble aller dans la bonne direction. Les bases théoriques de ce formalisme sont encore moins avancées, mais le travail réalisé par KNUTH [13], [55] , COURTIN [56] et d'autres chercheurs donne des raisons d'être optimiste.

BIBLIOGRAPHIE

Bibliographie

[1] M. GRIFFITHS, "Anatomy of a Compiler",
R.R.E. Memorandum 2296, Juin 1966

[2] D. BENSON et al., "A Language for Real-Time Computing Systems",
B.C.S. Quarterly, Décembre 1967

[3] I.F. CURRIE, M. GRIFFITHS, "CORAL 66 Manual",
R.R.E. Technical Note 732, Mai 1967

[4] J.G. GIBBS, "CORAL, an Initial Jovial Subset for Linesman Use",
R.R.E. Memorandum 2166, Septembre 1965

[5] J.W. BACKUS et al., "Report on the Algorithmic Language ALGOL 60",
CACM, Décembre 1960

[6] P.M. WOODWARD, "CORAL Re-Appraised",
R.R.E. Memorandum 2320, Août 1966

[7] I.F. CURRIE, M. GRIFFITHS, "A Self-Transferring Compiler",
R.R.E. Memorandum 2358, Février 1967

[8] J.M. FOSTER, "A Syntax Improving Device",
Computer Journal, Mai 1968

[9] M. GRIFFITHS, M. PELTIER, "A Macro-Generable Language for the 360 Computers",
à paraître

[10] M. GRIFFITHS, M. PECCOUD, M. PELTIER, "Incremental Compilation of ALGOL for a Multi-Access System",
I.M.A.G., Novembre 1967

[11] M. GRIFFITHS, "Incremental Compilation of PL/I for S/360",
SEAS, La Haye, Septembre 1968

[12] M. GRIFFITHS, M. PECCOUD, M. PELTIER, "Incremental Interactive Compilatior
IFIP, Edinburgh, Août 1968

[13] D.E. KNUTH, "Top-Down Syntax Analysis",
International Summer School, Copenhagen, Août 1967

[14] M. GRIFFITHS, G. TASSART, "Discrimination of Key-Words in PL/I",
I.M.A.G., Janvier 1969

[15] IBM System/360, "PL/I Language Specifications",
IBM Form Y33-6003

[16] H. CHLADEK, V. KUDIELKA, E. NEUBOLD, "Syntax of the New Programming Language PL/I",
IBM Vienna Report LR 0.006-1, Juin 1965

[17] M. GRIFFITHS, "One-Pass Type Checking in PL/I",
à paraître à SEAS, Grenoble, Septembre 1969

[18] E.W. DIJKSTRA, "On Cooperating Sequential Processes",
International Summer School, Villard de Lans, Septembre 1966

[19] IBM, "General Purpose Systems Simulator",
IBM Form 7090-CS-05X

[20] O.J. DAHL, K. NYGAARD, "SIMULA - An ALGOL Based Simulation Language",
CACM, Septembre 1966

[21] H.M. MARKOWITZ, B. HAUSNER, H.W. KARR, "SIMSCRIPT : A Simulation Programmi
Language",
RAND Report RM 3310, Novembre 1962

[22] C.J. SHAW, "A Specification of JOVIAL",
CACM, Décembre 1963

[23] A. van WINJGAARDEN, B.J. MAILLOUX, J.E.L. PECK, C.H.A. KOSTER,
"Final Draft Report on the Algorithmic Language ALGOL 68",
Amsterdam, MR 100, Décembre 1968

[24] N. WIRTH, "Communication à 10ème Anniversaire d'ALGOL", Zurich, Mai 1968

[25] M. GRIFFITHS, "Systèmes pour la Compilation Incrémentielle de PL/I, ALGOL et FORTRAN", Colloque sur les Systèmes Conversationnels, Grenoble, Novembre 1968 à paraître chez DUNOD

[26] K.E. IVERSON, "A Programming Language", Wiley, New York, 1962

[27] R.M. BURSTALL, R.J. POPPLESTONE, "The POP-2 Reference Manual", Machine Intelligence 2, Edinburgh, Oliver & Boyd, 1968

[28] G.Y. BREITBARD, G. WIEDERHOLD, "The ACME Compiler", IFIP, Edinburgh, Août 1968

[29] M. GRIFFITHS, M. PELTIER, "Grammar Transformation as an Aid to Compiler Production", I.M.A.G., Février 1968

[30] M. PELTIER, "The Macro System", I.M.A.G., Février 1969

[31] M. GRIFFITHS (ed.), "RREAC Programming Manual, Part 2", R.R.E. 1963

[32] D.E. KNUTH, "The Remaining Trouble Spots in ALGOL 60", CACM, Octobre 1968

[33] L. BOLLIET, "Compiler Writing Techniques", International Summer School, Villard de Lans, Septembre 1966

[34] L. BOLLIET, "Notation et Processus de Traduction des Langages Symboliques", Thèse d'Etat, Grenoble, Juin 1967

[35] A. COLMERAUER, "Précédence, Analyse Syntaxique et Langages de Programmation" Thèse d'Etat, Grenoble, Septembre 1967

[36] S.A. GREIBACH, "Formal Parsing Systems",
CACM, 1964

[37] M. GRIFFITHS, "How to Use the Grammar Transformer",
I.M.A.G., Octobre 1968

[38] M. ASSABGUI, "Interprétation d'une Chaîne Codée Générée par un Transformat de Grammaire",
à paraître, I.M.A.G.

[39] M. ASSABGUI, "Notation SRL et Génération Automatique comme Analyseur",
à paraître, I.M.A.G.

[40] J. BORDIER, "Thèse 3ème Cycle",
à paraître, I.M.A.G.

[41] F. MARTIN, "Détermination de Certains Caractéristiques des Grammaires et Langages 'Context-Free'",
Thèse 3ème Cycle, I.M.A.G., Mai 1969

[42] J. McCARTHY, "Recursive Functions of Symbolic Expressions and Their Evaluation by Machine",
CACM, Avril 1960

[43] P.M. WOODWARD, "List Processing in RREAC",
R.R.E. Memorandum 2385, Juillet 1967

[44] J.M. FOSTER, "List Processing",
Macdonald Computer Monograph, 1967

[45] M. GRIFFITHS, M. PELTIER, "Self-Contained ALGOL List Pack",
I.M.A.G., Février 1968

[46] J. COHEN, N. HUU DUNG, "Définition de Procédures LISP en ALGOL",
Chiffres, 1965

[47] E.W. DIJKSTRA, "ALGOL 60 Translation",
Supplément à ALGOL Bulletin 10, 1960

[48] B. RANDELL, L.J. RUSSELL, "ALGOL 60 Implementation", Academic Press, 1964

[49] K. JACKSON, R.J.W. KERSHAW, P.R. WETHERALL, "CORAL", à paraître à DATAFAIR 69, Manchester, Août 1969

[50] Dr. GOOS, "Implementing ALGOL 68", Tenth Anniversary Colloquium on ALGOL, Zurich, Mai 1968

[51] L. TRILLING, "Contribution à l'Etude des Mécanismes de Traduction des Langages de Programmation", Thèse de Docteur-Ingénieur, I.M.A.G., Décembre 1967

[52] D.T. ROSS, "The AED Free Storage Package", CACM, Août 1967

[53] Dartmouth College Computing Center, "BASIC", 1966

[54] G. TERRINE, "Rapport du Projet de D.E.A.", à paraître

[55] D.E. KNUTH, "On the Translation of Languages from Left to Right", Information and Control 8, 1965

[56] J. COURTIN, "Langages Analysables de Gauche à Droite : Construction d'un Analyseur pour Langages LR(1)", Thèse 3ème Cycle, I.M.A.G., Juin 1968

[57] J.P. VERJUS, "Etude et Réalisation d'un Système ALGOL Conversationnel", Thèse de Docteur-Ingénieur, I.M.A.G., Juillet 1968

[58] J.E. HOPCROFT, J.D. ULLMAN, "Formal Languages and Their Relation to Automata", Addison-Wesley, 1969

[59] G. TASSART, Thèse de Docteur-Ingénieur, I.M.A.G., à paraître

[60] N. WIRTH, "A Programming Language for the 360 Computers", Journal de l'ACM, Janvier 1968

[61] N. WIRTH, H. WEBER, "EULER : A Generalization of ALGOL and its Formal Definition",
CACM, Janvier 1966

[62] B. MAILLOUX, "On the Implementation of ALGOL 68",
Thèse, Amsterdam, Juin 1968

[63] V. BAJAR, D. CLAUZEL, "Compilateur FORTRAN Conversationnel",
à paraître

VU

Grenoble, le

Le Président de la Thèse

VU

Grenoble, le

Le Doyen de la Faculté des Sciences

Vu, et permis d'imprimer,

Le Recteur de l'Académie de GRENOBLE