UNIVERSITE DE NANCY 1

THESE

présentée à l'Université de Nancy 1

pour obtenir le grade de

Docteur de spécialité INFORMATIQUE

par

Jean-Claude MERSDORF

« COMPILATION CONVERSATIONNELLE SUR PETITE MACHINE »

Soutenue publiquement le 27 Septembre 1976 devant la Commission d'Examen :

Membres du Jury :

Messieurs	C. PAIR	Président
	A. FRUHLING	Examinateurs
	M. VERON	
	J.C. DERNIAME	
	J. SOUPIROT	

UNIVERSITE DE NANCY 1

THESE

présentée à l'Université de Nancy 1

pour obtenir le grade de

Docteur de spécialité INFORMATIQUE

par

Jean - Claude MERSDORF

« COMPILATION CONVERSATIONNELLE
SUR PETITE MACHINE »

Soutenue publiquement le 27 Septembre 1976 devant la Commission d'Examen:

Membres du Jury:

Messieurs	C. PAIR	Président
	A. FRUHLING	Examinateurs
	M. VERON	
	J.C. DERNIAME	
	J. SOUPIROT	

S O M M A I R E

Pages

Pages

AVANT - PROPOS

Je tiens à exprimer ma sincère reconnaissance à Monsieur le Professeur C. PAIR, Président de l'Institut National Polytechnique de Lorraine, pour la bienveillance qu'il m'a témoignée et pour l'honneur qu'il me fait en présidant la Commission d'Examen.

Que Monsieur le Professeur A. FRUHLING, Directeur du Laboratoire d'Electricité et d'Automatique, veuille trouver l'expression de ma profonde gratitude pour l'accueil qu'il m'a réservé dans son Laboratoire.

Ce travail a été effectué dans le cadre des activités du Centre Universitaire de Commande Numérique qu'anime avec dynamisme M. VERON, Maître de Conférences, que je tiens à remercier ici pour ses encouragements constants.

Je remercie vivement Monsieur J.-C. DERNIAME, Maître de Conférences, pour la formation qu'il m'a donnée et pour l'honneur qu'il me fait en participant à mon Jury.

Monsieur J.SOUPIROT Ingénieur à la Société CROUZET, est à l'origine du contrat dont ce travail fait l'objet. Qu'il trouve ici l'expression de ma reconnaissance pour l'intérêt qu'il y a constamment porté et pour sa participation à mon Jury.

J'adresse mes remerciements à O. DOUCHIN pour sa collaboration efficace et à tous mes camarades du groupe Commande Numérique pour l'ambiance amicale qu'ils y font régner.

Enfin, je remercie Madame J. SCHWARTZ pour la qualité et la rapidité de son travail dans la réalisation matérielle de ce mémoire.

INTRODUCTION

L'implantation du langage FORTRAN IV sur un mini-calculateur n'est pas une chose nouvelle en soi.

Devant le succès croissant de ce type de matériel et pour répondre aux besoins en programmes de leurs utilisateurs, les constructeurs ont en effet rapidement doté leurs mini-systèmes de langages tels que FORTRAN ou BASIC.

Disposant généralement d'un surcroît de mémoire centrale et équipés de mémoires de masse tels que les disques rapides, ces mini-calculateurs offrent souvent les facilités que l'on trouve sur les gros systèmes.

Bien que des études théoriques aient été menées [6], peu de tentatives ont été faites pour implanter sur une configuration réduite et dans le contexte habituel du mini-ordinateur -faible capacité mémoire, usage du ruban perforé- un compilateur FORTRAN qui soit d'une mise en oeuvre aisée.

Notre réalisation qui se situe dans le cadre d'un contrat avec la Société CROUZET a été menée dans ce sens.

Dans la configuration minimale retenue et comportant un mini-calculateur CAMPANULE-CROUZET de 12 K mots, un télé-imprimeur et un disque souple, le système était plus particulièrement destiné à l'enseignement et au traitement de petits problèmes scientifiques.

L'objet de ce travail est alors de montrer, après un rappel de quelques notions de compilation (chapitre 1) et la présentation du système (chapitre 2), comment l'implanta-

tion d'un tel logiciel a été rendue possible en décrivant l'architecture adoptée pour le compilateur (chapitre 3) et en justifiant les choix effectués dans la réalisation de l'analyseur (chapitre 4).

Les problèmes rencontrés à la définition du programme-objet sont précisés (chapitre 5) et quelques principes de réalisation introduits (chapitre 6).

Des aspects propres à la mini-informatique sont exposés enfin (chapitre 7).

CHAPITRE I

COURTES GENERALITES SUR LA COMPILATION

1.1. Position du problème de la compilation :

L'emploi des premiers ordinateurs exigeait de la part de l'utilisateur l'apprentissage du "code interne" de la machine, appelé encore langage machine ; la programmation était alors souvent affaire de spécialistes. L'utilisation d'un tel langage impliquant un effort important d'écriture et de mise au point, la nécessité d'introduire des langages plus synthétiques et qui apportent un certain allègement pour le programmeur s'est fait ressentir assez rapidement.

Ceci a entrainé le développement des langages d'assemblage d'abord, encore assez proches du code de la machine, des langages évolués, ensuite, assez près de la formulation mathématique et par certains aspects du langage naturel.

Un programme écrit dans un langage évolué n'étant pas directement assimilable par la machine, sa mise en oeuvre exige au préalable une phase de traduction sous une forme plus proche du code interne.

Le programme qui réalise cette traduction est appelé compilateur et l'étape de traduction la compilation.

La donnée du compilateur est désignée sous le terme de programme-source et le résultat sous celui de programme-objet.

Les techniques de compilation s'appuyent largement sur la théorie des langages [3] dont nous allons rappeler succinctement le contenu en se limitant aux notions qui serviront dans le cadre de cette étude.

1.2. Rappel sur les langages :

1.2.1. Langages :

Un langage est formé de "mots" constitués par des "symboles" appartenant à un "alphabet".

Donnons quelques définitions.

Alphabet : c'est un ensemble non vide et fini d'éléments.

Mot : c'est une suite finie $(a_1, a_2, \ldots, a_n)$ d'éléments appartenant à l'alphabet.
On note plus simplement un mot $a_1\ a_2 \ldots a_n$.

Mot vide : mot qui ne contient pas de symboles.

Longueur d'un mot : un mot $a_1\ a_2 \ldots a_n$ a pour longueur n.

Monoïde libre sur A : l'ensemble des mots sur un alphabet A, noté A^*, est appelé monoïde libre sur A.

Plus généralement, on appelle monoïde un ensemble muni d'une loi de composition interne associative et admettant un élément neutre.

Pour A^* la "concaténation" est une telle loi qui s'introduit naturellement.

Soit α et β deux mots sur A notés :

$$\alpha = a_1\ a_2\ \ldots\ a_n$$
$$\beta = b_1\ b_2\ \ldots\ b_m$$

La concaténation de α et β conduit au mot $\alpha\beta$ noté :

$$\alpha\beta = a_1\ a_2\ \ldots\ a_n\ b_1\ b_2\ \ldots\ b_m$$

L'élément neutre de la concaténation est le mot vide de longueur 0, que l'on note habituellement Λ.

Langage sur A : on appelle langage sur l'alphabet A toute partie de A^*.

Le premier problème qui se pose à propos d'un langage sur un alphabet A est celui de la reconnaissance :

étant donné un mot α sur A, α appartient-il au langage ?

L'introduction de la notion de grammaire apporte une solution à ce problème.

1.2.2. Grammaires :

Pour reconnaître si un mot appartient à un langage, nous sommes conduits à définir des "règles" de construction. Celles-ci permettront de disposer d'un algorithme de génération qui constituera en même temps, comme nous le verrons, un algorithme de reconnaissance.

Pour définir les règles de constructions, les mots sur A sont classés en diverses catégories syntaxiques qui elles-mêmes sont regroupées dans des catégories de niveau supérieur et ainsi de suite.

Ces catégories syntaxiques sont désignées par des symboles "auxiliaires" ou "non terminaux" par opposition aux éléments de l'alphabet qualifiés de "terminaux".

On appellera l'alphabet A "alphabet terminal" et l'ensemble des symboles non terminaux "alphabet non terminal".

Nous venons de voir qu'il existait une certaine hiérarchie entre symboles non terminaux. Le symbole non terminal associé à la catégorie syntaxique de niveau le plus élevé est privilégié et désigné sous le terme "axiome".

Ces notions étant introduites, il reste à définir les règles de construction (ou règles de production). Elles seront du type $B \rightarrow \alpha$, où B est un symbole auxiliaire (premier membre de la règle),

et α un mot sur la réunion des alphabets terminal et auxiliaire (le deuxième membre de la règle).

La règle $B \rightarrow \alpha$ peut se lire :

"B se réécrit α".

On peut définir maintenant une grammaire comme étant la donnée d'un quadruplet :

$$G = (T , N , : : = , X)$$

où :

T et N ensembles finis disjoints sont respectivement l'alphabet terminal et non terminal ;

X est l'élément de N appelé l'axiome ;

::= la relation binaire appelée règle de production entre N et V^* (V est la réunion de T et N et V^* l'ensemble des mots sur V).

La notion de dérivation permet de mettre en évidence le lien qui existe entre la grammaire et le langage.

Une dérivation pour une grammaire donnée est une suite (δ_o, δ_1, ..., δ_n) de mots sur la réunion de l'alphabet terminal et non terminal, telle que pour i variant de 1 à n, on passe de δ_{i-1} à δ_i en remplaçant un symbole non terminal B par un mot α, B :: = α étant une règle de production.

Il est possible alors de définir le langage engendré par une grammaire comme étant l'ensemble des mots sur l'alphabet terminal qui dérivent de l'axiome.

Nous disposons ainsi du critère de reconnaissance suivant :

pour une grammaire G donnée, un mot α appartient au langage engendré si et seulement si il existe une dérivation qui conduise de l'axiome au mot α.

1.2.3. Arbre syntaxique - Représentation post-fixée :

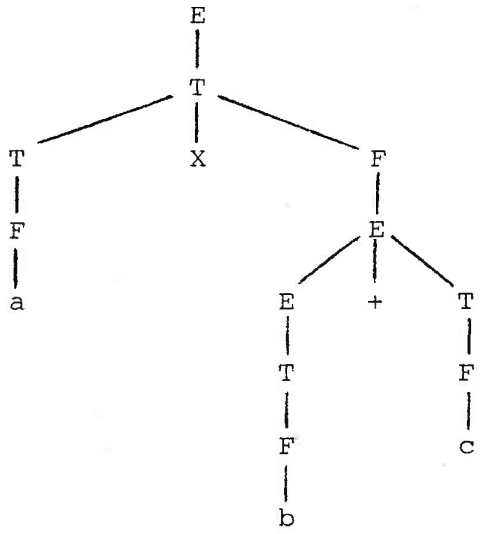
Arbre syntaxique :

En pratique, à toute dérivation peut être associé un schéma arborescent. Pour illustrer ce résultat, nous donnons ici une grammaire simplifiée des expressions arithmétiques en notation BNF [10].

```
<E>   :: =  <E>  +  <T> |1|     <T> |2|
<T>   :: =  <T>  x  <F> |3|     <F> |4|
<F>   :: =  ( <E> ) |5| a |6|  b |7|  c |8|
```

où <E> désigne l'axiome
[T,F] l'alphabet non terminal
et [a,b,c,(,),+,x] l'alphabet terminal.

La dérivation qui conduit alors de l'axiome au mot terminal a x (b+c) peut être représentée par le schéma arborescent suivant :



En règle générale, une structure arborescente peut toujours être associée à une dérivation conduisant à un mot terminal.

Les éléments composant la structure arborescente (ou arbre syntaxique) appartiennent à NuT, les feuilles de l'arbre sont les éléments de T, les noeuds les éléments de N et les branches symbolisent les règles de production employées.

C'est l'axiome qui constitue la racine de l'arbre.

Représentation post-fixée d'un arbre syntaxique :

Une représentation plus commode de l'arbre syntaxique peut être obtenue en notant les feuilles et les noeuds rencontrés lors d'un parcours systématique du schéma arborescent opéré en explorant la structure de gauche à droite et de bas en haut et en ne prenant en considération un noeud que lorsque toutes les branches qui en partent ont été examinées.

Il s'avère utile en pratique de remplacer chaque symbole non terminal par un numéro associé à la règle correspondante (notée || dans la grammaire).

Pour l'exemple précédent, cette définition conduit au résultat suivant :

a 64 x (b 742 + c 841) 532.

Lors de la traduction d'un programme, il importe de savoir si les mots qui le composent sont bien formés, ce qui, nous l'avons vu, revient à vérifier qu'ils dérivent de l'axiome suivant les règles en vigueur dans la grammaire associée au langage utilisé.

Cette procédure correspond à la phase d'analyse syntaxique. Les diverses méthodes d'analyse font l'objet de l'exposé qui suit.

1.3. L'analyse syntaxique :

Il existe deux grandes familles de méthodes d'analyse syntaxique :

- les méthodes d'analyse syntaxique ascendantes (Bottom - Up)

- les méthodes d'analyse syntaxique descendantes (Top - Down)

1.3.1. Analyse ascendante :

1.3.1.1. Principe

L'analyse ascendante repose sur le principe suivant : partant du mot terminal proposé, on cherche à le "réduire" à l'axiome de la grammaire.

A tout mot terminal appartenant au langage peut être associé un arbre syntaxique.

Si nous transposons le principe de l'analyse ascendante à la construction de l'arbre, la méthode revient à construire la structure arborescente en partant des feuilles et en allant vers la racine.

Signalons qu'à chaque étape de la construction, nous avons le choix entre ajouter une feuille ou un noeud.

1.3.1.2. Algorithme général

C'est un algorithme indéterministe car il y a choix à chaque étape entre lire un caractère dans la donnée (ajouter une feuille à l'arbre) et opérer une réduction (ajouter un noeud à l'arbre).

La procédure est essentiellement itérative et on peut la décrire très schématiquement de la manière suivante :

```
iter :  choix (lect : début lire (c) ; aller à iter fin ,
              red  : début réduire   ; aller à iter fin ,
```

L'algorithme peut cependant être affiné de façon à devenir déterministe. Diverses techniques sont utilisées. Signalons :

- la lecture de k caractères à l'avance dans le mot proposé ; si le déterminisme peut être réalisé ainsi, la grammaire est dite LR(k) [11] ;
- la consultation d'une matrice de précédence ; cette matrice renseigne sur l'existence de certaines relations entre deux symboles du vocabulaire total de sorte que dans chaque situation de choix, un simple examen peut permettre de ne retenir qu'une au plus des alternatives possibles.

1.3.2. Analyse descendante :

1.3.2.1. Principe

L'analyse descendante [12] procède de façon inverse. Partant de l'axiome, elle cherche à engendrer le mot terminal proposé, ce qui revient à construire l'arbre syntaxique en partant de la racine.

En pratique, il n'y a pas de génération à proprement parler, mais plutôt vérification de la présence dans le mot terminal de l'élément attendu. L'analyse est dite "prédictive".

Il est concevable dès à présent qu'il ne sera pas toujours facile de faire du premier coup les bonnes prédictions, du fait de l'existence des alternatives dans la grammaire.

1.3.3.2. Algorithme général

Appliquer un algorithme d'analyse descendante revient essentiellement à écrire une procédure d'analyse qui :

- pour un élément B de l'alphabet auxiliaire fait le choix d'une règle de production parmi toutes celles qui ont pour premier membre B ;

- pour un élément B de l'alphabet terminal vérifie que le caractère à lire dans le mot terminal est bien B.

Illustrons ce qui précède sur un exemple :

soit la grammaire d'axiome X :

```
<X>  ::=  <A> | <B>
<A>  ::=  <D> <A> | <D>
<B>  ::=  'b'
<D>  ::=  'd'
```

où b et d désignent des terminaux.

Les procédures d'analyse associées aux non-terminaux <X> , <A> , <B> et <D> s'écrivent :

```
procédure  analyseX   début   Choix (analyseA , analyseB)
                      fin

procédure  analyseA   début
           Choix (début analyseC ; analyseA fin , analyseC)
                      fin

procédure  analyseB   début   analyseb   fin

procédure  analyseD   début   analysed   fin
```

Les procédures associées aux éléments terminaux b et d s'écrivent :

```
procédure  analyseb   début
           lire(c);   si  c ≠ 'b'  alors  erreur
                      fin

procédure  analysed   début
           lire(c);   si  c ≠ 'd'  alors  erreur
                      fin
```

Le programme principal associé donne :

```
début   analyseX ; lire(c) ;  si c ≠ ⊣  alors erreur  fin
```

Remarques :

- Les procédures "lire" et "erreur" permettent respectivement de lire un caractère dans le mot terminal proposé et de traiter un cas où les choix effectués ont conduit à une impasse.

- La présence du caractère '⊣ ' délimiteur du mot proposé est indispensable pour s'assurer que la donnée a été lue intégralement.

L'indéterminisme de l'analyse se manifeste sur cet exemple dans les procédures associées à X et A.

1.4. L'analyse sémantique :

La grammaire adoptée ne traduit pas toutes les règles du langage. Par exemple, le fait qu'une étiquette ai déjà fait l'objet d'une définition ne peut s'exprimer à l'aide d'une telle grammaire.

On est donc amené à introduire des séquences de programme pour compléter la vérification.

L'ensemble des règles qui permettent d'attribuer une signification aux mots du langage définit sa "sémantique", la phase de la compilation qui s'attache à vérifier le respect de celles-ci constituant l'analyse sémantique.

1.5. L'analyse lexicographique :

En cours de traduction, certains objets, externes en particulier, les identificateurs devront être remplacés par leur représentation en machine. Pour cela, il faudra disposer d'un certain nombre de renseignements concernant leur genre, type et implantation.

Une table sera donc construite au fur et à mesure de l'analyse du programme-source.

La structure et la méthode d'accès de cette table constituent le problème essentiel de l'étude lexicographique.

1.6. Le langage FORTRAN IV :

En 1956, I.B.M. élabore puis développe le langage FORTRAN (FORmula TRANslator) destiné à l'ordinateur 704 de sa gamme. Repris alors par tous les constructeurs, ce langage a connu un certain nombre d'améliorations et, au fur et à mesure de son évolution, a pris les noms de FORTRAN II puis de FORTRAN IV. C'est cette dernière version définie par les normes AFNOR qui est étudiée ici.

FORTRAN étant certainement le mieux connu des langages de programmation [1], [2] nous ne décrirons donc pas toutes ses spécifications, mais les notions indispensables à la compréhension de l'exposé sont introduites au fur et à mesure des besoins.

Précisons d'ores et déjà qu'un programme FORTRAN est constitué d'un programme principal accompagné ou non d'un ou plusieurs sous-programmes. Dans la suite, le terme "unité" désigne indifféremment le programme principal ou un sous-programme. Une unité consiste en une suite d'instructions de déclarations ou formules écrites dans l'ordre logique où elles devront être exécutées.

Des instructions de séquencement autorisent l'exécution conditionnelle ou itérative de séquences de programme repérées par des étiquettes.

CHAPITRE II
DESCRIPTION DU SYSTEME

Dans ce chapitre, on se propose de décrire la configuration qui a servi de cadre à cette réalisation tout en précisant, tant du point de vue du matériel que du logiciel, les particularités d'un tel système.

2.1. Configuration :

L'ordinateur utilisé pour cette étude est le modèle CAMPANULE (Chaine Adaptable par Micro-Programmation pour l'Analyse Numérique et LogiquE) de la Société Crouzet, dans la configuration suivante :

- mémoire centrale de 12 K mots de 24 bits,
- double unité de disques souples d'une capacité de 90 K mots chacun; temps d'accès à un secteur de 150 ms à 1 s,
- lecteur de ruban 120 car/seconde,
- perforateur de ruban 47 car/seconde,
- télé-imprimeur 10 car/seconde.

2.2. Le logiciel :

2.2.1. Généralités :

Dans la configuration initiale du système existaient pour l'essentiel :

- un assembleur,
- un générateur de texte apparenté à un méta-assembleur, mais non intégré à l'assembleur.

La configuration finale avec les disques souples disposait en plus :

- d'un système d'enchainement de tâches en mono-programmation, DISCAMP,

- d'un moniteur d'entrées-sorties,
- de quelques procédures utilitaires pour la gestion des disques.

2.2.2. *Le système DISCAMP* :

Le superviseur DISCAMP est un système d'enchainement de travaux en mono-programmation utilisant le disque souple pour le stockage de la bibliothèque de programmes et pour les fichiers utilisateurs.

L'enchainement des travaux est réalisé par l'intermédiaire d'un langage de commande décrivant les étapes à effectuer dans un travail.

Le système repose sur l'existence d'une bibliothèque de processeurs sur disque, le noyau résidant en permanence en mémoire centrale, n'assurant que les fonctions essentielles.

Remarques :

- L'utilisateur assume aussi les fonctions d'"opérateur". Il active toute étape d'un travail et peut interrompre à tout moment l'exécution d'un processeur.

- La configuration minimale définie pour l'implantation du compilateur FORTRAN est :

  . une mémoire centrale de 12 K mots,
  . une unité de disques souples,
  . un téléimprimeur.

2.3. Le calculateur :

L'organisation du calculateur CAMPANULE s'écarte sensiblement de celle généralement adoptée pour la réalisation d'un ordinateur. Il est intéressant de donner ici quelques informations significatives à ce propos.

2.3.1. Concept général :

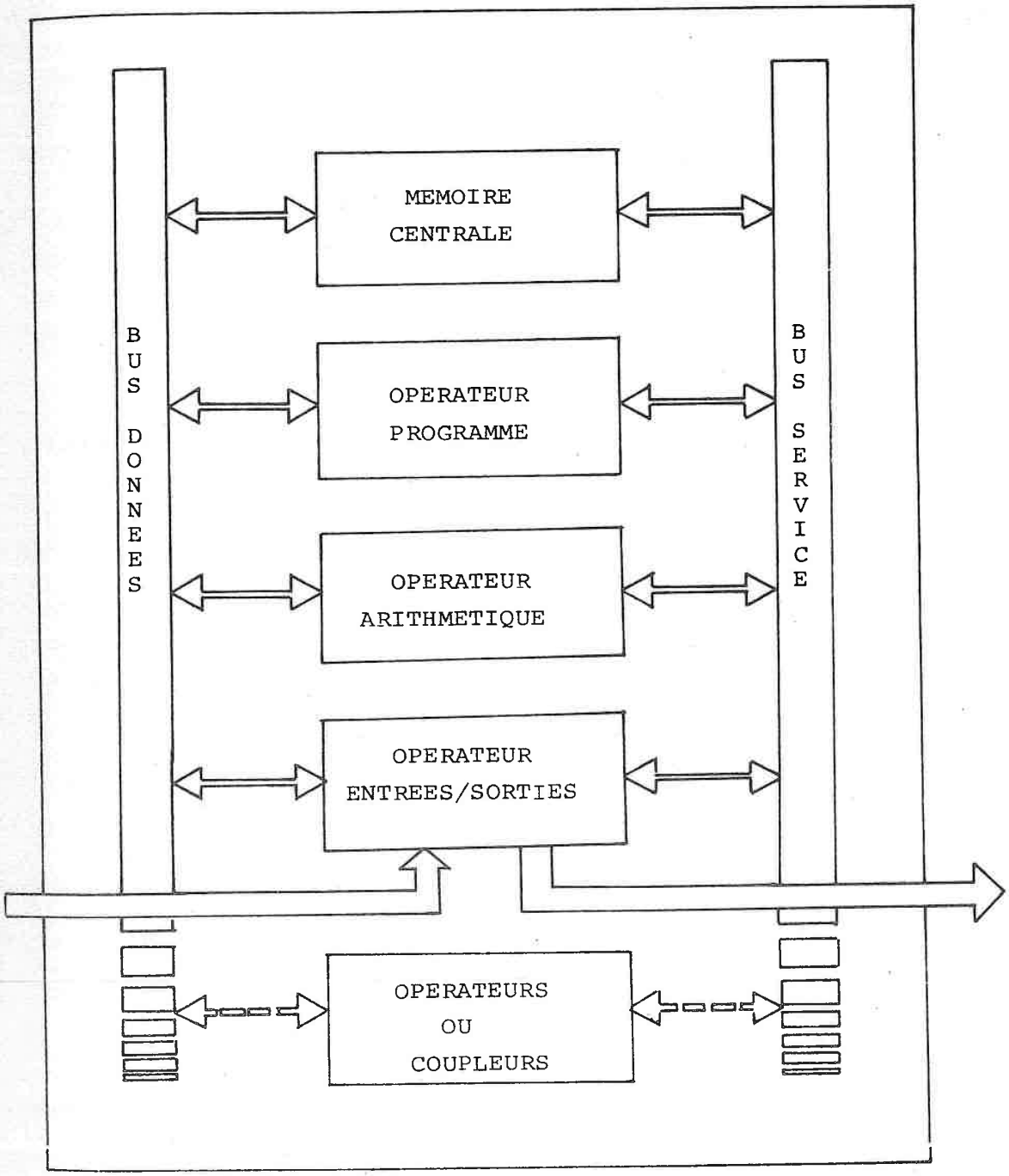
Le calculateur CAMPANULE a été conçu en s'appuyant sur deux techniques relativement récentes :

- l'architecture modulaire à bus,
- la micro-programmation.

2.3.2. Architecture modulaire :

Rappelons quelques fonctions essentielles d'un calculateur :

- stockage du programme et des données,
- interprétation des instructions du programme machine,
- traitement logique et arithmétique,
- communication avec l'environnement.



Architecture générale

Figure 2.1.

Les fonctions sont réalisées sur CAMPANULE par des modules qui sont à la fois des entités physiques et fonctionnelles. Ces modules désignés encore sous le terme "opérateurs" accèdent aux "bus" suivant le schéma décrit par la figure 2.1.

L'un des opérateurs est amené à jouer un rôle prépondérant : c'est l'opérateur programme qui gère l'ensemble des autres fonctions de la machine par l'intermédiaire du bus "service".

L'échange des informations entre opérateurs se fait à travers le "bus données".

2.3.3. Micro-programmation :

Les techniques de micro-programmation sur le CAMPANULE sont appliquées aux fonctions de décodage et aux logiques séquentielles.

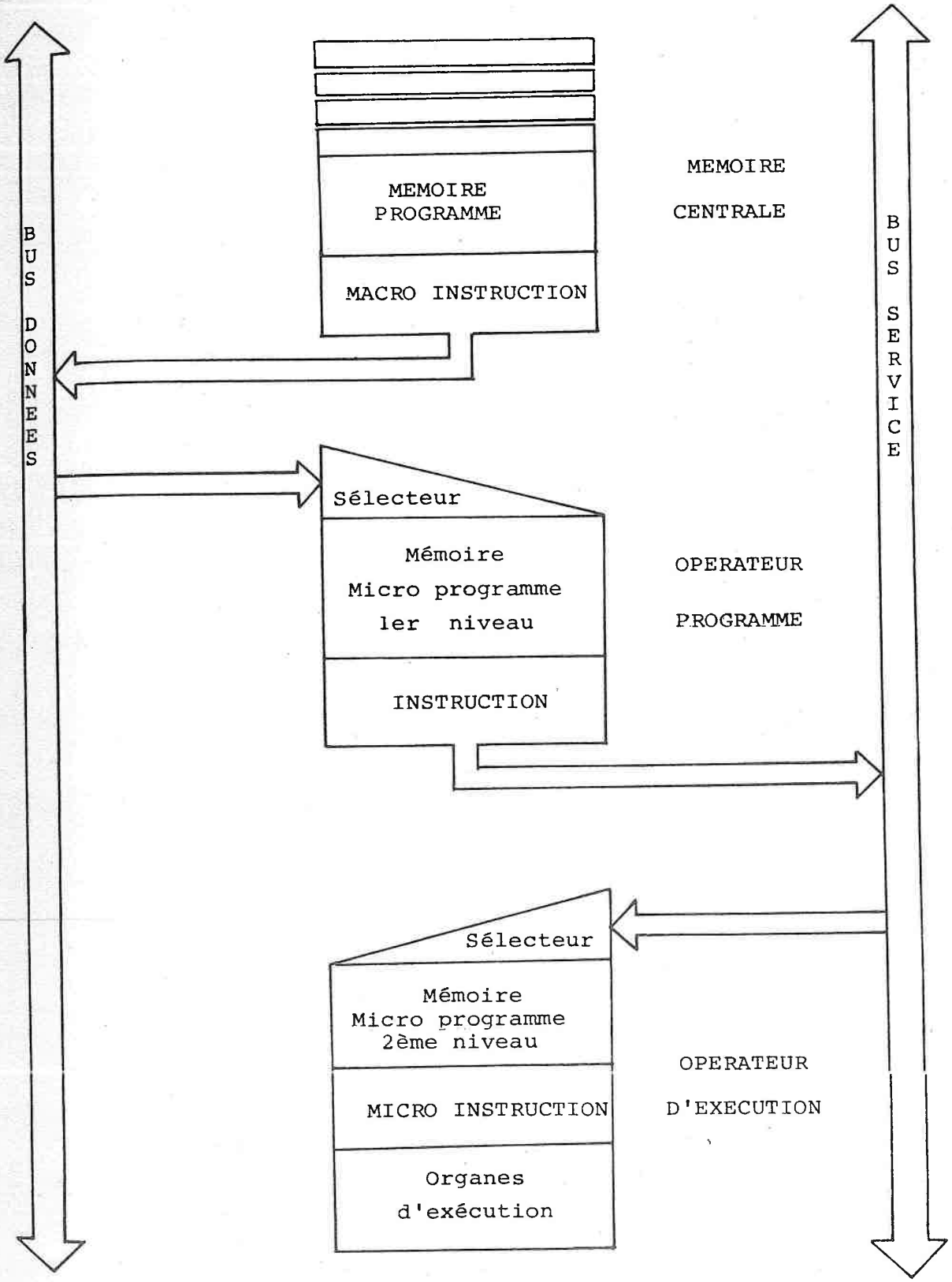
Le calculateur possède deux niveaux de micro-programmation :

- le premier est situé dans l'"opérateur programme",
- le second est propre à chaque opérateur.

Cette caractéristique est illustrée sur la figure 3.2.

Toute instruction du programme (appelée encore macro-instruction) est interprétée par l'opérateur programme en une suite d'instructions traitées par les différents opérateurs.

A la réception d'une instruction, l'opérateur désigné exécute des opérations simples ou complexes grâce à une mémoire de micro-programme générant séquentiellement les ordres nécessaires à ses organes de traitement (comptage, transfert, test, décalage, etc...).



Micro - programmation

Figure 2.2

2.3.4. Mémoire - Registres - Instructions :

Nous donnons ici quelques informations sur la mémoire et les instructions de la machine, pour mettre en évidence les contraintes imposées par le système utilisé.

2.3.4.1. La mémoire

La mémoire du CAMPANULE est divisée en pages physiques de 4096 mots de 24 bits. La représentation des objets en mémoire est la suivante :

- un programme est une suite d'instructions représentées chacune sur un mot de 24 bits ;

- les données ont deux représentations possibles :
  - . une représentation fixe sur 24 bits (qui occupe 1 mot),
  - . une représentation flottante sur 32 bits (qui occupe 2 mots) dont :
    - une mantisse de 24 bits (1 bit de signe)
    - un exposant de 8 bits (1 bit de signe)

    précisant la place de la virgule.

2.3.4.2. Les registres

Ils sont au nombre de 16 et peuvent contenir un nombre flottant ou une information d'un autre type. Dans le premier cas, ils disposent de 32 bits utiles et dans le second de 24 bits. Tous les registres peuvent servir à l'indexation.

2.3.4.3. Les instructions

La structure d'une instruction machine est présentée par la figure 2.3.

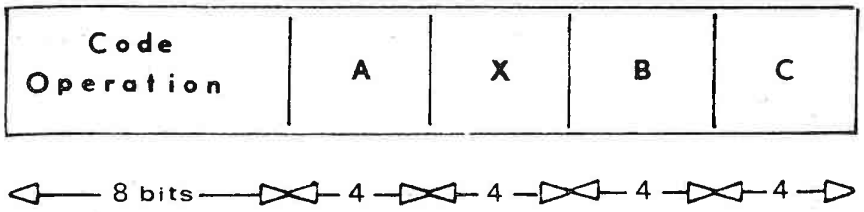


Figure 2.3.

où A désigne le registre opérande.

L'utilisation des zones X, B et C dépend du mode d'adressage :

- mode 3 registres :

  On effectue une opération entre 2 registres spécifiés en zone B et C ; le résultat est rangé dans le registre précisé dans la zone A (X est inutilisé) ;

- mode immédiat :

  La zone X - B - C est prise comme une seule entité et représente une valeur immédiate ;

- mode direct ou indirect :

  La zone X - B - C donne alors l'adresse en mémoire du deuxième opérande en l'adresse d'un pointeur ($\leqslant$ 4095) ;

- mode indexé :

  X spécifie un registre ; B - C donne une adresse mémoire ($\leqslant$ 255) sur laquelle porte l'indexation.

Le répertoire des instructions contient pour l'essentiel des instructions :

- arithmétiques en virgule fixe et flottante,
- logiques au niveau du mot ou du bit,
- de décalage-codage et comparaison,
- de test (signe d'un nombre - valeur d'un bit - ...),
- de transcodage (binaire <--> BCD),
- de calcul de fonctions complexes telles que sinus, cosinus, racine carrée, logarithme neperien et exponentielle.

Signalons que l'on ne dispose pas d'instructions d'accès à l'octet.

Par ailleurs, le calculateur est dépourvu de logique de déroutement ; ainsi les anomalies internes d'exécution telle que la référence à une adresse inexistante sur l'installation ou la division entière par zéro provoque simplement l'arrêt du calculateur.

2.3.5. La pagination :

Le champ d'adresse de 12 bits d'une instruction CAMPANULE ne permet d'adresser a priori qu'une zone de 4096 mots, soit une page de 4 K mots.

La mémoire étant extensible pour des tailles allant de 8 K mots à 64 K mots, un mécanisme d'adressage particulier a été introduit dans la logique du calculateur.

Ainsi, pour toute instruction à référence mémoire, la résolution d'adresse est obtenue de la manière suivante :

- un registre de 4 bits contenant le numéro de la page courante désigne une zone de 4 K mots dans la mémoire ;

- le champ d'adresse de l'instruction fournit le déplacement dans cette zone.

Il existe deux registres de ce type :

- le premier désigne la page "instructions" courante,
- le deuxième désigne la page "opérandes" courante.

Un changement de page opérande est obtenu sur exécution d'une instruction explicite de changement de page.

Un changement de page instruction a lieu :

- en fin de la page instruction courante,
- éventuellement lors d'une rupture de séquence indirecte ou indexée.

A l'écriture d'un programme machine, la gestion de ces deux registres demande de connaître l'implantation en mémoire des données et du programme.

CHAPITRE III

ARCHITECTURE DU COMPILATEUR

Les contraintes apportées par ce système dans l'organisation générale du compilateur sont dégagées dans ce chapitre.

3.1. Les contraintes :

Pour l'essentiel, elles sont au nombre de deux.

La première est celle de la taille de la mémoire centrale, l'espace alloué au compilateur étant de 8 K mots. Compte-tenu de l'existence d'une mémoire de masse, la segmentation du compilateur est une solution qui s'introduit naturellement.

La deuxième contrainte importante a trait au support extérieur du programme FORTRAN. En matière de coût à l'équipement et à l'utilisation, la supériorité du ruban sur la carte n'est plus à démontrer, mais son emploi reste cependant incommode. En effet, la moindre modification sur un fichier ruban exige sa réécriture complète et la localisation d'une information n'y est pas immédiate. Pour rendre viable un système qui utilise un tel support, on a été conduit à proposer, par programme, une aide à la création et à la modification d'un source FORTRAN. D'autres facilités sont offertes par introduction d'un mode de compilation conversationnelle.

3.2. Le système conversationnel :

Les techniques conversationnelles ont un champ d'application très vaste [9]. Dans notre réalisation, elles s'appliquent à la compilation et on se limitera ici à en exposer le principe. Le lecteur trouvera en annexe (A.17) le détail des commandes et de la mise en oeuvre.

3.2.1. Le_principe :

A la compilation un dialogue système-utilisateur du type "question-réponse" a été instauré, nécessitant un périphérique qui permette cette intéraction : un téléimprimeur ou une console.

Le système repose sur l'existence de directives de compilations à la disposition de l'utilisateur et qui permettent de travailler dans un des trois modes suivants :

- création d'un programme-source FORTRAN,
- modification d'un programme existant,
- recompilation d'un source existant.

Quel que soit le mode de travail choisi, la compilation est conversationnelle, c'est-à-dire qu'elle permet la correction directe d'erreurs syntaxiques et sémantiques.

3.2.2. Le_mode_création :

Dans ce mode, l'utilisateur soumet à l'analyseur des instructions FORTRAN à partir de son clavier.

Les instructions validées par le compilateur sont ajoutées au fichier source FORTRAN en sortie. Les instructions reconnues comme erronées font l'objet d'un message d'erreur et sont ignorées.

Avant toute saisie d'une ligne FORTRAN, le système émet à l'attention de l'utilisateur le numéro de séquence de l'enregistrement courant dans le fichier source en sortie.

Comme il y a identité entre ligne FORTRAN et enregistrement dans le fichier source, le numéro de séquence désigne aussi le numéro de ligne dans le programme-source.

3.2.3. Le mode modification :

Ce mode a pour objet de permettre la modification d'un programme existant, ce qui est rendu possible par l'analyse sélective d'une partie de celui-ci et l'adjonction d'un certain nombre d'autres instructions.

Dans le fichier source en entrée, les instructions sont repérées grâce au numéro de séquence évoqué ci-dessus.

Les définitions successives de plages du programme par la donnée d'un couple, numéro de la première et de la dernière instruction à prendre en compte, permettent alors l'analyse sélective du source existant.

Ce mode est accompagné implicitement de la recopie sur le fichier en sortie des instructions analysées et validées par le compilateur.

3.2.4. Le mode recompilation :

Son rôle est de permettre de recompiler dans son intégralité un source FORTRAN existant.

3.2.5. Remarques :

La syntaxe de FORTRAN a été légèrement modifiée pour faciliter la frappe au clavier. En effet, l'usage de la carte perforée autorise un découpage rigide d'une ligne FORTRAN en zone étiquette, suite, instruction et commentaire.

Cette décomposition mal adaptée à la saisie à partir du clavier d'un téléimprimeur est abandonnée, ici au profit d'une nouvelle syntaxe plus souple exposée en annexe (A.19).

3.2.6. Incidence sur le compilateur :

L'organisation générale a été sensiblement modifiée, la notion de compilateur s'étendant à celle plus large de "système conversationnel de compilation".

La prise en compte des directives de compilation nous a conduit à l'écriture d'un module apte à commuter la lecture d'un périphérique à l'autre, à lire sélectivement les enregistrements sur un fichier en entrée et à réaliser des tâches de service telles que l'impression d'un listing, la génération des instructions en sortie, etc...

Ce module assume à lui seul les servitudes découlant des différents modes de travail et décharge le compilateur de toute opération d'entrée-sortie.

L'organisation est présentée schématiquement par la figure 5.1.

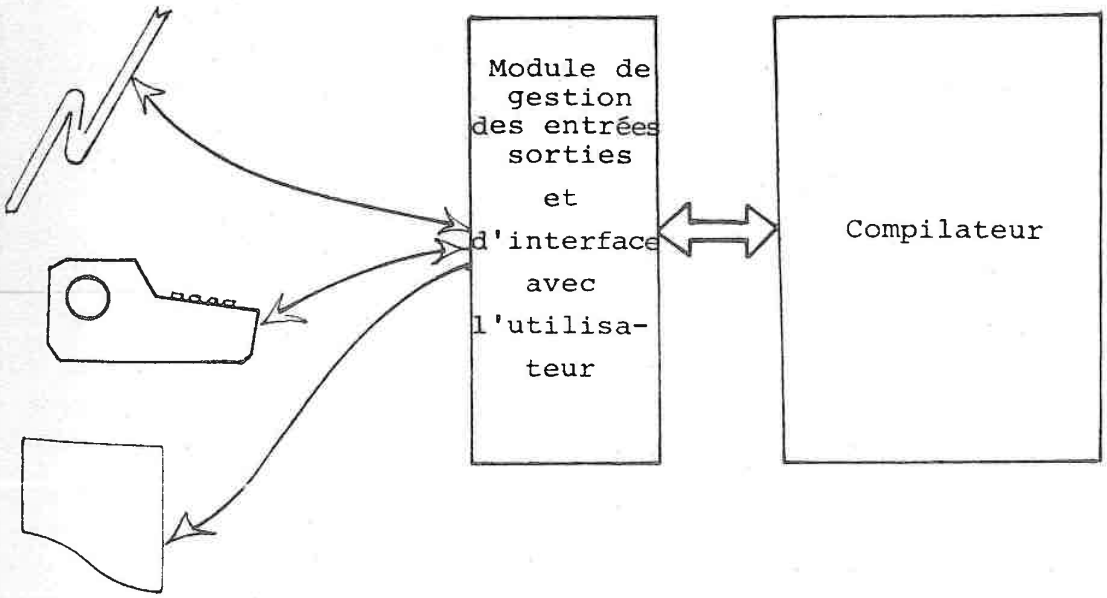


Figure 5.1

Une autre exigence du mode de compilation conversationnel modifie plus fondamentalement la logique du compilateur.

En effet, une instruction FORTRAN ne doit être prise en compte que si elle est indemne de toute erreur. Or, l'analyse d'une instruction nous amène au fur et à mesure à entreprendre des actions telles que la modification de la table des identificateurs et la génération. Il s'agira donc de différer ces actions jusqu'au moment où sera acquise la certitude que l'instruction est correcte, tant du point de vue de la syntaxe que de la sémantique, ce qui demande au minimum un passage complet sur l'instruction.

Faut-il alors éviter un nouveau passage sur l'instruction et comment faire dans ce cas pour réaliser les actions élémentaires introduites ?

Une solution possible consisterait lors du premier passage à stocker au fur et à mesure les informations utiles. Une autre possibilité serait de générer un texte intermédiaire plus concis qui nous permette de tirer parti des résultats de l'analyse, en ce qui concerne par exemple la reconnaissance du type d'une instruction et de ses structures partielles. Une représentation de l'arbre syntaxique pourrait convenir.

De telles solutions nécessiteraient cependant un surcroît d'espace de travail pour la construction, la représentation et l'exploitation de ces formes intermédiaires. Les contraintes d'espace mémoire nous ont fait rejeter ces solutions au profit de celle qui consiste simplement à réanalyser le texte source de l'instruction en gardant toutefois le bénéfice de la reconnaissance de son type.

Ainsi, à toute instruction FORTRAN, nous associons une procédure d'analyse spécifique qui fait appel aux procédures d'analyse correspondant aux structures partielles de l'instruction. Chaque procédure appelée introduit un certain nombre d'actions élémentaires et de contrôles conditionnées par le numéro du passage sur l'instruction.

On exécute deux fois la procédure attachée à une instruction :

- la première exécution effectue les contrôles seuls, les actions étant ignorées ;

- la seconde exécution ignore les contrôles, mais réalise les actions élémentaires.

3.3. La segmentation :

3.3.1. Généralités :

Globalement, un compilateur peut être décomposé en plusieurs sections, chacune jouant un rôle spécifique telles que l'analyse et la génération.

On peut facilement assimiler ces sections à des sous-programmes ayant chacun ses données et ses résultats propres.

Lorsque le compilateur tient facilement dans la mémoire centrale d'un ordinateur, ces sous-programmes sont appelés de manière périodique, de sorte qu'une partie du programme d'entrée est complètement compilée avant que le reste ne soit examiné.

Dans le cas d'une implantation sur petit calculateur, de taille mémoire limitée, il est plus opportun de segmenter le compilateur suivant ces sous-programmes, chacun accomplissant sa tâche en totalité avant l'appel du suivant.

3.3.2. Le langage intermédiaire :

Les tâches d'analyse et de génération ne se déroulant alors pas simultanément, a priori un seul passage sur le programme-source ne suffit pas.

Dans le paragraphe 3.2.6., l'opportunité d'introduire à l'étape d'analyse un langage intermédiaire pour éviter de réexaminer le texte source a été étudiée. Le problème se pose ici à nouveau, mais en d'autres termes car les contraintes d'encombrement mémoire sont moins aiguës. En effet, compte-tenu de sa longueur, le texte intermédiaire devra être stocké sur disque et donc n'entamera en rien la ressource mémoire. De plus, l'espace nécessaire à l'écriture des modules de production et d'exploitation de ce langage sera réparti entre deux segments du compilateur. La solution mettant en jeu un texte intermédiaire étant adoptée, il reste à définir le contenu de celui-ci.

Pour mettre en évidence les informations à transmettre au module de génération du compilateur, on se propose de faire l'inventaire des objets rencontrés dans un programme FORTRAN.

Dans un source FORTRAN apparaissent :

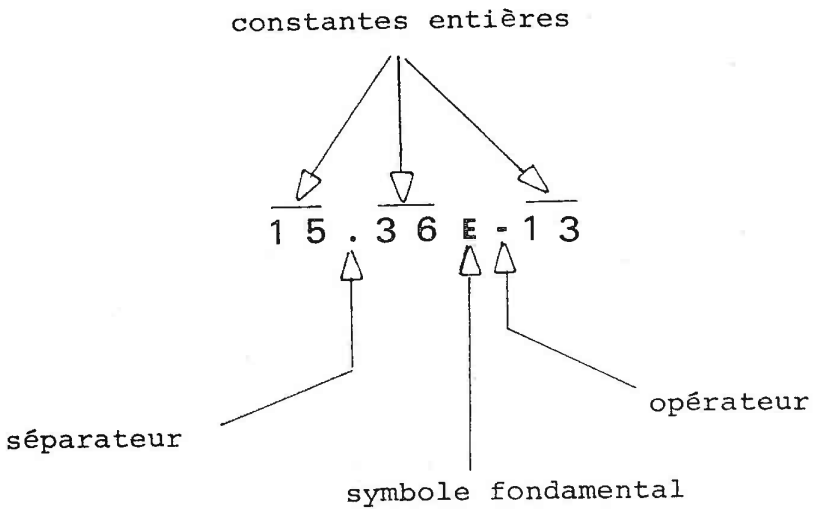
- des symboles clés : exemples : GOTO, IF, FORMAT, ... ;
- des identificateurs ;
- des constantes : numériques, logiques, alphanumériques ;
- des étiquettes ;
- des séparateurs ou délimiteurs : exemples : ',' , '('';')' ;
- des opérateurs : exemples : + ; - ; / ; .AND. ; .NOT.

Faisons quelques remarques :

- les séparateurs et délimiteurs n'ont plus d'intérêt tels quels lorsque l'étape d'analyse a effectué le découpage en opérandes d'une instruction ;

- les symboles fondamentaux ainsi que les opérateurs ne traduisent que des règles particulières de production. On peut convenir d'affecter à chaque règle un numéro qui la désigne de manière unique ;

- une constante numérique peut toujours se ramener à une suite de constantes entières, de séparateurs, d'opérateurs et de symboles fondamentaux ; exemple :



- les valeurs .TRUE. et .FALSE. d'une constante logique peuvent être assimilées à des symboles fondamentaux et entrer dans cette catégorie ;

- une étiquette d'instruction n'est rien d'autre qu'une constante entière avec un sens particulier.

Il apparaît ainsi qu'un programme FORTRAN peut se réduire à une suite de :

- numéros de règles,
- identificateurs,
- constantes entières,
- constantes alphanumériques.

Cette nouvelle forme n'est rien d'autre qu'une représentation post-fixée (cf. 1-2-3) d'un programme FORTRAN dans laquelle les identificateurs, constantes entières et constantes alphanumériques sont considérés comme terminaux de la grammaire.

3.3.3. *Production du langage intermédiaire* :

Intéressons-nous maintenant à la manière dont un tel langage est produit à l'analyse du texte source.

Les identificateurs, constantes entières et alphanumériques sont générées par leur procédure d'analyse respective au fur et à mesure de leur saisie dans le programme source.

Les numéros de règles sont générés par les procédures de l'analyseur, conformément aux règles de production que celles-ci expriment.

Toutes ces générations font partie des actions élémentaires à entreprendre à l'analyse lors du deuxième passage sur l'instruction.

Soit le sous-ensemble de règles suivant où la notation [ ] désigne les numéros de règles correspondants :

```
<listconstant>  ::=  <constant>  [1]  |  <constant> <virgule>
                                              <listconstant>  [2]
<constant>      ::=  <chiffre>   [3]  |  <chiffre>  <constant> [4]
```

Les procédures d'analyse associées s'écrivent :

```
procédure  analistconstant  début  analconstant ;
             ttq C = ' ' faire  début  générer ('2') ; lire (C) ;
                                fin ;  analconstant ;
             générer ('1')
                          fin
```

```
procédure  analconstant  début  générer ('$') ;
             ttq C = chiffre  faire  début  générer (C); lire(C) fin
             générer ('@')
                          fin
```

Dans ce qui précède, les caractères ¢ et @ sont utilisés comme délimiteurs dans le texte intermédiaire.

Cet exemple montre que la production du langage intermédiaire s'intègre facilement dans les procédures d'analyse du source FORTRAN.

3.3.4. Analyse du langage intermédiaire :

La grammaire du langage intermédiaire est donnée ci-dessous :

```
    <prog> ::= <listelem>
<listelem> ::= <elem> | <elem><listelem>
<elem>     ::= <regle>|<pref1><constant><suf>
                      <pref2><ident>   <suf>
                      <pref3><constalpha><suf>
<regle>    ::= <constant>
<constant> ::= <chiffre>|<chiffre><list chiffre>
<ident>    ::= <listcar>
<constalpha>::=<listcar>
<listcar>  ::= <car>|<car><listcar>
<pref1>    ::=  '¢'
<pref2>    ::=  '↑'
<pref3>    ::=  '/'
<suf>      ::=  '@'
<chiffre>  ::=  '0'|'1'|'2'|...|'9'
<car>      ::= <chiffre>|'A'|'B'|...|'Z'|'+' ...
```

Cette grammaire appelle quelques remarques :

- les caractères ¢, @ , ↑ , / ont pour rôle de délimiter les différents éléments et de lever les ambiguïtés possibles entre
  <constant> et <constalpha> d'une part et
  <ident> et <constalpha> d'autre part ;

- il n'est pas utile de préciser la syntaxe exacte de ident , cette catégorie syntaxique ayant déjà fait l'objet d'un contrôle à l'étape d'analyse.

Les procédures d'analyse associées s'écrivent :

```
procédure  analprog  début  ttq  C ≠ '⊣'  faire  analelem  fin

procédure  analelem  début  si C = '$'  alors  début  lire (C) ;
                                                      analconstant
                                                    fin ,
                            si C = '↑'  alors  début  lire (C) ;
                                                      analident
                                                    fin
                            si C = '/'  alors  début  lire (C) ;
                                                      analconstalpha
                                                    fin
                            sinon  analregle
                       fin

procédure  (analconstant, analregle)  debut
             ttq C = chiffre  faire  analchiffre ; lire (C)
                                   fin

procédure (analident, analconstalpha)  début
             ttq C ≠ '@'  faire  analcar ; lire (C)
                                 fin
```

Les procédures analchiffre et analcar se résument à la lecture d'un caractère et à un traitement spécifique, par exemple assemblage de la valeur binaire d'une constante entière ou rangement dans une zone de travail d'une constante alphanumerique.

Ces procédures d'analyse sont toutes déterministes et non récursives. La simplicité du langage rend l'analyseur performant, tant du point de vue temps d'exécution que du point de vue encombrement mémoire.

3.3.5. La_génération_du_programme_objet :

3.3.5.1. Principe

La génération du programme objet nécessite quelques choix importants que l'on justifiera.

Pour mieux appréhender les problèmes posés, examinons les tâches élémentaires à réaliser.

Le programme objet génèré manipule des opérandes disposés en mémoire et dans les registres pour l'évaluation des expressions arithmétiques et logiques. Il a recours au besoin à des sous-programmes de bibliothèque et doit traduire aussi la structure mise en évidence par les instructions de séquencement dans le programme FORTRAN. Il réalise enfin les actions élémentaires du programme source, affectation, lecture, écriture, etc...

Toutes ces tâches relèvent étroitement du problème de la traduction.

Avant d'en arriver à un programme machine exécutable, il aura fallu cependant :

- satisfaire les références en avant,
- assembler les constantes,
- gérer le compteur d'emplacement,
- assurer le lien avec les sous-programmes de bibliothèque, etc...

Les contraintes de mémoire nous amenant de toute façon à segmenter cette phase de génération, on a jugé intéressant de constituer deux segments qui prennent en charge séparément ces deux types de problèmes, le premier segment accomplissant sa tâche avant l'appel du suivant. Il apparaît d'autre part que le

travail du second segment peut très bien être réalisé par un assembleur, pour peu que le premier segment délivre un programme écrit en langage d'assemblage.

Cette solution étant adoptée, dans une première approche, la compilation peut se décrire schématiquement de la manière suivante (figure 3.2) :

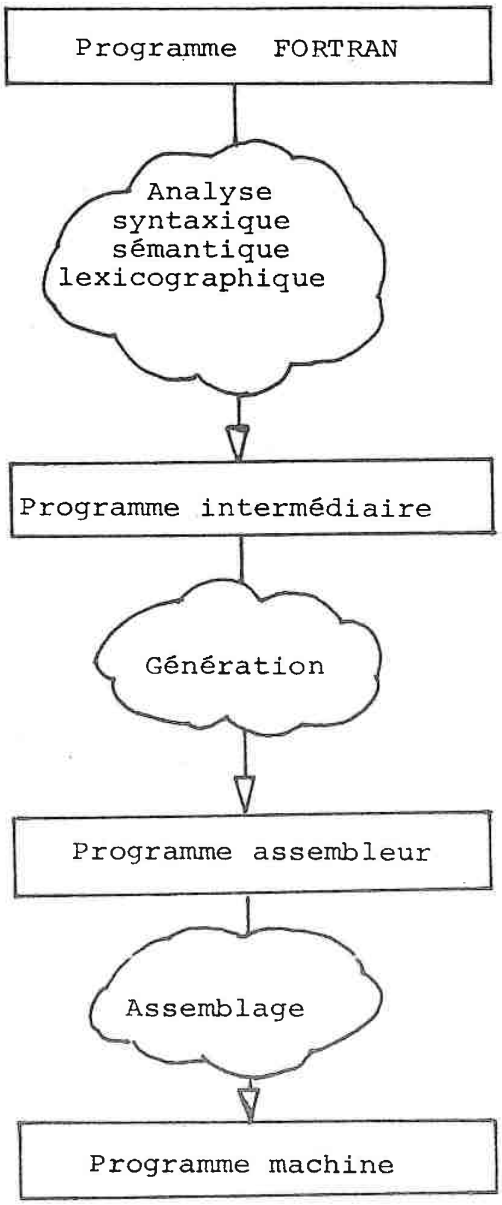


Figure 3.2

Dans l'enchainement des étapes, ce schéma ne tient pas compte d'un aspect important de FORTRAN, le découpage du programme en unités indépendantes. A l'exception du nom des zones "common" et des sous-programmes, les symboles utilisés sont locaux à une unité. Aussi, à condition de les exploiter directement, les tables des symboles et des étiquettes peuvent être réinitialisées chaque fois que l'on change d'unité.

Pour des raisons d'encombrement mémoire, nous avons jugé utile de tirer parti de cette possibilité, la contrainte en étant de générer les séquences assembleurs correspondant aux différentes unités au fur et à mesure de leur analyse.

Un autre avantage de cette solution est de permettre de réinitialiser le fichier temporaire utilisé pour stocker le langage intermédiaire, chaque fois que l'on change d'unité.

Nous illustrons l'enchainement de toutes les étapes de la compilation sur l'exemple de la figure 3.3.

Au total, deux fichiers-disques sont utilisés :

- l'un pour permettre de stocker le langage intermédiaire, puis le programme machine ;
- l'autre pour contenir le programme assembleur obtenu en mettant bout à bout les séquences partielles.

3.3.5.2. Traitement de caractères à la génération du programme assembleur

Cette étape de la compilation, par la nature même de son résultat, exige une manipulation importante de chaines de caractères.

Le calculateur étant dépourvu d'instructions de manipulation d'octets, l'accès par programme a été réalisé de la manière suivante :

Deux sous-programmes correspondant aux opérations spécifiques de lecture et d'écriture d'un octet ont été écrits.

De plus, une adresse fictive d'octets a été introduite avec les conventions suivantes :

- le premier octet de la mémoire a pour adresse 0 ;
- le i^e octet (i = 1, 2, 3) d'un mot d'adresse n ($n \geq 0$) a pour adresse d'octet 3n + i - 1.

Les modalités d'appel des deux procédures d'accès à l'octet sont alors les suivantes :

pour l'écriture : si R désigne le registre qui contient l'octet à ranger
R+1 contient l'adresse de rangement ;

pour la lecture : R+1 contient l'adresse de chargement
au retour R contient l'octet lu.

Les caractères étant disposés de manière contiguë en mémoire et généralement traités de façon séquentielle, l'intérêt de l'introduction d'une adresse d'octet est de permettre le passage de l'adresse d'un caractère à celle du suivant par une simple incrémentation ou décrémentation d'une unité.

3.3.6. La segmentation à l'étape d'analyse :

L'importance des traitements effectués à l'étape d'analyse a nécessité l'introduction d'un niveau supplémentaire de segmentation. Le découpage est le suivant :

- un premier segment regroupe les procédures relatives aux instructions FORTRAN non exécutables ;

- un deuxième segment se compose des procédures mises en jeu lors de l'analyse des instructions FORTRAN exécutables.

L'expérience montre que, pour des raisons de définition du langage ou d'habitude du programmeur, les instructions de déclaration sont habituellement disposées en tête d'une unité FORTRAN. La prise en compte de toute une unité n'exige alors que deux chargements de segments.

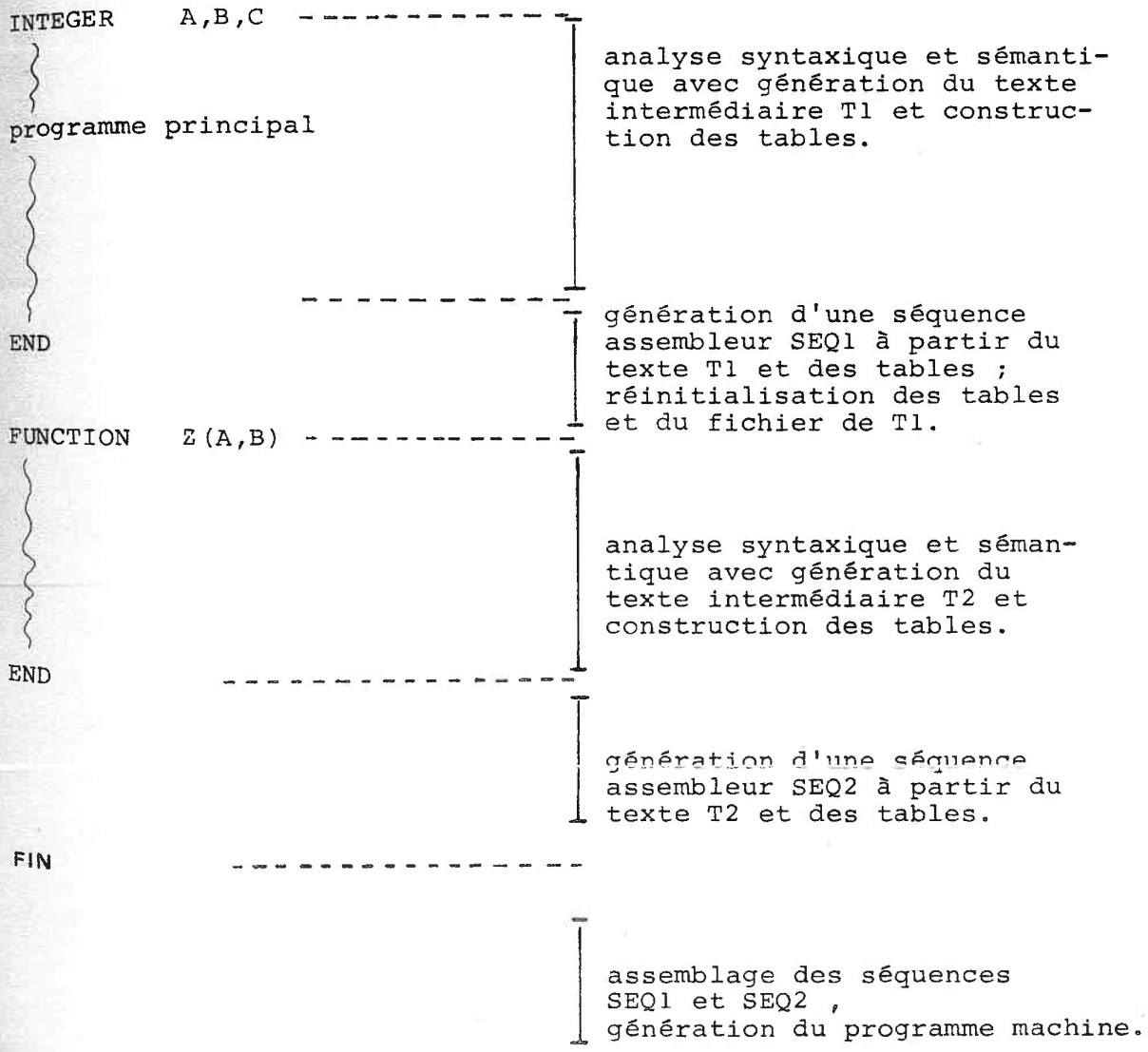


Figure 3.3.

3.3.7. L'arbre de recouvrement :

La segmentation est résumée par l'arbre de la figure 5.4.

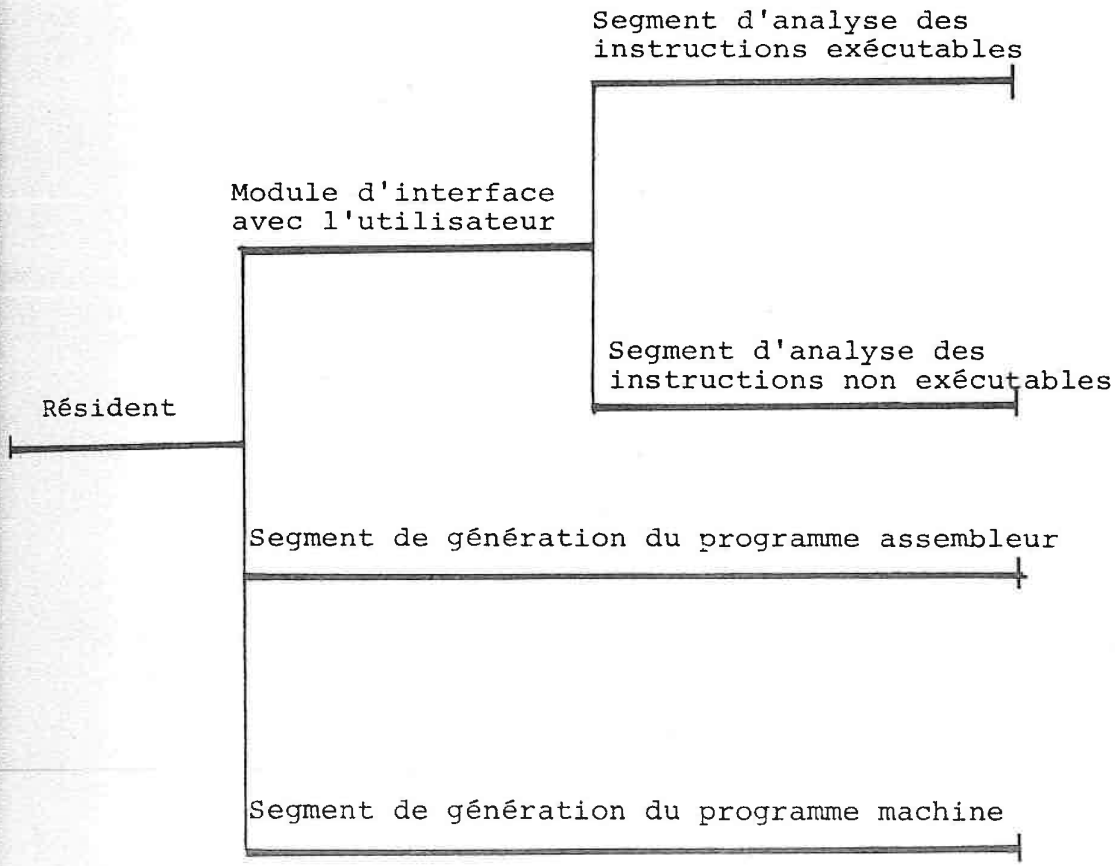


Figure 5.4

3.3.8. Conclusion :

Du point de vue de l'utilisateur, une compilation apparaît comme une suite de périodes pendant lesquelles il sera alternativement actif (frappe au clavier d'instructions FORTRAN, correction d'erreurs) puis passif (génération à la fin d'une unité FORTRAN de la séquence assembleur correspondante). Un certain nombre de dispositions ont déjà été prises pour minimiser la durée des périodes où l'activité de l'utilisateur n'est pas requise. Citons parmi celles-ci l'introduction d'un langage intermédiaire plus simple à analyser et la décomposition de la génération du programme-objet en deux étapes. Dans le chapitre qui suit, d'autres mesures seront évoquées.

CHAPITRE IV

REALISATION DE L'ANALYSEUR

Dans le chapitre I, nous avons décrit le support théorique de cette étude. Nous évoquons les problèmes soulevés par son application, et présentons les solutions retenues.

Dans le cadre de la théorie exposée dans le chapitre I, l'écriture d'une grammaire est l'étape préliminaire à la réalisation de l'analyseur. La représentation BNF de la grammaire que nous avons définie est donnée en annexe A1.

4.1. L'analyse syntaxique :

Choix d'une méthode d'analyse syntaxique :

Il nous a semblé vain de faire entrer des considérations de performance dans la discussion du choix d'une méthode d'analyse syntaxique. En effet, cette analyse s'opérant dans un contexte de conversation avec l'utilisateur, c'est-à-dire avec des temps d'attente et d'entrée-sortie importants, la plus grande rapidité d'un algorithme par rapport à un autre ne pouvait avoir qu'une faible incidence sur le temps total de passage sur la machine.

Dès lors, ce sont des considérations de modularité et de clarté de programmation qui ont conduit au choix d'une méthode d'analyse descendante. La comparaison de leurs algorithmes généraux respectifs en 1.3.1.2. et 1.3.2.2. montre que les méthodes d'analyse ascendante et descendante sont sensiblement différentes sous cet angle.

Naturellement modulaire et reflétant bien la syntaxe du langage, l'algorithme d'analyse descendante se prêtait mieux à notre sens à une programmation rapide et claire.

Dans la réalisation pratique de l'analyseur syntaxique, il restait alors à résoudre les problèmes de l'indéterminisme et de la récursivité qui font l'objet maintenant de la suite de l'exposé.

4.1.1. Traitement de l'indéterminisme :

Nous avons déjà évoqué quelques techniques de réduction de l'indéterminisme. Illustrons les maintenant en nous appuyant sur quelques exemples tirés de la grammaire de FORTRAN.

4.1.1.1. Lecture d'un caractère à l'avance et mise en commun d'une structure

Rappelons quelques règles de production relatives à l'instruction EXTERNAL.

```
<external>   ::=  <EXTERNAL> <listid>   Ret
<listid>     ::=  <id>  |  <id> <virgule> <listid>
<virgule>    ::=  ','
<EXTERNAL>   ::=  'EXTERNAL'
```

Ret désigne ici le caractère "retour chariot" utilisé comme délimiteur de l'instruction.

La procédure d'analyse associée à <listid> s'écrit :

```
procédure  analyse listid  début
                    choix   (analyse id,
                               début  analyse id ;
                                      analyse virgule ;
                                      analyse listid
                               fin)
                    fin
```

Il apparaît que les deux alternatives possibles commencent par l'analyse de la même structure syntaxique <id> ; on peut donc repousser le choix au retour de celle-ci. A ce niveau de l'analyse, il suffit d'anticiper la lecture du caractère suivant et de le tester pour être sûr de faire d'emblée le seul choix possible ; en effet le premier choix conduit nécessairement à la comparaison du caractère à lire avec RET , tandis que le deuxième nous amène à vérifier que le caractère est bien une virgule.

D'où la nouvelle procédure d'analyse dans laquelle l'alternative a disparu.

```
procédure  analyse listid   début   analyse id ; lire(c) ;
           si c = ','   alors     analyse listid
           sinon   retour
                              fin
```

Remarquons que dans le cas où le caractère testé est différent de la virgule, le retour à la procédure appelante se fait avec un caractère lu à l'avance. De telles situations étant fréquentes, il s'avère utile d'adopter la convention suivante :

"l'entrée dans toute procédure d'analyse se fait avec la lecture d'un caractère à l'avance".

La procédure précédente se réécrit alors :

```
procédure  analyse listid   début   analyse id ;
           si c = ','  alors   début  lire (c) ; analyse listid fin
           sinon  retour
                              fin
```

Si, dans la majeure partie des cas l'indéterminisme se résoud ainsi, il subsiste cependant des situations qui échappent à cette forme de réduction.

4.1.1.2. Indéterminisme

Cas_des_expressions_généralisées :

Reprenons la règle de production des "expressions généralisées"

<expgen> ::= <expbool>|<exparith>|<constalpha>

En FORTRAN la définition d'un paramètre effectif à l'appel d'un sous-programme répond à la notion d'expression généralisée et rien ne nous renseigne a priori sur le type et la forme d'un tel paramètre.

Il apparaît que les techniques de lecture de k caractères à l'avance s'appliquent mal ici. En effet, soient les expressions arithmétiques et booléennes suivantes :

```
B ** 2 - 4 * A * C
B ** 2 - 4 * A * C . EQ . 0 .
```

Le choix entre les deux premières possibilités dans l'analyse de <expgen> ne peut se faire qu'en fonction de la lecture des caractères '.EQ.' La structure commune relativement courte ici pourraît être arbitrairement longue et complexe.

Il nous a semblé préférable d'avoir recours alors au traitement successif des deux cas possibles avec retour arrière en cas d'échec. Cette solution, si elle s'avère lourde du fait du retour en arrière possible, a l'avantage d'être d'une mise en oeuvre simple et donc peu couteuse en place mémoire.

Dans l'essentiel elle se ramène à :

- sauvegarder, au moment du choix, le contexte de l'analyse constitué de quelques variables, telles que le pointeur sur le caractère courant dans la donnée ;

- noter les différents choix qui ont conduit à un échec ;
- restaurer, en cas d'impasse, le contexte sauvegardé à l'instant du choix.

Reconnaissance des instructions FORTRAN :

Rappelons qu'en règle générale une instruction en FORTRAN est reconnaissable à ses caractères initiaux qui constituent un symbole clé tel que :

GOTO , IF , FORMAT , etc...

Seules font exception à cette règle les instructions d'affectation et de définition de fonctions formules. Précisons que les premières ont pour objet d'affecter la valeur d'une expression généralisée à une variable du programme et que les secondes permettent la définition formelle d'une fonction mathématique.

Les principales règles de production relatives à ces deux instructions sont données ci-dessous :

- pour l'affectation :

```
<affec>      ::=   <var> <signe égal> <expgen>
<var>        ::=   <id> | <id> <parenthouv>
                        <list eaparith> <parenthfer>
<signe égal> ::=   '='
<parenthouv> ::=   '('
<parenthfer> ::=   ')'
```

- pour la définition de fonction formule :

```
<foncform>  ::=  <id><parenthouv><listid><parenthfer>
                    <signeegal> <expgen>
```

Il apparaît sur ces éléments que la syntaxe des deux instructions peut être rigoureusement identique.

Par exemple :

PROD (I,J) = I * J correspond syntaxiquement aussi bien à l'une qu'à l'autre des instructions.

Seule une considération d'ordre sémantique permet de lever l'ambiguité.

En effet, si l'identificateur du premier membre est déclaré comme identificateur de tableau, il ne peut s'agir que d'une affectation, sinon il doit s'agir d'une définition de fonction formule.

En ne prenant en considération que ce qui a été dit jusqu'ici, un algorithme possible de reconnaissance consisterait à décider que :

- la présence d'un symbole fondamental en début d'instruction détermine son type ;
- l'absence d'un tel symbole conduit nécessairement à une instruction d'affectation ou de définition de fonction.

En réalité, les choses sont un peu plus compliquées car les symboles clés n'étant pas réservés et le caractère espace non significatif, les constructions ci-dessous sont des affectations légales :

```
DO   10   I = 1
IF (I) = 3
CALL   AB (I,J) = 4
```

Un algorithme qui tient compte de ce nouvel élément est proposé par F.R.A. Hopgood [5].

Son principe repose sur la constatation suivante :

"dès que l'on a déterminé qu'une instruction FORTRAN n'est pas une affectation ou une fonction formule, il est possible de reconnaître n'importe quelle instruction à ses caractères initiaux".

L'auteur propose une façon simple de reconnaître une instruction d'affectation ou de définition de fonction : il s'agit de vérifier qu'il n'y a pas de virgule de niveau zéro à droite d'un signe '=' de niveau zéro.
Précisons qu'il est question ici de niveau de parenthèsage.

On peut reprocher à cette méthode de nécessiter un traitement préliminaire à l'analyse syntaxique et une première lecture complète de l'instruction.

Ces considérations nous ont amenés à préférer l'algorithme suivant, dans lequel deux cas sont envisagés :

- l'instruction ne commence pas par un mot clé :
  elle est analysée alors avec l'hypothèse qu'il ne peut s'agir que d'une affectation ou d'une définition de formule fonction ; un échec de l'analyse conduit à conclure à une erreur de syntaxe ;

- l'instruction commence par un mot clé :
  on analyse en supposant qu'on est en présence de l'instruction annoncée par le mot clé.
  Si l'analyse conduit à un succès, l'instruction sera reconnue.
  Sinon l'analyse est relancée avec l'hypothèse que l'instruction est une affectation ou une fonction formule.

  Un échec dans cette deuxième analyse nous fait conclure à une erreur de syntaxe.

Cet algorithme appelle quelques remarques :

- contrairement au précédent, il s'intègre parfaitement dans l'analyseur ;
- il réutilise la procédure de traitement d'un choix introduite pour les expressions généralisées ;
- le retour arrière a lieu seulement si le programmeur utilise des symboles clés pour l'identification de ses variables. L'expérience montre que ces situations sont relativement rares.

4.1.2. Traitement de la récursivité :

L'examen de la grammaire laisse prévoir de nombreux appels récursifs dans la procédure d'analyse syntaxique. Une étude plus approfondie nous a amené à distinguer deux types de récursivité :

- la récursivité de liste introduite chaque fois que l'on veut exprimer la règle de production d'une liste .

Exemples :

```
<listid>  ::= <id>|<id><virgule>
                          listid
<listexparith>  ::= <exparith>|<exparith><virgule>
                                  <listexparith>
```

- la récursivité réelle due à la nature même de la construction d'une structure et qui s'accompagne en général d'une imbrication de parenthèses.

Exemples : - les expressions arithmétiques et booléennes ;
- les DO implicites ;
- les spécifications de groupe dans un format d'entrées/sorties.

Nous allons décrire successivement dans chaque cas les solutions retenues.

4.1.2.1. La récursivité de liste

Reprenons l'exemple de l'identificateur ci-dessus et la procédure d'analyse associée.

```
procédure  analyse listid  début  analyseid  ;
                Si C = ',' alors  début  lire (C) ; analyselistid
                                         fin
                sinon  rien
                       fin
```

Il est important de remarquer qu'ici la récursivité est directe. On se rendra compte alors que l'algorithme suivant, dans lequel la récursivité a disparu, remplace avantageusement le précédent

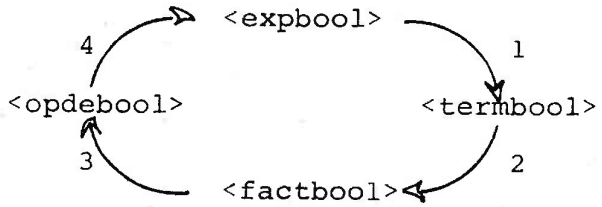
```
procédure  analyse listid  début  analyseid  ;
             ttq  C = ','  faire  début  lire(C) ; analyseid  fin
                           fin
```

4.1.2.2. La récursivité réelle

Un sous-ensemble des règles associées aux expressions booléennes est donné ci-dessous à titre d'exemple :

```
<expbool>  ::=  <termbool>  | <termbool><.OR.><termbool>
<termbool>::=  <factbool>  | <factbool><.AND.><factbool>
<factbool>::=  <opdebool>  | <.NOT.> <opdebool>
<opdebool>::=  <parenthouv><expbool><parenthform>
               <constbool> | <exprel>|<primbool>
<parenthouv>  ::=  '('
<parenthferm>::=  ')'
 .                  .
 .                  .
 .                  .
```

La récursivité, introduite par la règle de production de <opdebool> qui, dans le premier alternant, fait appel au non terminal <expbool> , est réalisée selon la boucle



On remarquera qu'il s'agit ici d'un appel récursif croisé.

Supprimer ce type de récursivité ne peut se faire qu'en remettant en cause l'algorithme général défini en 1.3.3.2. qui est parfaitement en accord avec les exigences de modularité et de sous-programmation que nous nous sommes fixées dans cette étude. Par ailleurs, des études ont montré [8] que c'est sous une forme récursive que la procédure d'analyse est la plus concise.

Nous conserverons donc la récursivité naturelle ; voyons alors comment la procédure d'analyse syntaxique écrite en assembleur peut s'en accomoder.

4.1.2.3. La récursivité dans le programme assembleur

Dans un programme assembleur, l'existence d'appels récursifs exige, pour chaque procédure mise en jeu, une gestion particulière des variables qui lui sont locales.

Cette gestion revient essentiellement à :

- empiler les informations à l'entrée de la procédure,
- les dépiler à la sortie.

En pratique, seule l'adresse de retour est réellement locale à une procédure d'analyse.

La gestion de toutes les autres variables nécessite uniquement un empilement et dépilement global. Ces deux opérations sont prises en charge par la procédure associée à la catégorie syntaxique de niveau le plus élevé et susceptible d'introduire la récursivité (<expgen> dans l'exemple ci-dessus).

4.1.3. Diagnostic et traitement des erreurs de syntaxe :

Diagnostic :

A l'analyse d'une instruction FORTRAN, une erreur de syntaxe est détectée lorsque les conditions suivantes sont réunies :

- un caractère lu est différent du caractère attendu,
- il n'y a pas d'autre hypothèse d'analyse.

Traitement :

L'apparition d'une erreur de syntaxe provoque l'arrêt de l'analyse et l'émission d'un message qui fournit à titre indicatif le rang du caractère à partir duquel la syntaxe de l'instruction analysée diffère de celle du langage.

Le mode de compilation conversationnel permet d'ignorer l'instruction erronée tout en offrant à l'utilisateur la possibilité de remédier directement à la situation d'erreur.

4.2. L'analyse sémantique :

4.2.1. Les déclarations :

Le rôle des instructions de déclaration est de définir le genre, le type et l'implantation des variables et des procédures. L'analyse sémantique consiste essentiellement à vérifier la compatibilité des déclarations relatives à un même identificateur, une table construite au fur et à mesure mémorisant les informations associées à un symbole déjà défini. Une simple

consultation de cette table permet en général de vérifier la validité d'une déclaration.

Cependant, le mode de compilation conversationnel amène des difficultés dans la détection et le traitement de certaines erreurs.

Considérons par exemple l'instruction FORTRAN suivante :

DIMENSION A (10 , 15) , A (20 , 10)

où l'identificateur A est supposé apparaître la première fois.

Dans un contexte de compilation non conversationnelle, la double déclaration dont est l'objet le tableau A peut très bien n'être mis en évidence qu'au moment de la prise en compte de la deuxième déclaration "A(20,10)", une simple recherche dans la table des identifications préliminaire à toute adjonction signalant à ce moment l'existence d'une déclaration A(10,15) antérieure et incompatible.

La première déclaration ayant été prise en compte, le traitement de l'erreur pourraît consister à ignorer la deuxième partie de l'instruction en le signalant à l'utilisateur, cette anomalie exigeant de toute façon une récompilation ultérieure.

Dans notre réalisation, un tel traitement était à exclure puisqu'une exigence du mode conversationnel est de pouvoir corriger directement ce genre d'erreur sémantique.

La solution adoptée consiste alors à vérifier, lors du premier passage sur l'instruction et à l'aide d'une liste construite au fur et à mesure de l'analyse, qu'il n'y a pas répétition d'un même identificateur dans les déclarations.

Il s'est avéré que la construction de cette liste et la vérification de la non répétition ont pu être confiées au module d'analyse lexicographique, attaché aux identificateurs pour peu que les conditions suivantes soient remplies :

- le traitement n'est effectué que pour les instructions de déclarations ;
- certaines répétitions légales ne sont pas considérées comme des erreurs.
  Ainsi DIMENSION A(N,N).

L'introduction d'un indicateur pour valider les contrôles y a suffit.

4.2.2. Les formules arithmétiques et booléennes :

La sémantique du langage nous conduit dans ce type de construction à vérifier essentiellement :

- la compatibilité de type des différents opérandes,
- la conformité entre l'utilisation d'un identificateur et sa définition,
- la validité du paramètre ou de l'indice dans une référence à un sous-programme ou à un tableau.

Pour mener à bien ces contrôles, nous avons associé à chaque opérande un descripteur qui nous renseigne sur deux de ses caractéristiques :

- son type : entier, réel, booléen, double précision, complexe ;
- son genre : constante, variable simple, variable indicée, variable fonction, expression.

Ces informations nous permettent dans une opération binaire :

- de vérifier par comparaison que les types des opérandes sont compatibles,
- d'évaluer le type du résultat.

Pour fixer les idées, nous présentons dans la figure 4.1 le tableau à deux entrées qui met en évidence les incompatibilités et fournit le type du résultat dans les opérations arithmétiques binaires.

Par suite, des imbrications possibles dans les expressions arithmétiques et logiques, nous avons été amenés à introduire une pile pour gérer les descripteurs.

1er opérande \ 2e opérande	entier	réel	double précision	complex
entier	entier	réel	double précision	complex (1)
réel	réel	réel	double précision	complex (1)
double précision	double précision	double précision	double précision	interdit
complex	complex	complex (1)	interdit	complex (1)

Figure 4.1

(1) Cette combinaison est interdite pour les opérations d'exponentiation.

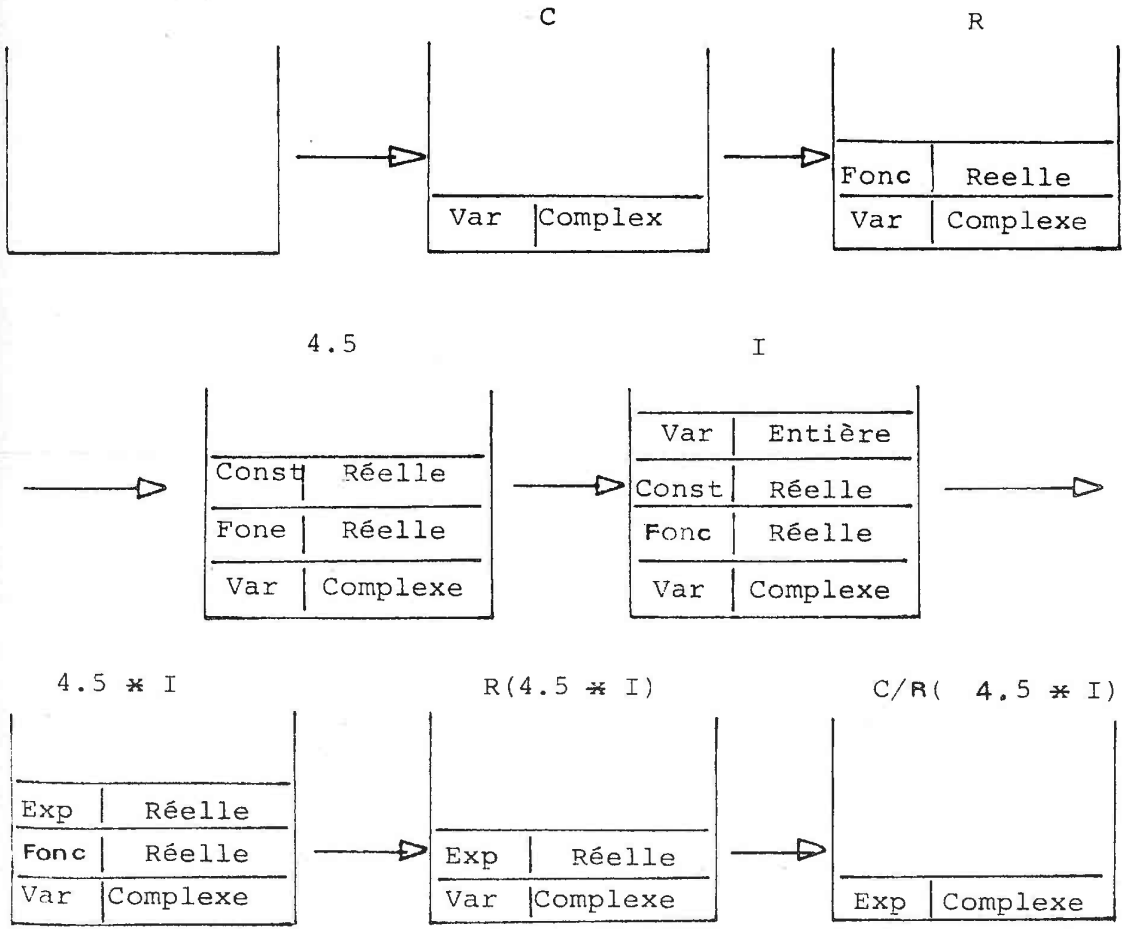
Les opérations élémentaires sur cette pile sont :

- à la saisie d'un opérande, empiler le descripteur associé ;
- à la réalisation d'une opération binaire, dépiler les deux éléments de sommet de pile et empiler le descripteur associé au résultat ;
- à la réalisation d'une opération unaire, dépiler le descripteur de l'opérande et empiler celui du résultat.

La gestion de cette pile est telle que :

- à la fin de l'analyse d'une structure partielle (indice, paramètre, etc...), l'élément de sommet de pile nous renseigne sur le type et le genre du résultat partiel ;
- à la fin de l'analyse d'une structure complète (expression booléenne, arithmétique second membre d'une affectation, etc...), la pile se réduit à l'élément descripteur du résultat final.

Nous donnons à titre d'exemple l'évolution de la pile lors de l'analyse de l'expression C/R (4.5 * I) où C et I désignent des variables simples respectivement complexe et entière et R une fonction réelle.



Remarques :

La gestion de l'information "genre" présente encore un autre intérêt lié, cette fois-ci, à l'analyse syntaxique.

En effet, certaines structures partielles en FORTRAN sont constituées de variables simples ou indicées, par exemple le premier membre d'une affectation ou la liste des variables d'entrée-sortie dans un ordre READ ou WRITE, ces catégories syntaxiques étant des sous-ensembles des expressions généralisées ; il est intéressant d'appliquer le module d'analyse de celles-ci à la reconnaissance de ces éléments. Ceci exige pourtant, en fin d'analyse, de s'assurer que la structure partielle était bien celle attendue (variable simple ou variable indicée), ce que l'on fait en vérifiant le genre du descripteur du résultat.

4.2.3. Les instructions de séquencement :

La sémantique des instructions de branchement inconditionnel "GOTO" et conditionnel "IF" ne posent pas de problèmes particuliers, les principales difficultés provenant des structures itératives "DO".

Rappelons à leur propos quelques détails de fonctionnement.

L'instruction "DO" sert à répéter un nombre de fois donné la séquence de programme qui la suit immédiatement. Le nombre de répétition est gouverné par la progression d'un indice dont l'incrémentation est assumée automatiquement à partir d'une valeur donnée jusqu'à atteindre un maximum fixé par le programme.

La dernière instruction de la séquence à répéter est repérée par une étiquette donnée dans le "DO".

Schématiquement, cette construction peut se représenter de la manière suivante :

```
DO   N   I = M1, M2, M3
     ≈
suite d'instruction
     ≈
N dernière instruction
```

où N désigne l'étiquette de fin de boucle, I l'indice de boucle et M1, M2, M3 respectivement ses valeurs initiale, finale et son incrément.

Compte-tenu des imbrications possibles de telles structures, une sémantique précise est définie en ce qui concerne les indices et étiquettes de fin de boucle.

Ses règles essentielles sont :

- un indice de boucle ne doit pas être utilisé dans une boucle plus interne ;
- si une structure itérative DO contient elle-même un autre DO, toutes les instructions de cette dernière doivent être incluses dans la première ; cette règle concerne évidemment l'ordre de définition des étiquettes de fin de boucle.

Ces imbrications respectant le principe :

"dernière itérative commencée, première terminée"

on est donc naturellement conduit, pour effectuer les contrôles, à utiliser une pile sur laquelle les opérations élémentaires illustrées par la figure 4.2 sont :

- empilement de l'étiquette et de l'identificateur de l'indice à la rencontre d'un "DO" ;
- dépilement de ce couple d'information à la rencontre de l'étiquette de fin de boucle.

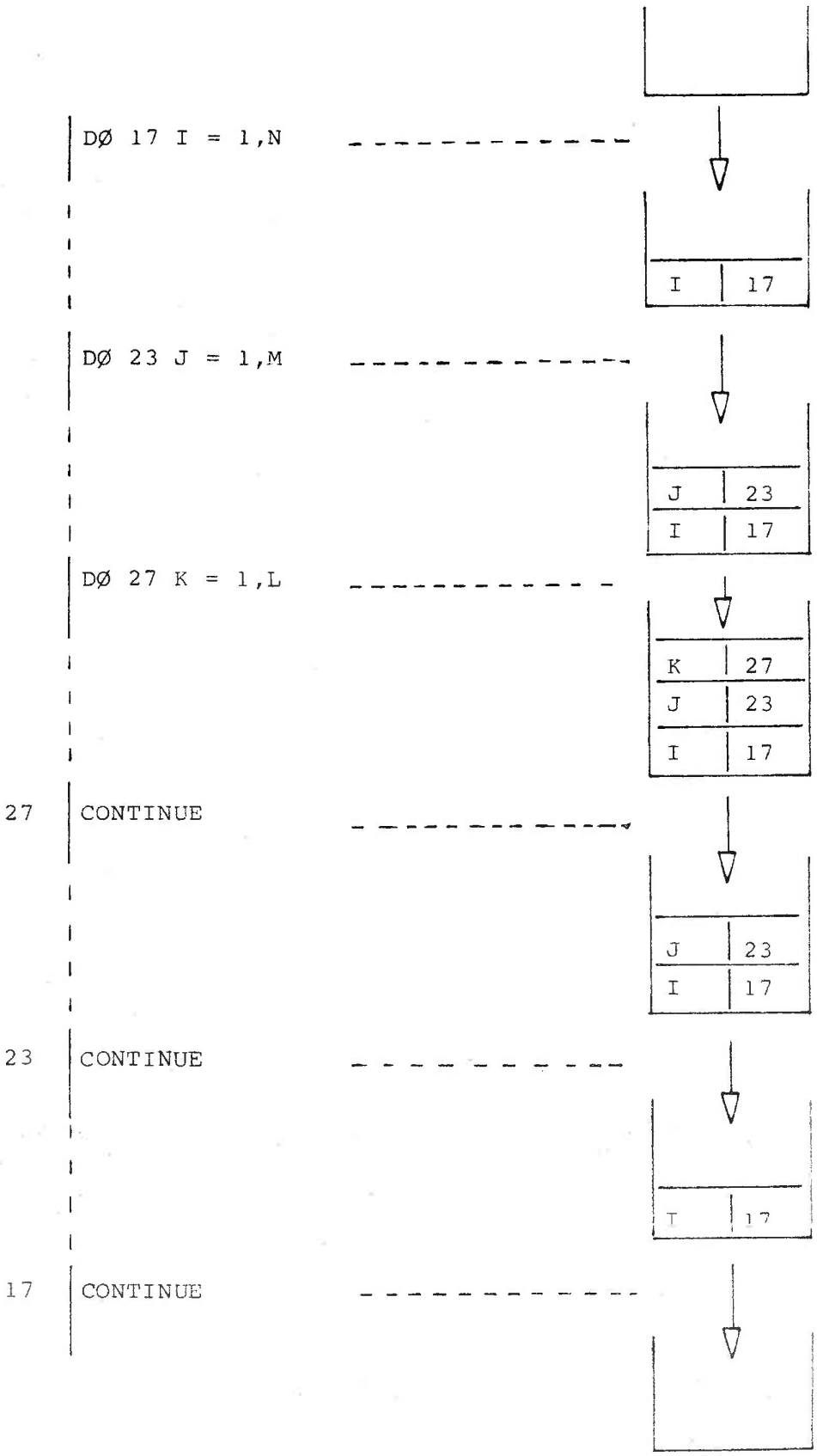

DØ 17 I = 1,N
I 17
DØ 23 J = 1,M
J 23
I 17
DØ 27 K = 1,L
K 27
J 23
I 17
27 CONTINUE
J 23
I 17
23 CONTINUE
I 17
17 CONTINUE

Figure 4.2

Les contrôles effectués sont dans l'ordre :

- à l'apparition d'un DO : vérifier que l'indice de boucle introduit ne se trouve pas déjà dans la pile et que, si elle est déjà apparue, l'étiquette de fin de boucle est le dernier élément de la pile ;
- à la définition d'une étiquette : vérifier que, si l'étiquette existe déjà dans la pile, elle s'y trouve au sommet.

Il faut tenir compte, dans ces algorithmes, du fait qu'une étiquette de fin de boucle peut être commune à plusieurs structures et donc qu'elle peut se trouver à plusieurs niveaux de la pile.

La pile utilisée est la même que celle qui sert à l'évaluation des expressions généralisées.

4.2.4. Traitement des erreurs sémantiques :

Les contrôles sémantiques sont menés parallèlement à l'analyse syntaxique lors du premier passage sur l'instruction et les erreurs éventuelles sont mémorisées.

Le bien fondé des diagnostics d'erreurs sémantiques supposant la reconnaissance du type et des structures partielles de l'instruction, la liste des erreurs sémantiques détectées n'est émise que si l'analyse syntaxique a conduit à un succès.

Dans le cas contraire, seule l'erreur de syntaxe est signalée.

Pour des raisons d'encombrement mémoire, au lieu d'un message en clair, le système ne fournit qu'un numéro d'erreur sémantique, un tableau des codes d'erreur avec leur signification étant donné dans la notice d'utilisation.

4.3. L'analyse lexicographique :

4.3.1. Les modules d'analyse lexicographique :

Le paragraphe 3.3.3. a montré que les procédures d'analyses attachées à des unités syntaxiques telles qu'une constante entière ou un identificateur jouent un rôle particulier dans la production du langage intermédiaire introduit entre l'étape d'analyse et de génération.

D'autres tâches sont encore confiées à ces mêmes modules qui constituent de réelles "primitives" d'analyse.

Signalons pour une constante entière l'assemblage de sa valeur binaire, pour un identificateur, sa recherche dans la table et le contrôle de non répétition dans une instruction de déclaration déjà évoqué en 4.2.1.

Ce rôle de primitive justifie en partie une petite entorse au principe de l'algorithme général d'analyse descendante dans la réalisation pratique de ces procédures.

Soit en effet les règles de production d'un identificateur :

```
   <ident>    ::=  <lettre> | <lettre> <listletchif>
<listletchif> ::=  <letchif> | <letchif> <listletchif>
  <letchif>   ::=  <lettre> | <chiffre>
   <lettre>   ::=   A |B ... |Z
  <chiffre>   ::=   0 |1 ... |9
```

L'algorithme général défini en 1.3.3.2. conduirait à l'écriture d'une procédure par non terminal ci-dessus.

La solution suivante s'avère cependant plus avantageuse en n'introduisant qu'une procédure d'analyse.

```
procédure analident début si c = lettre alors erreur syntaxe;
                          ttq c = lettre ou chiffre faire lire (c) ;
                          fin
```

Une certaine optimisation est encore apportée dans la réalisation des tests lettre et chiffre. On n'opère pas par comparaisons successives avec tous les caractères alphabétiques, respectivement numériques, mais on situe le code du caractère dans certaines plages qui correspondent à des domaines alphabétiques, respectivement numériques de la codification ASCII.

Une solution analogue est utilisée pour l'analyse d'une constante entière.

4.3.2. La table des identificateurs :

4.3.2.1. Les objectifs

Lors de la traduction, les consultations de cette table étant relativement fréquentes, on est tenté de choisir des techniques de rangement qui permettent de gagner du temps dans la recherche d'un élément. Les techniques de hashing [7] qui sont les plus performantes à ce point de vue ne conviennent pas ici puisqu'elles supposent des représentations assez couteuses en place mémoire.

Il s'agissait donc de trouver une structure d'information qui soit à même de garantir un bon temps d'accès dans un espace mémoire utilisé au mieux.

4.3.2.2. La recherche dichotomique

Rappelons brièvement le principe de la recherche dichotomique.

Soit une table de n éléments triée suivant un critère donné et α l'élément cherché.

Soit $[\![1,n]\!]$ une notation symbolique de cette table et E(q) la partie entière d'un rationnel q.

On procède comme suit.

La table est divisée symboliquement en deux sous-tables $[\![1 , E(\frac{n}{2})]\!]$ et $[\![E(\frac{n}{2})+1 , n]\!]$.

On situe, suivant le critère de tri, l'élément cherché α par rapport à l'élément "charnière" de rang $E(\frac{n}{2})$, ce qui permet, pour la suite de la recherche, de ne retenir qu'une moitié de table.

On itère cette procédure jusqu'à obtenir l'élément cherché ou une sous-table réduite à un élément.

Remarques :

- la recherche reste rapide ; par exemple, il suffit de 10 itérations en moyenne pour accéder à un élément dans une table à 1000 entrées ;
- en cas d'échec de la recherche, on dispose du rang que devrait occuper l'élément dans la table ;
- la méthode suppose qu'à tout instant la table soit triée.

4.3.2.3. L'insertion dichotomique

Pour obtenir une table triée, un moyen simple consiste à s'assurer que chaque adjonction d'un nouvel élément tient compte du critère de tri.

Une adjonction se traduit alors par une insertion et s'accompagne du déplacement de la partie de la table qui suit l'élément inséré. La recherche qui précède nécessairement chaque adjonction fournit le rang de l'insertion dans la table.

4.3.2.4. Table majeure et vecteurs complémentaires

Pour que la méthode soit valable, il est impératif que tous les éléments occupent une place de même amplitude dans la table. Or le nombre de renseignements attachés à un identificateur varie suivant son genre. Par exemple, en FORTRAN, un tableau exige, en plus des informations communes à toutes les variables, la connaissance de ses bornes, de son nombre de dimensions, etc... Pour atteindre ces renseignements complémentaires, nous avons été conduits à utiliser des techniques d'accès indirect par chainage à partir d'une table majeure.

La figure 4.3 schématise les implantations respectives de la table majeure et des vecteurs complémentaires dans l'espace libre alloué à la table des identificateurs.

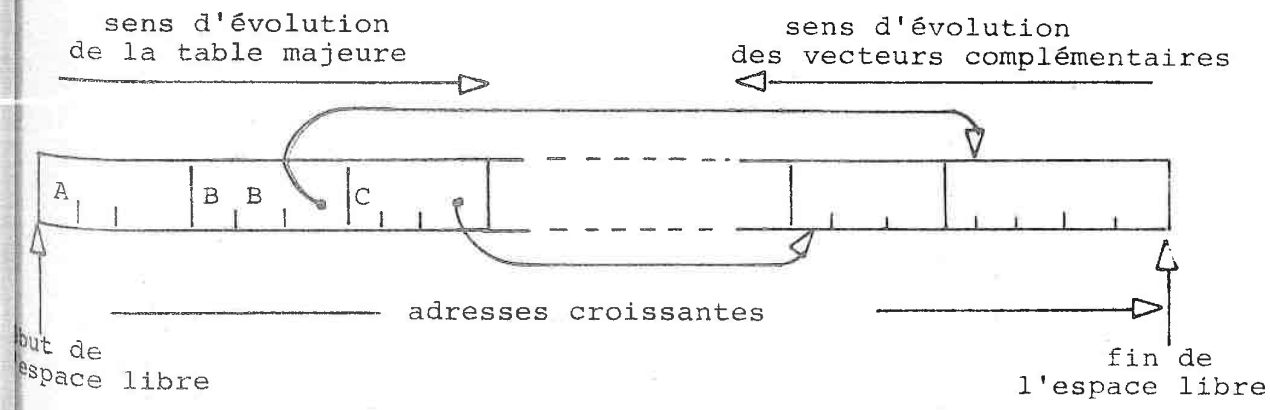


Figure 4.3

4.3.3. La table des étiquettes :

Parallèlement à la table des identificateurs, la procédure de compilation construit une table des étiquettes, essentiellement pour effectuer des contrôles sémantiques. Les doubles définitions d'étiquettes et les étiquettes non définies sont détectées.

Les consultations de cette table étant peu fréquentes, nous avons préféré à la rapidité d'accès une meilleure utilisation de l'espace mémoire disponible.

Table majeure et table des étiquettes sont ainsi contigües en mémoire, suivant le schéma de la figure 4.4.

Figure 4.4

Une extension de la table majeure empiète alors nécessairement sur la table des étiquettes. Pour libérer la zone recouverte, on se contente de déplacer les premiers éléments de la table des étiquettes vers la fin de celle-ci. La recherche séquentielle, seule à pouvoir s'accomoder de cette méthode de construction, est alors utilisée pour accéder à une étiquette.

L'exemple de la figure 4.5 illustre l'évolution globale des tables dans le cas de l'adjonction d'un identificateur (BB).

Précisons qu'une entrée dans la table des symboles occupe 3 mots mémoire contre un mot pour la table des étiquettes.

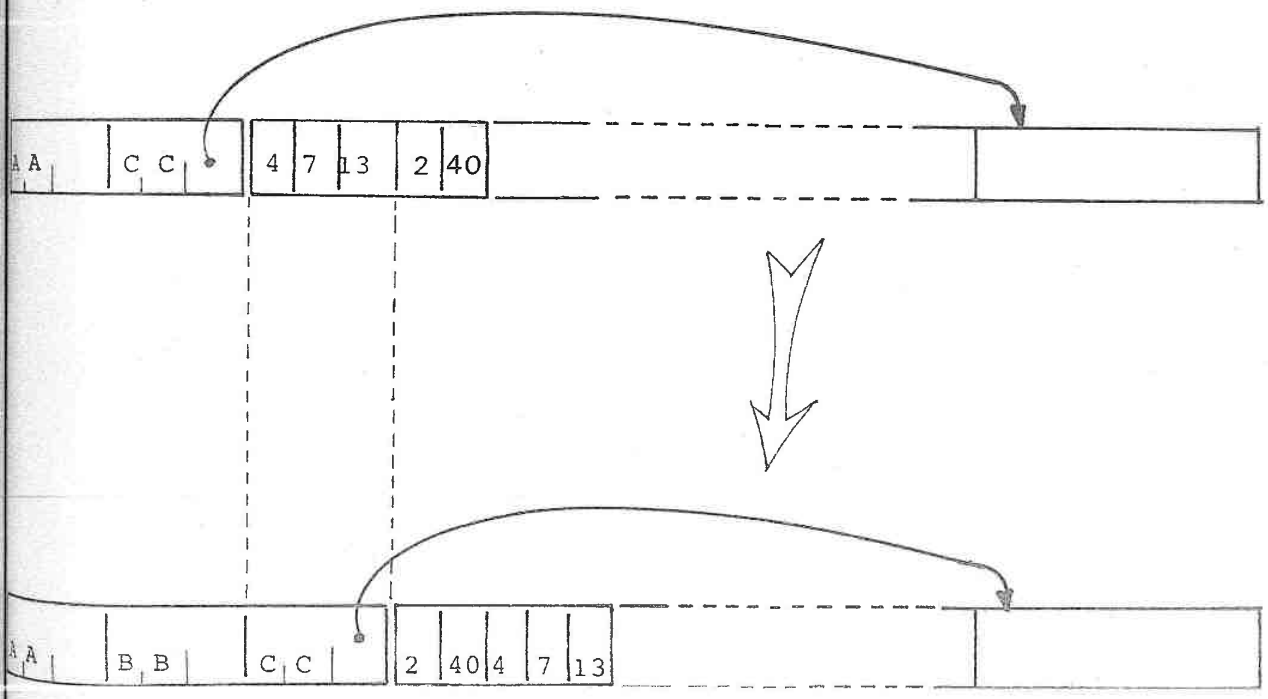


Figure 4.5

4.3.4. Remarque :

Dans cette réalisation, il était d'autant plus impératif de disposer d'une organisation de tables des identificateurs qui permette un accès rapide à l'information,que l'architecture du compilateur exige un nombre important de consultations.

A l'analyse, le mode conversationnel impose à lui seul pour chaque identificateur rencontré deux recherches dans la table :

- la première recherche a lieu lors du premier passage sur l'instruction et sert à opérer des contrôles sémantiques ;
- la deuxième recherche conduite lors du deuxième passage est nécessaire pour compléter le vecteur de renseignements, attaché à cet identificateur, avec les informations que renferme l'instruction.

D'autre part, la nécessité de dissocier la tâche d'analyse de celle de la génération exige une nouvelle recherche lors de l'étape de traduction en assembleur.

Les contraintes de taille mémoire expliquent enfin l'utilité d'une bonne gestion de l'espace libre.

CHAPITRE V

DEFINITION DU PROGRAMME OBJET ET STRUCTURE DES INFORMATIONS

Les problèmes posés par la définition du programme objet et la structure des informations sont évoqués dans ce chapitre.

5.1. Représentation des données :

L'inventaire des informations simples manipulées à l'exécution du programme objet est assez réduit. On distinguera :

- les constantes,
- les variables élémentaires,
- les tableaux.

La représentation des deux premiers types est immédiate et on ne s'y attardera pas. Elle dépend étroitement des choix faits pour la représentation des nombres en virgule fixe et flottante décrite en 2.3.4.1.

La représentation d'un tableau est parfaitement définie lorsque sont données :

- la représentation de ses éléments,
- la représentation de la fonction d'accès.

Dans cette réalisation, un élément de tableau a la même représentation qu'une variable simple de même type et l'accès se fait par adressage calculé.

Dans la compilation d'un programme en FORTRAN, l'allocation de la mémoire aux différents objets à représenter est statique et ne pose, dans son principe, pas de problèmes particuliers. Seul l'adressage des différents objets présente des difficultés.

5.2. Les problèmes d'adressage :

L'introduction d'un registre page opérandes et d'un registre page instructions pour pallier à l'impossibilité d'adresser avec un champ d'adresse de 12 bits, toute la mémoire disponible a été évoquée en 2.3.3.4.

A l'écriture d'un programme machine, la gestion de ces deux registres demande de connaître l'implantation en mémoire des données et du programme.

En langage d'assemblage, le programmeur est maitre de cette implantation et la gestion évoquée n'est pas trop lourde.

Il n'en n'est pas de même lorsque le programme machine est généré par un compilateur, surtout dans la perspective d'une bonne utilisation de la mémoire.

Pour les ruptures de séquences du programme objet, la solution adoptée consiste à procéder systématiquement par indirection sur un mot contenant l'adresse du saut.

Pour l'adressage des variables et des constantes, on a retenu la solution de la segmentation.

La segmentation désigne l'opération de découpage en segments des zones utilisées par un programme, le mot segment étant à prendre au sens donné par CROCUS [4] et n'impliquant aucune idée de recouvrement en mémoire.

Une adresse segmentée est constituée du couple :

- adresse origine du segment,
- déplacement dans le segment.

La résolution de cette adresse est réalisée comme suit :

- un registre général utilisé comme registre d'index pointe sur l'origine du segment,
- le champ d'adresse dans l'instruction (à indexation) fournit le déplacement.

Ces solutions dispensent le programme objet de gérer à l'exécution les registres page opérandes et instructions.

5.3. Les segments à l'exécution :

5.3.1. Définition :

Par définition, un segment regroupe des objets de même nature et de même durée de vie. Plusieurs types de segments peuvent être considérés alors :

- les segments de variables locales,
- les segments de variables temporaires,
- les segments de variables communes (COMMON FORTRAN),
- les segments de communication des variables paramètres,
- les segments de constantes.

L'indexation qui est le mode utilisé pour réaliser l'adressage segmenté ne permet d'atteindre que les 256 premiers mots d'un segment (cf. 2.3.4.3).

Un nom dont le déplacement par rapport à l'origine du segment est supérieur à 256 est atteint de la manière suivante.

Soit n un tel déplacement par rapport au début d'une zone pointée par un registre noté RX.

Si R désigne le registre dans lequel par exemple est chargé l'élément donné et RM un registre de manoeuvre, les opérations à effectuer sont les suivantes :

- chargement dans RM de la valeur immédiate n ;
- addition dans RM du contenu de RX ;
- chargement dans R du mot d'adresse contenue dans RM.

Pour éviter que cette situation ne se produise trop fréquemment, une certaine optimisation est apportée dans l'affectation mémoire des objets manipulés.

La première optimisation consiste à multiplier les segments dans une limite raisonnable.

En effet, chaque nouveau segment permet de définir une tranche de 256 mots d'accès privilégié, mais nécessite l'allocation d'un registre de base comme registre d'index.

La solution suivante nous a semblé constituer un bon compromis :

pour chaque unité FORTRAN on introduit :

- un segment de variables locales,
- un segment de variables temporaires,
- un segment de constantes,
- éventuellement un segment de communication des variables paramètres,

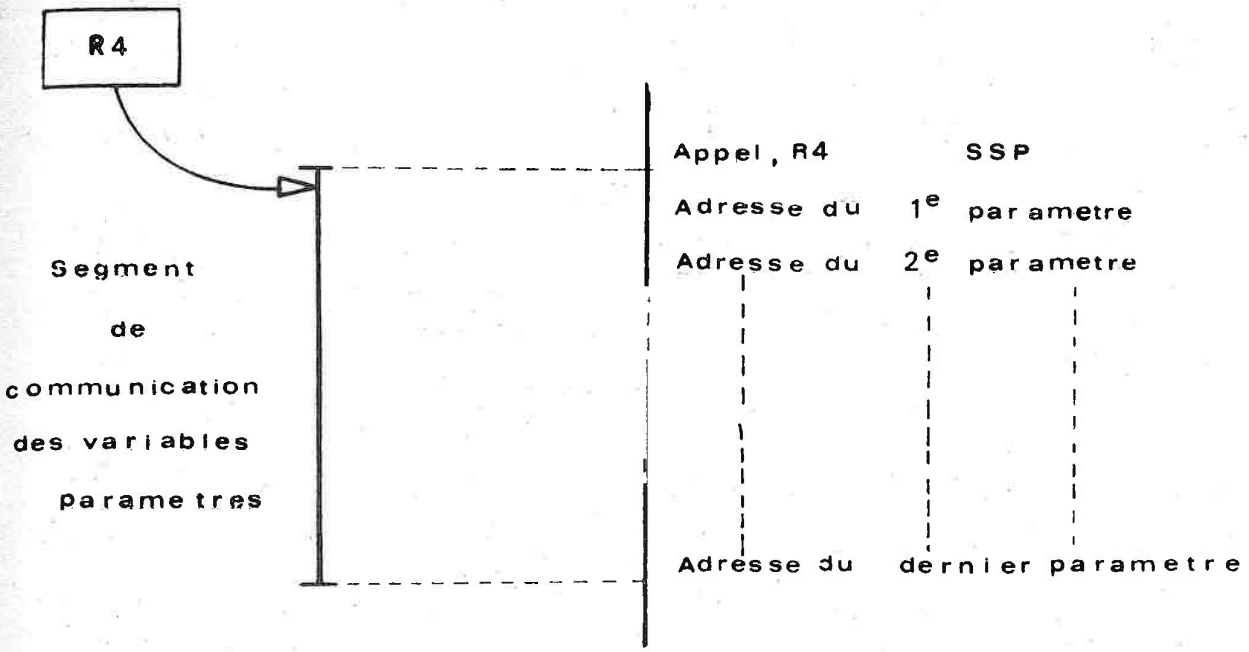
ce qui exige l'allocation de quatre registres que l'on notera respectivement R1, R2, R3 et R4.

5.3.2. *Structure des segments* :

En ce qui concerne les variables locales, les affectations mémoire sont effectuées de façon à disposer les variables simples en tête du segment. On accède ainsi facilement au plus grand nombre de variables.

Le segment des variables temporaires ne contient que des variables simples.

Les paramètres d'un sous-programme FORTRAN sont passés par nom. Le segment de communication est en fait la suite des adresses des paramètres effectifs disposés en séquence de l'instruction d'appel. C'est le registre de liaison qui joue le rôle de l'index.



L'accès à une variable paramètre se fait par indirection.

Pour un segment de constantes, nous n'avons pas introduit de stratégie particulière d'allocation.

5.3.3. *Environnement d'une unité FORTRAN :*

Chaque unité d'un programme FORTRAN est caractérisé par un environnement propre, composé du segment des variables locales, temporaires, du segment des constantes et, éventuellement, du segment de communication des variables paramètres.

Etudions la manière dont s'opèrent, à l'appel et au retour d'un sous-programme, les changements d'environnement.

5.3.4. *Changements d'environnement* :

Dans ce qui suit, l'"appelante" désigne l'unité FORTRAN qui réalise l'appel et l'"appelée" l'unité appelée.

A l'appel d'un sous-programme, le contexte de l'appelante, constitué en fait du contenu des registres R1, R2, R3 et R4, est sauvegardé dans le segment local associé à l'appelée.

L'environnement de l'appelée est mis en place par l'initialisation des registres R1, R2, R3 et R4 avec les adresses origines des segments auxquels ils sont associés.

Au retour d'un sous-programme, il suffit de restituer le contexte de l'appelante par restauration des registres R1, R2, R3 et R4.

5.4. Traitement des descriptions de FORMAT :

En FORTRAN, deux classes d'instructions peuvent être mises en évidence :

- les instructions de déclarations qui fournissent les renseignements sur les objets manipulés et qui se traduisent à la compilation par des mises à jour de la table des identificateurs ;
- les instructions exécutables qui décrivent le traitement à effectuer sur ces objets et que le processus de compilation traduit en autant de séquences du programme objet.

Une instruction pourtant échappe à cette classification : l'instruction FORMAT.

Rappelons qu'elle est associée à des opérations d'entrées-sorties et que son rôle est principalement d'indiquer le découpage de l'information, telle qu'elle apparaît ou doit apparaître sur le support physique de l'E/S et de préciser les conversions nécessaires pour passer de la représentation externe à la représentation interne et vice-versa.

Donc, une instruction FORMAT renferme des informations qui ne peuvent être exploitées qu'à l'exécution et beaucoup de compilateurs se contentent de la mettre telle qu'elle dans le programme objet, laissant à charge d'un module du système l'interprétation des spécifications qu'elle fournit.

La syntaxe de l'instruction étant assez complexe, les interprètes de FORMAT peuvent être encombrants.

Dans notre cas, nous avons jugé utile de procéder, lors de la compilation, à un pré-traitement de l'instruction FORMAT afin de faciliter son exploitation ultérieure et ainsi de réduire l'encombrement du module d'interprétation.

Ce traitement consiste, pour l'essentiel, à traduire l'instruction sous une forme plus simple à analyser.

La procédure de traduction est analogue à celle qui est utilisée pour produire le langage intermédiaire à l'étape de compilation et qui est décrite en 3.3.2. et 3.3.3.
Le résultat est en effet une représentation post-fixée des spécifications de format.

La nouvelle forme plus simple à analyser se réduit ainsi à :

- des numéros de règles,
- des constantes entières,
- des constantes alphanumériques.

Une optimisation supplémentaire a consisté dans l'assemblage préalable des constantes entières et des numéros de règles sous leur forme binaire.

A l'exécution du programme objet, l'analyse de ce texte sera menée suivant les règles qui régissent l'exploration d'un FORMAT, une table donnant pour chaque numéro de règle rencontré l'adresse du module qui réalisera l'action associée.

Une instruction FORMAT et la nouvelle forme de ses spécifications sont données ci-dessous à titre d'exemple :

```
FORMAT ( / , 'EXEMPLE' , 3 (F8.2 , 4X)).
```

Texte intermédiaire

0	19	9	E	X	E	M	P	L	E	22	3	13	12	7	8	15	2	3	4	13	7	16	1

Les nombre soulignés désignent les valeurs entières fournies par l'instruction de format (facteur de répétition, taille d'une zone, etc...).
Les autres nombres désignent des numéros de règles.

Tableau des numéros de règles

0	Début d'un FORMAT : initialisation de l'exploration
1	Fin d'un FORMAT : retour éventuel à la position de reprise
3	Saisie du 2ème paramètre d'une spécification F et conversion dans ce format
7	Marque de spécification X et saut de caractères
9	Début d'une constante alphanumérique
12	Début d'une spécification de groupe : rangement de la position de reprise
13	Saisie d'un facteur de répétition

15	Saisie du 1er paramètre d'une spécification F
16	Fin du traitement d'une spécification de groupe

19	Spécification / : passage à l'enregistrement suivant
22	Fin d'une constante alphanumérique

5.5. Diagnostic et traitement des erreurs à l'exécution :

A l'exécution d'un programme FORTRAN, deux types d'erreurs sont mis en évidence :

- des erreurs relatives aux opérations d'entrée-sortie,
- des anomalies internes d'exécution.

Le premier type d'erreur relève d'un défaut ou d'une mauvaise affectation de périphériques, ou encore d'une mauvaise organisation des données sur un support externe.

Ces erreurs facilement détectées sont signalées à l'utilisateur grâce à un message le renseignant sur leur nature exacte. Le contrôle est rendu au superviseur et une réexécution du programme peut être demandée.

Il n'en est pas de même pour les incidents du deuxième type qui sont :

- l'exécution d'une division entière par zéro,
- l'exécution d'une instruction faisant référence à une adresse mémoire inexistante sur l'installation.

L'absence de logique de déroutement déjà évoquée en 2.3.3.3. mettrait le système, après l'apparition d'un tel évènement, dans l'impossibilité de reprendre le contrôle d'une part,de diagnostiquer la nature de l'incident d'autre part.

Dans le cas de la division entière, un contrôle du diviseur préliminaire à toute opération de ce type a constitué une solution autorisant le diagnostic et la prévention de l'anomalie.

Une solution de prévention s'est par contre avérée difficilement applicable dans le cas de la deuxième anomalie, par suite de la diversité des situations qui peuvent l'engendrer : mauvais indicage d'un tableau, mauvais passage de paramètre, etc...

Remarquons néanmoins que, d'une part des deux inconvénients évoqués, l'arrêt du calculateur et l'absence de diagnostic, le moins contraignant est sans conteste le premier, une relance du calculateur étant en effet tout-à-fait admissible dans le contexte de son utilisation en mono-programmation et que, d'autre part il y a dans cette situation un diagnostic implicite puisque l'anomalie de la division entière par zéro est détectée et signalée.

Le problème de la localisation d'une erreur dans le texte FORTRAN peut être abordé maintenant.

5.6. L'aide à la mise au point d'un programme FORTRAN

Dans la définition d'un outil de mise au point, il était important de veiller à ne pas trop dégrader les performances du programme engendré par le compilateur. Aussi la solution retenue, qui met partiellement à contribution le pupitre de mise au point décrit en 7.4.2., répond-elle dans son principe à cette préoccupation.

Les facilités offertes sont les suivantes :

- localisation précise d'une erreur dans un programme grâce à l'indication du numéro de la dernière instruction FORTRAN exécutée ;
- suivi à la "trace" de l'exécution d'un programme.

Le détail de la réalisation est donné maintenant.

Lorsque l'utilisateur a recours à l'option mise au point de la compilation, un appel à un module SYSMAP est introduit dans le programme-objet avant toute séquence d'instructions-machines correspondant à une instruction FORTRAN exécutable.

Chaque appel ainsi réalisé dispose en paramètre du numéro de cette instruction FORTRAN dans le programme source.

Le module SYSMAP se charge alors simplement de ranger ce numéro dans un mot d'adresse fixe et connue. Ainsi la visualisation de ce mot permet-elle, après un arrêt intempestif, d'identifier l'instruction FORTRAN en cause.

La possibilité de procéder à l'aide du pupitre de mise au point à des arrêts sur adresse, avec visualisation des opérandes de l'instruction surveillée, est mise à profit pour permettre à l'utilisateur de suivre "à la trace" l'exécution de son programme FORTRAN.

Il suffit en effet de mettre sous surveillance le point d'entrée du module SYSMAP pour voir apparaître sur la rampe de visualisation du pupitre successivement les numéros de toutes les instructions FORTRAN exécutées.

L'activation et l'abandon de ce mode de travail ainsi que la progression dans l'exécution sont commandés à l'aide de commutateurs et boutons disposés sur ce même pupitre.

CHAPITRE VI

EXPOSE DE QUELQUES PRINCIPES DE REALISATION

Schématiquement, le traitement d'un problème sur ordinateur se décompose en trois étapes :

1. établissement d'un modèle par formalisation d'un problème concret ;
2. description d'un algorithme dans un langage qui peut ou non être destiné à l'interprétation par une machine ;
3. construction d'un programme exécutable destiné à mettre en oeuvre l'algorithme sur une machine donnée.

Quelques principes qui ont guidé la réalisation des deux dernières étapes sont décrits maintenant.

6.1. Analyse en "pseudo-algol" :

Le langage pseudo-algol est un langage de description d'algorithmes inspiré de l'ALGOL. Il a été utilisé dans cet exposé pour détailler les traitements.

Un des avantages de ce langage est d'être synthétique et de disposer de constructions "puissantes" du type :

- tant que .... faire ....
- pour i de 1 à n pas p faire ....
- si .... alors .... sinon ....

Dans cette étude, cette forme de description des algorithmes a été systématiquement utilisée, les programmes étant écrits en assembleur.

Compte-tenu des limitations du langage d'assemblage, le passage de l'algorithme au programme a introduit un certain nombre de distorsions.

Certaines précautions ont été prises pour limiter les effets de ces distorsions sur la validité du programme assembleur.

6.2. Traduction d'un algorithme en langage d'assembleur :

Les opérations décrites dans un langage d'assemblage sont les instructions d'une machine donnée. La nécessité de décrire les algorithmes à l'aide d'opérations très élémentaires fait qu'un tel langage se prête assez mal à une programmation claire.

Nous avons essayé de remédier à ce défaut par le respect d'une certaine discipline de programmation, dont un des premiers aspects est la traduction dans une forme standard des constructions du langage d'analyse.

6.2.1. Structure conditionnelle :

Soit la structure :

si C alors T1 sinon T2

où C désigne une condition et T1, T2 les traitements à réaliser suivant que la condition est remplie ou non.

Introduisons auparavant quelques notations :

- un branchement inconditionnel sera représenté symboliquement par :

BRI étiquette

- un branchement conditionné par la réalisation d'une condition C sera noté :

TEST = C , étiquette

où étiquette fournit l'adresse du saut ;

- le branchement sur une condition inverse sera représenté par :

TEST ≠ C , étiquette.

La traduction dans le langage symbolique d'assemblage donne alors pour la structure :

si C alors T1 sinon T2

évaluation de la condition C

TEST ≠ C , ETI1

traitement T1

BRI ETI2

ETI1

traitement T2

ETI2

Cette description appelle quelques commentaires :

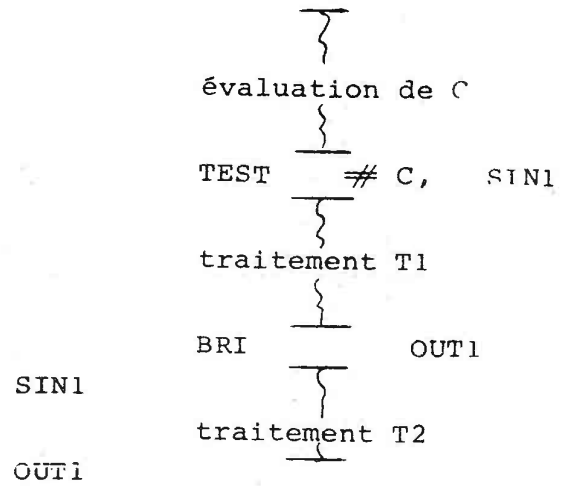
- le fait de tester la condition inverse permet de disposer le traitement de T1 avant celui de T2, c'est-à-dire de respecter l'ordre d'apparition des traitements dans la structure de l'analyse ;

- cette traduction a introduit deux étiquettes de saut. Chacune des constructions à traduire amènera ainsi la définition de telles étiquettes qu'il nous a semblé intéressant de particulariser, afin de mieux faire ressortir la structure qui les a introduites. Ceci a été réalisé par décomposition d'une étiquette en deux champs :

  . un préfixe qui précise la nature de l'étiquette sous forme d'un symbole de 3 caractères,
  . un suffixe constitué d'un nombre de 3 chiffres qui permet de différencier les étiquettes de même nature.

Ainsi, sur l'exemple proposé, à l'étiquette ETI1 qui repère le début du traitement associé au "sinon" sera préférée l'étiquette SIN1. De même, l'étiquette OUT1 sera substituée à ETI2 pour repérer le point de reprise commun à la sortie de la structure.

D'où la nouvelle forme suivante :



Les deux variantes suivantes ont été traitées d'après le même principe :

si C alors T1

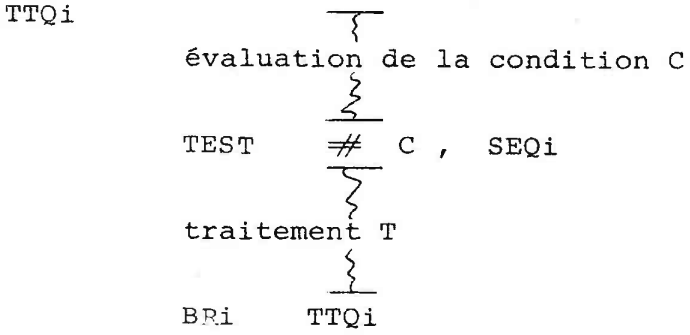
si C1 alors T1, si C2 alors T2 ... sinon T_n

6.2.2. *Structure itérative avec condition de fin* :

Examinons le cas de la structure :

tant que C faire T

La traduction proposée est la suivante :

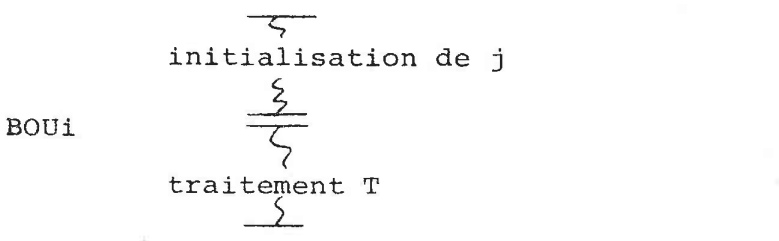


6.2.3. *Structure itérative avec compteur* :

On propose enfin la traduction d'une structure itérative :

pour j de i à f pas p faire T

Soit C la condition $j \leqslant f$; il vient :



Remarquons que cette structure itérative n'introduit qu'une seule étiquette BOUi qui correspond au point d'entrée dans le corps de boucle.

6.3. Gestion des registres à la compilation :

Pour obtenir rapidement un programme assembleur valide, il est important d'apporter un soin particulier à la gestion des registres.

Les registres constituent des mémoires à accès privilégié, car les informations qui s'y trouvent peuvent être mises en jeu directement dans les opérations élémentaires de la machine. Il apparaît donc là une source d'optimisations possibles du programme assembleur.

Une gestion optimale des registres est aisée lorsque le programme est simple, mais exige, au fur et à mesure que le programme gagne en complexité, un effort de plus en plus important de contrôles et de corrections, tant à l'écriture qu'à la mise au point.

En général, la complexité d'un programme ne croît pas proportionnellement au nombre d'instructions qu'il comporte, mais bien plus suivant le degré d'imbrications des modules qui le composent. En effet, chaque module ou sous-programme appelé dispose d'un contexte d'opérandes sensiblement, sinon totalement, différent de celui du module appelant.

Or, des exigences de modularité et de "sous-programisation poussée" destinées à minimiser l'encombrement mémoire introduisent dans cette réalisation un nombre important de tels modules, ce qui nous a conduit à adopter la stratégie d'utilisation des registres suivants.

Plusieurs classes de registres sont à distinguer :

- une première classe de registres qui recèlent des valeurs numériques couramment utilisées ; ainsi les registres 0, 1 et 2 contiennent respectivement les valeurs 0, 1 et -1. Ceci est intéressant compte-tenu des nombreuses initialisations et incrémentations en mémoire, chacune de ces opérations nécessitant la présence de ces valeurs dans un registre ;

- une deuxième classe de registres utilisés comme registres de liaison à l'appel d'un sous-programme (les registres 13, 14 et 15) ;

- une troisième classe sert enfin à mémoriser des variables "globales".
  Par exemple, lors de la procédure d'analyse, le registre 4 contient toujours le caractère lu dans le texte source ; de même, lors de la traduction des expressions arithmétiques, le registre 6 contient le descripteur de l'opérande en cours de traitement ;

- une dernière classe de registres utilisés comme registres de travail ou destinés à passer les paramètres lors de l'appel d'un sous-programme (les registres 3, 5, 7, 8, 9, 10, 11 et 12).

Intéressons-nous maintenant à la durée de vie de leur contenu respectif pour mettre en évidence les faibles risques d'erreur qu'introduit cette stratégie :

- les registres "constantes" ne sont pas modifiables ;
- les registres "variables globales" ne sont modifiés que par des procédures spécifiques qui leur sont attachées, par exemple le registre 4 ne sera modifié que par la procédure de lecture d'un caractère ;

- le contenu des registres de liaison n'est garanti qu'entre deux appels successifs à des sous-programmes qui sollicitent le même registre. Ces situations sont relativement rares car le calculateur dispose d'une instruction d'appel d'un sous-programme qui range l'adresse de retour dans le premier mot de celui-ci et ne nécessite donc pas de registre de liaison. Lorsque celui-ci doit être utilisé, son contenu sera sauvegardé à l'appel et restitué au retour du sous-programme, ceci étant réalisé automatiquement dans la procédure d'analyse syntaxique par empilement et dépilement systématique du contexte, du fait de la récursivité ;
- en ce qui concerne les registres de travail, le principe retenu est de ne tenir leur contenu pour significatif qu'entre deux appels successifs à des sous-programmes, ce qui diminue largement les risques d'erreur amenés par l'imbrication des appels.

Cette restriction a eu pour effet de disposer dans certains cas les variables de travail en mémoire plutôt que dans un registre.

La désignation de la variable par le symbole déjà utilisé à l'analyse nous a permis alors de mettre à profit son introduction pour améliorer la clarté du programme.

De manière générale, nous avons estimé qu'il était souhaitable que le programme lui-même reflète la description permettant de comprendre sa structure et son fonctionnement.

Un algorithme en pseudo-algol et sa traduction en langage d'assemblage sont donnés ci-dessous à titre d'exemple.

```
procédure  analysexternal  début
             empl ; addet := 0 ; sauctx ;
             psg ::= psg+1 ; pos := pos-1 ; C := ','
             ttq  C = ','  faire début  lire (C) ;
                                anal104 ; acto48
                                fin ;
             typins := -1 ; retcdt ; dep2
                           fin
```

```
* * ANALYSE EXTERNAL * * 48 *
*
*
ANA48   BSP      14,EMP1
        RGE      Ø,ADDET
        RGE      Ø,PSG
        BSP      14,SAUCTX
        INCP     1,PSG
        INCP     2,POS
        CHE      4,=","
TTQ11   CPE   #  4,=","
        BRI      SEQ11
        BSP      14,LIREC
*
* ANAL  IDENT **
        BSP      15,ANA1Ø4
        BSP      14,ACTØ48
        BRI      TTQ11
SEQ11   RGE      2,TYPINS
        BSP      14,RETCDT
        BSP      14,DEP2
*
*
```

6.4. Réalisation des fonctions d'accès aux données :

Dans un langage d'assemblage, les structures de données sont assez rudimentaires ; or un travail de compilation conduit à manipuler des structures d'informations assez complexes : tables, piles, vecteurs de renseignements, etc... Leur implantation demandera donc de créer de toutes pièces des fonctions d'accès dont la réalisation exige en général le déroulement de plusieurs instructions en assembleur. Pour des raisons évidentes de gain de place en mémoire, nous avons constitué, pour chaque fonction ainsi introduite, un sous-programme spécifique pour lequel un interface particulier (passage de paramètres, modalités d'appel, etc...) a été défini.

Cette façon de procéder présente encore d'autres intérêts :

- une certaine modularité : une modification dans la fonction d'accès ou dans la structure d'informations n'exigera que la réécriture du sous-programme et non des parties du programme qui utilisent l'accès à cette structure (à condition que l'interface soit le même) ;

- pour peu que la procédure d'accès dispose de tests de validité de l'utilisation, elle apportera une aide à la mise au point facilitée par la détection précoce des erreurs.

L'exemple ci-dessous présente les modules de gestion et d'accès à une pile.

h et hmax désignent respectivement la hauteur courante et maximale de la pile et "erreur" une procédure de traitement de l'erreur :

```
procédure empiler (x) début  si h = hmax alors erreur (1)
                             sinon début pile (h) ::= x ;
                                         h := h+1
                                   fin
                       fin

procédure dépiler (x) début  si h = 0  alors erreur (2)
                             sinon début h ::= h-1 ;
                                         x ::= pile (h)
                                   fin
                       fin

procédure accéder (n,x) début  si n ≥ hmax alors erreur (3)
                               si n ≤ 0 alors erreur (4)
                               sinon x ::= pile (n)
                         fin
```

Les erreurs 1 et 2 correspondent respectivement au débordement de la pile vers le haut et vers le bas.

La procédure "accéder" permet de lire un élément de la pile après avoir vérifié que le niveau indiqué y est présent (erreur n° 3 et 4).

Lors de la mise au point de notre système, la procédure erreur visualisait l'adresse d'appel du module, ainsi que l'élément le plus significatif dans la situation d'anomalie, l'expression des contenus de registres venant compléter ces informations.

CHAPITRE VII

ASPECTS PROPRES A LA MINI-INFORMATIQUE

Profitant de l'expérience acquise lors de cette réalisation, il est tentant de vouloir dégager tous les aspects propres à la mini-informatique et de donner quelques considérations d'ordre général sur le problème de l'implantation d'un logiciel important sur mini-calculateur. Cette tâche se heurte pourtant à deux difficultés majeures :

- le flou dans la définition de la mini-informatique,
- les particularités inévitables d'un calculateur ou d'une configuration donnée.

S'il n'est donc pas facile de donner d'une manière exhaustive tous les aspects de la mini-informatique, il est possible néanmoins d'en énumérer certains, dont on sait avec certitude qu'ils contribuent à faire d'une configuration donnée un mini-système.

Certains des aspects déjà évoqués dans cet exposé seront reprécisés en rappelant, moyennant quelles modifications importantes dans sa conception, un compilateur FORTRAN peut s'en accomoder.

Quelques particularités qui ont trait à la mise au point d'un logiciel sur un tel système sont précisées ensuite.

7.1. Faible capacité mémoire :

Des contraintes d'espace mémoire se retrouvent en général sur tous les mini-calculateurs, mais assez souvent à un degré moindre. Seuls 8 k mots étaient ici alloués au compilateur, l'implantation a été néanmoins rendue possible grâce au recouvrement en mémoire de certaines parties du programme.

Pour cela, il a fallu définir un arbre de recouvrement et introduire un langage intermédiaire tel que le temps de compilation ne soit pas pénalisé par des chargements trop fréquents des segments en mémoire, la contre-partie de la modicité du prix d'une unité de disques souples étant un accès assez peu rapide à l'information.

En règle générale et pour des raisons d'encombrement, la préférence a été donnée aux solutions peu couteuses en mémoire et ceci au détriment parfois du temps de traitement. D'autre part, à certaines étapes de la compilation, le temps d'unité centrale reste négligeable devant les temps d'entrées sorties qui, compte-tenu des périphériques utilisés, constituent les 9/10 du temps de passage.

7.2. Petit format des mots-machine :

Les mini-calculateurs possèdent souvent des mémoires à mots de 16 bits (TELEMECANIQUE T1600 et Solar 16, CII Mitra 15, DATA GENERAL Nova, DIGITAL EQUIPEMENT PDP 11) et quelquefois de 24 bits (ICL et CROUZET Campanule).

Les instructions-machine sont rarement de longueur variable comme pour le PDP 11, mais le plus souvent de même longueur que le mot mémoire.

Des techniques d'adressage particulières sont introduites alors pour palier à l'impossibilité d'adresser en 8 ou 12 bits un mot quelconque de la mémoire disponible.

Dans le cas du CAMPANULE, ces techniques mettent en jeu des registres page opérandes et page instructions et nous avons déjà évoqué les difficultés à assurer leur gestion à l'exécution du programme objet.

La solution de la segmentation décrite à cette occasion a permis de résoudre ce problème dans n'importe quelle configuration mémoire.

7.3. L'usage du ruban perforé :

Le faible coût du ruban perforé à l'équipement et à l'utilisation lui donne sur la carte un avantage certain.

Alors que l'utilisation de la carte dans un système exige au minimum un lecteur de cartes et une perforatrice, ce qui représente un certain investissement, le ruban perforé peut être mis en oeuvre sur des périphériques moins couteux, tels que lecteurs et perforateurs de ruban, ou même sur un seul téléimprimeur.

Les modifications à apporter dans un fichier ruban sont assez difficiles à réaliser et représentent l'inconvénient majeur de ce type de support.

Un mode conversationnel à la compilation et des tâches de services disponibles permettent d'y remédier et rendent, à notre sens, l'utilisation du ruban perforé viable dans ce type de système.

7.4. Les outils de mise au point :

Les techniques de mise au point d'un programme assembleur sur mini-calculateur diffèrent sensiblement de celles utilisées sur les systèmes plus importants.

Elles se traduisent en général par un haut degré d'interaction entre le programme qui s'exécute et l'utilisateur qui en fait la mise au point.

Deux grandes classes de techniques sont à distinguer :

- celles qui font appel à un logiciel d'aide à la mise au point,
- celles qui utilisent un "pupitre" de mise au point.

7.4.1. Les programmes d'aide à la mise au point :

Les logiciels d'aide à la mise au point
(par exemple : ODT 11 sur PDP 11 de Digital Equipement,
PROGAID sur T 2000 de TELEMECANIQUE)
reposent sur le principe suivant.

Le programme étant chargé en mémoire, son exécution est effectuée sous le contrôle d'un superviseur.

L'utilisateur peut alors, par le biais de commandes spécialisées, demander la réalisation de certaines fonctions, les plus courantes étant :

- la visualisation ou le chargement de mots mémoire ou registres,
- l'exécution instruction par instruction d'une séquence,
- la réalisation d'une "trace",
- l'introduction de points d'arrêts d'exécution.

Certains systèmes, comme PROGAID, permettent même l'usage des notations symboliques du programme, pour peu que l'on dispose encore de la table des symboles constituée à l'assemblage.

Conclusion :

Ces systèmes sont bien adaptés au problème de la mise au point mais nécessitent un surcroît d'espace mémoire à l'exécution.

7.4.2. Les pupitres de mise au point :

Certains mini-ordinateurs sont équipés (quelquefois en option) d'un pupitre disposé en face avant du calculateur, dont le but est de permettre à l'utilisateur, par un jeu de clés et de boutons sélecteurs, d'exécuter son programme dans un mode de fonctionnement apte à lui fournir les informations nécessaires à la détection et à la correction des erreurs.

La description du pupitre du CAMPANULE (figure 8.1) est donnée à titre d'exemple :

S1 : commutateur associé à l'affichage V1 qui permet à l'exécution d'une instruction de sélectionner l'opérande à visualiser sur V1 ;

S2 : sélecteur qui précise le mode de fonctionnement ;

I3 : cet inverseur, en position clavier, inhibe le programme en mémoire et valide l'instruction affichée sur le clavier C1 ;

C2 : clavier qui permet d'afficher, après prise en compte par le bouton B2, l'adresse binaire surveillée et visualisée par la rampe V2 ;

B1 : bouton qui commande l'exécution dans le mode de fonctionnement choisi.

Les fonctions réalisées sont les suivantes :

- arrêt du programme sur une instruction donnée,
- déroulement du programme instruction par instruction,
- visualisation des opérandes et du résultat d'une instruction,
- exécution d'instructions supplémentaires au clavier,
- lecture et écriture en mémoire du contenu d'un mot.

L'éventail des fonctions disponibles est plus réduit que celui proposé par un logiciel de mise au point, mais leur réalisation étant entièrement assurée par une logique micro programmée, donc n'entamant en aucune façon la ressource mémoire centrale, rendait cette solution plus intéressante pour l'étude que nous avons menée.

La description du pupitre du CAMPANULE (figure 8.1) est donnée à titre d'exemple :

S1 : commutateur associé à l'affichage V1 qui permet à l'exécution d'une instruction de sélectionner l'opérande à visualiser sur V1 ;

S2 : sélecteur qui précise le mode de fonctionnement ;

I3 : cet inverseur, en position clavier, inhibe le programme en mémoire et valide l'instruction affichée sur le clavier C1 ;

C2 : clavier qui permet d'afficher, après prise en compte par le bouton B2, l'adresse binaire surveillée et visualisée par la rampe V2 ;

B1 : bouton qui commande l'exécution dans le mode de fonctionnement choisi.

Les fonctions réalisées sont les suivantes :

- arrêt du programme sur une instruction donnée,
- déroulement du programme instruction par instruction,
- visualisation des opérandes et du résultat d'une instruction,
- exécution d'instructions supplémentaires au clavier,
- lecture et écriture en mémoire du contenu d'un mot.

L'éventail des fonctions disponibles est plus réduit que celui proposé par un logiciel de mise au point, mais leur réalisation étant entièrement assurée par une logique microprogrammée, donc n'entamant en aucune façon la ressource mémoire centrale, rendait cette solution plus intéressante pour l'étude que nous avons menée.

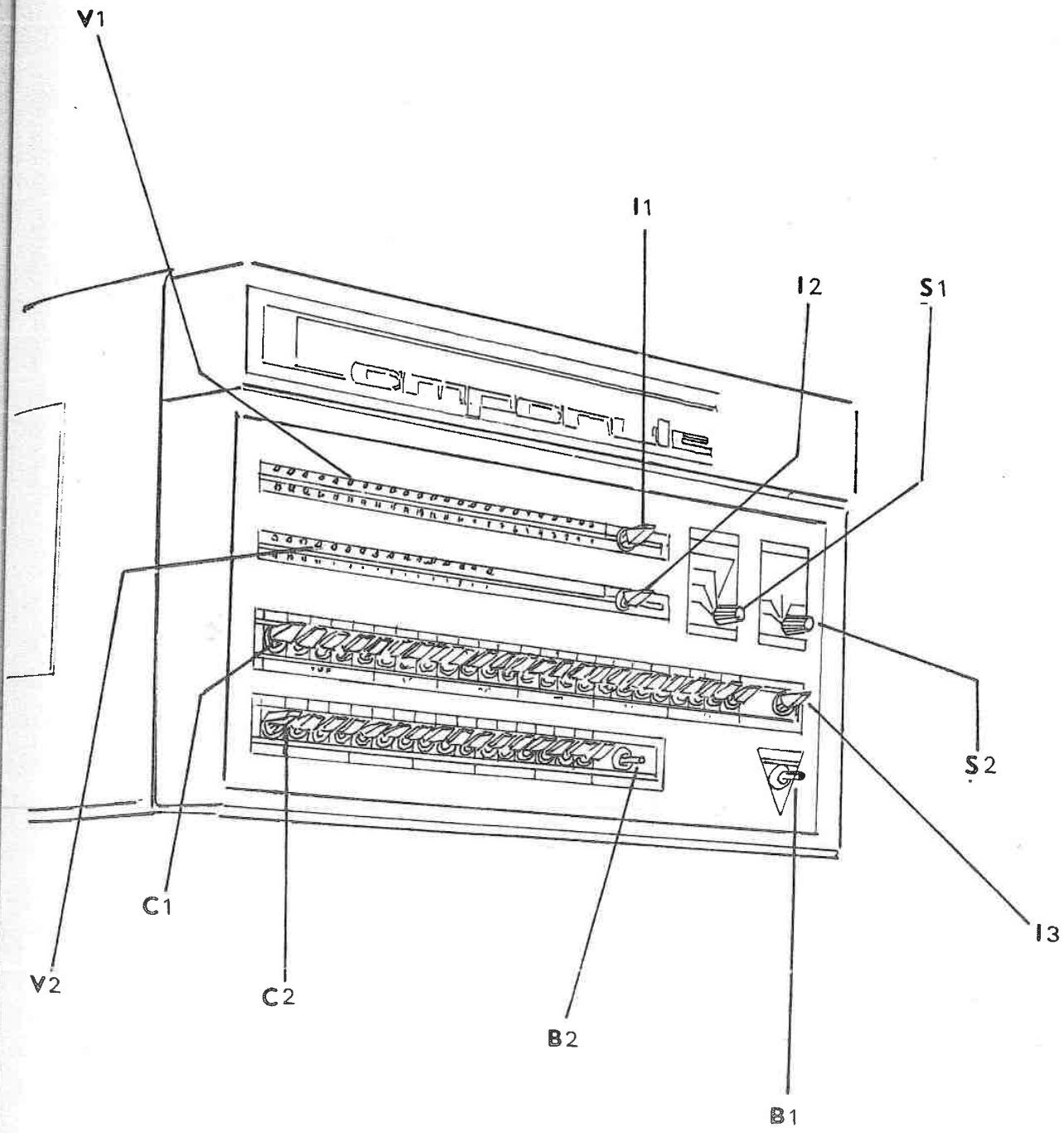


Figure 7.1.

Pupitre du CAMPANULE

CONCLUSION

Le logiciel ainsi réalisé répond largement, à notre sens, aux objectifs fixés dans cette étude. La programmation en FORTRAN à partir d'une configuration réduite s'avère finalement aisée.

Un travail de compilation, grâce à une architecture appropriée du système, profite dans une large mesure de la présence d'une mémoire de masse sans être pour autant pénalisé par les caractéristiques somme toute modestes d'un disque souple.

L'existence des calculs complexes et réels en double précision devrait être appréciée dans le traitement des problèmes scientifiques.

Néanmoins, l'écriture d'un éditeur de liens et l'introduction du recouvrement à l'exécution d'un programme utilisateur qui n'ont pu être réalisées ici devraient être entreprises ultérieurement pour tirer un meilleur parti de la mémoire disponible.

Les facilités offertes par le mode conversationnel de la compilation et l'assistance à la création et à la modification de programme devraient contribuer à faire du système un bon outil d'enseignement.

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

[1] M. DREYFUS : FORTRAN IV - Collection du CIRO, DUNOD (1969).

[2] J.-P. LAMOITIER : Le langage FORTRAN IV - Bibliothèque Technique Philips, DUNOD (1973).

[3] C. PAIR : Cours de compilation - Ecole d'Eté d'Informatique, NEUCHATEL (1972).

[4] CROCUS : Systèmes d'exploitation des ordinateurs - DUNOD UNIVERSITE (1974) page 73.

[5] F.R.A. HOPGOOD : Techniques de compilation, DUNOD (1970).

[6] C. BOXENBAUM : Moyens et problèmes de la compilation sur petites machines -
Revue Française d'Informatique et de Recherche Opérationnelle (revue de l'AFCET) (Mai 1970).

[7] D.-E. KNUTH : The Art of Computer Programming -
Volume 3, Sorting and Searching, pages 506 à 548.

[8] D.-W. BARROW : Techniques récursives en programmation - DUNOD (1970) pages 25 à 33.

[9] L. BOLLIET : Les systèmes conversationnels -
Monographie d'informatique, AFCET
DUNOD (1968).

[10] P. NAUR : Revised report on the algorithmic Language
Algol 60 -
CACM 6 (1963) pages 1 à 17.

[11] J.-J. HORNING : LR Grammars and analysers -
Lecture notes in computer Science 21,
SPRINGER VERLAG (1974).

[12] D.-E. KNUTH : Top down syntax analysis -
ACTA INFORMATICA 1, pages 79-110
SPRINGER VERLAG (1971).

A N N E X E S

TABLE DES MATIERES.

GRAMMAIRE DU LANGAGE COMPILE EN NOTATION BNF

<PROG> ::= <PROGPRINC> I <PROGPRINC> <LISTSSP>

<LISTSSP> ::= <SSP> I <SSP> <LISTSSP>

<PROGPRINC> ::= <LISTINST> <END>

<SSP> ::= <ENTETE> <LISTINST> <END>

<LISTINST> ::= <INST> I <INST> <LISTINST>

<END> ::= 'END' EOL

<ENTETE> ::= <DCLSUB> I <DCLFUNC> I <BLOCKDATA>

<DCLSUB> ::= <SUBROUTINE> <ID> I <SUBROUTINE> <IDPARAM>

<DCLFUNC> ::= <FUNCTION> <IDPARAM> I <DECLAR> <FUNCTION>
<IDPARAM>

<BLOCKDATA> ::= 'BLOCKDATA' EOL

<SUBROUTINE> ::= 'SUBROUTINE'

<FUNCTION> ::= 'FUNCTION'

<ID> ::= <LETTRE> I <LETTRE> <LISTLETCHF>

<IDPARAM> ::= <ID><PARENTOUV><LISTID><PARENTFER>

<INST> ::= <ETI> <INSTEXEC> I <INSTEXEC> I <INSINEXEC>

<INSTEXEC> ::= <INSTAFFC> I <INSTCTR> I <INSTES>

<INSTNEXEC> ::= <INSTORG> I <INSTDATA > I <INSTFORMAT> I <FONCTFORM>

<INSTAFFC> ::= <AFFC> I <INSTASSIGN>

<INSTCTR> ::= <INSTGOTO> I <IFARITH> I <IFLOG> I <INSTCALL>
I <INSTRETURN> I <INSTCONTINUE> I <INSTSTOP>
I <INSTDO> I <INSTPAUSE>

<INSTES> ::= <INSTREAD> I <INSTWRITE> I <ESAUX>

<INSTORG> ::= <INSTDIMENSION> I <INSTCOMMON> I <INSTEXTERNAL>
I <INSTEQUIVALENCE> I <INSTTYP>

<INSTDATA> ::= <DATA> <LISTVARCST><EOL>

<EOL> ::= EOL

<LISTVARCST> ::= <VARCST> I <VARCST> <VIRGULE>
<LISTVARCST>

<VARCST> ::= <LISTVAR><SLASH><LISTCST><SLASH>

```
<INSTFORMAT> ::= <FORMAT> <PARENTOUV> <LISTDESC>
                      <PARENTFER>

<LISTDESC> ::= <DESC> I <DESC> <LISTDESC>

<DESC> ::= <FACTREP> <SPECIF> I <SPECIF>

<SPECIF> ::= <SPECIFELEM> I <PARENTOUV> <LISTDESC>
                            <PARENTFER>

<LISTSEPAR> ::= <SEPAR> I <SEPAR> <LISTSEPAR>

<SEPAR> ::= ',' I '/'

<FACTREP> ::= <CONSTENT>

<SPECIFELEM> ::= <F> <CONSTENT> <POINT> <CONSTENT> I
                 <E> <CONSTENT> <POINT> <CONSTENT> I
                 <G> <CONSTENT> <POINT> <CONSTENT> I
                 <D> <CONSTENT> <POINT> <CONSTENT> I
                 <I> <CONSTENT> I  <A> <CONSTENT>  I
                 <L> <CONSTENT> I <H> <LISTALPHA>  I
                 <X> I <COTE> <LISTALPHA> <COTE>

<A> ::= 'A'

<F> ::= 'F'

<E> ::= 'E'

<G> ::= 'G'

<D> ::= 'D'

<I> ::= 'I'

<L> ::= 'L'

<H> ::= 'H'

<X> ::= 'X'

<COTE> ::= '''

<FORMAT> ::= 'FORMAT'

<SLASH> ::= '/'

<FONCFORM> ::= <ID> <PARENTOUV> <LISTID> <PARENTFER>
                <EGAL> <EXPGEN>

<EXPGEN> ::= <EXPARITH> I <EXPLOOL> I < CONSTALPHA>

<EXPARITH> ::= <T> <K> I <T>
```

```
<K> ::= <PLUS> <T> I <MOINS> <T> I <PLUS> <T> <K>
        <MOINS> <T> <K>

<T> ::= <FE> <LEX> I <FE>

<LEX> ::= <STAR> <LEX> I <SLASH> <LEX> I <STAR> <FE> I
          <SLASH> <FE>

<FE> ::= <PLUS> <OP> I <MOINS> <OP> I <OP>

<OP> ::= <PARENTOUV> <EXPAFITH> <PARENTFER> I <P>
         <P> <STAR> <STAR> <F>

<P> ::= <ID> I <ID> <PARENTOUV> <LISTEXPGEN>
        <PARENTFER> I <CONSTNUM>

<CONSTNUM> ::= <CONSTENT> I <CONSTREEL> I <CONSTCPLX>

<LISTEXPGEN> ::= <EXPGEN> I <EXPGEN> <LISTEXPGEN>

<CONSTENT> ::= <CHIFFRE> I <CHIFFRE> <CONSTENT>

<CONSTREEL> ::= <MANTISSE> I <MANTISSE> <E> <EXPOSANT>

<MANTISSE> ::= <CONSTENT> <POINT> I <POINT> <CONSTENT> I
               <CONSTENT> <POINT> <CONSTENT>

<EXPOSANT> ::= <PLUS> <CONSTENT> I <MOINS> <CONSTENT> I
               <CONSTENT>

<CONSTCPLX> ::= <PARENTOUV><CONSTRELSIG><VIRGULE>
                <CONSTRELSIG><PARENTFER>

<CONSTRELSIG> ::= <PLUS> <CONSTREEL> I <MOINS> <CONSTREEL>
                  I <CONSTREEL>
<CONSTALPHA> ::= <CONSTENT> <H> <LISTALPHA> I
                 <COTE> <LISTALPHA> <COTE>

<LISTALPHA> ::= <ALPHA> I <ALPHA> <LISTALPHA>

<ALPHA> ::= <LETTRE> I <CHIFFRE> I <CARACSPEC>

<EB> ::= <TE> I <TE> <POINT> <OR> <POINT> <EE>

<TE> ::= <FE> I <FE> <POINT> <AND> <POINT> <TE>

<FE> ::= <OPE> I <POINT> <NOT> <POINT> <OPE>

<OPE> ::= <PE> I <PARENTOUV> <EE> <PARENTFER>

<PE> ::= <CONSTBOOL> I <ER> I <ID> I <ID> <PARENTOUV>
                    <LISTEXPGEN> <PARENTFER>
```

```
<CONSTBOOL> ::= <POINT> <TRUE> <POINT> I
                <POINT> <FALSE> <POINT>

<EF> ::= <EXPGEN> <POINT> <OPEF> <POINT> <EXPGEN>

<OPEF> ::= <LT> I <LE> I <EQ> I <NE> I <GE> I <GT>

<LT> ::= 'LT'

<LE> ::= 'LE'

<EQ> ::= 'EQ'

<NE> ::= 'NE'

<GE> ::= 'GE'

<<GT> ::= 'GT'

<OR> ::= 'OR'

<AND> ::= 'AND

<NOT>::= 'NOT'

<TRUE> ::= 'TRUE'

<FALSE> ::= 'FALSE'

<AFFC> ::= <VAR> <EGAL> <EXPGEN>

<INSTASSIGN> ::= <ASSIGN> <ETI> <TO> <VAR> <FET>

<INSTGOTO> ::= <GOTOINCOND> I <GOTOCALCULE> I
                <GOTOIMPOSE>

<GOTOINCOND> ::= <GOTO> <ETI> <FET>

<GOTOCALCULE> ::= <GOTO> <PARENTOUV> <LISTETI> <PARENTFER>
                        <VIRGULE> <EXPARITH> <FET>

<GOTOIMPOSE> ::= <GOTO> <VAR> <VIRGULE> <PARENTOUV>
                    <LISTETI><PARENTFER> <FET>

<LISTETI> ::= <ETI> I <ETI> <VIRGULE> <LISTETI>

<ETI> ::= <CONSTENT>

<ASSIGN> ::= 'ASSIGN'

<TO> ::= 'TO'

<GOTO> ::= 'GOTO'
```

```
<IFARITH> ::= <IF> <PARENTOUV> <EXPARITH> <PARENTFER>
                    <ETI> <VIRGULE> <ETI> <VIRGULE> <ETI>

<IFLOG> ::= <IF> <PARENTOUV> <EE> <PARENTFER> <INSTEXEC> <RET>

<INSTCALL> ::= <CALL> <ID> <PARENTOUV> <LISTEXPGEN>
                                <PARENTFER> I <CALL> <ID>

<IF> ::= 'IF'

<CALL> ::= 'CALL'

<INSTRETURN> ::= <RETURN> <RET>

<INSTCONTINUE> ::= <CONTINUE> <RET>

<INSTSTOP> ::= <STOP> <RET>

<INSTPAUSE> ::= <PAUSE> <RET>

<RETURN> ::= 'RETURN'

<CONTINUE> ::= 'CONTINUE'

<STOP> ::= 'STOP'

<PAUSE> ::= 'PAUSE'

<INSTDO> ::= <DO> <ETI> <ID> <EGAL> <LISTVAL> <RET>

<LISTVAL> ::= <VAL> <VIRGULE> <VAL> <VIRGULE> <VAL> I
              <VAL> <VIRGULE> <VAL>

<VAL> ::= <EXPARITH>

<DO> ::='DO'

<INSTREAD> ::= <READ> <PARENTOUV> <UNITE> <VIRGULE> <ETI>
                        <PARENTFER> <LISTVARES> <RET>

<INSTWRITE> ::= <WRITE> <PARENTOUV> <UNITE> <VIRGULE>
                            <PARENTFER> <LISTVARES> <RET> I
                    <WRITE> <PARENTOUV> <UNITE> <VIRGULE> <ETI>
                            <PARENTFER> <RET>

<LISTVARES> ::= <VARES> I <VARES> <VIRGULE> <LISTVARES>

<VARES> ::=  <VAR> I <DOIMPLICITE>

<VAR> ::= <ID> I <ID> <PARENTOUV> <LISTEXPARITH> <PARENTFER>

<DOIMPLICITE> ::= <PARENTOUV> <LISTDOIMP> <VIRGULE> <ID>
                        <EGAL> <LISTVAL> <PARENTFER>

<LISTDOIMP> ::= <ELEMDOIMP> I <ELEMDOIMP> <VIRGULE>
                                        <LISTDOIMP>
```

<ELEMDOIMP> ::= <VAR> I <DOIMPLICITE>

<LISTEXPARITH> ::= <EXPARITH> I <EXPARITH> <VIRGULE>
<LISTEXPARITH>

<UNITE> ::= <CONSTENT>

<READ> ::= 'READ'

<WRITE> ::= 'WRITE'

<ESAUX> ::= <INSTREWIND> I <INSTBACKSPACE> I <INSTENDFILE>

<INSTREWIND> ::= <REWIND> <UNITE> <RET>

<INSTBACKSPACE> ::= <BACKSPACE> <UNITE> <RET>

<INSTENDFILE> ::= <ENDFILE> <UNITE> <RET>

<REWIND> ::= 'REWIND'

<BACKSPACE> ::= 'BACKSPACE'

<ENDFILE> ::= 'ENDFILE'

<INSTDIMENSION> ::= <DIMENSION> <LISTDCL> <RET>

<LISTDCL> ::= <DCL> I <DCL> <VIRGULE> <LISTDCL>

<DCL> ::= <ID> I <ID> <PARENTOUV> <LISTEOFN> <PARENTFER>

<LISTEOFN> ::= <EOFN> I <EOFN> <VIRGULE> <LISTEOFN>

<EOFN> ::= <ID> I <CONSTENT>

<INSTCOMMON> ::= <COMMON> <LISTCOM> <RET>

<LISTCOM> ::= <COM> I <COM> <VIRGULE> <LISTCOM>

<COM> ::= <LISTVCOM> I <SLASH> <SLASH> <LISTVCOM> I
<SLASH> <ID> <SLASH> <LISTVCOM>

<LISTVCOM> ::= <VCOM> I <VCOM> <VIRGULE> <LISTVCOM>

<VCOM> ::= <ID> I <ID> <PARENTOUV> <LISTCONSTENT> <PARENTFER>

<INSTEQUIVALENCE> ::= <EQUIVALENCE> <LISTEQU> <RET>

<LISTEQU> ::= <EQU> I <EQU> <VIRGULE> <LISTEQU>

<EQU> ::= <PARENTOUV> <LISTVEQU> <PARENTFER>

```
<LISTVEQU> ::= <VEQU> I <VEQU> <VIRGULE> <LISVEQU>

<VEQU> ::= <ID> I <ID> <PARENTOUV> <LISTCONSTENT> <PARENTFER>

<INSTEXTERNAL> ::= <EXTERNAL> <LISTID> <RET>

<DIMENSION> ::= 'DIMENSION'

<COMMON> ::= 'COMMON'

<EQUIVALENCE> ::= 'EQUIVALENCE'

<EXTERNAL> ::= 'EXTERNAL'

<INSTTYP> ::=  <DECLAR> <LISTDCL> <RET>

<DECLAR> ::= <INTEGER> I <REAL> I <COMPLEX> I <LOGICAL> I
             <DOUBLEPRECISION>

<LISTDCL> ::= <DCL> I <DCL> <VIRGULE> <LISTDCL>

<DCL> ::= <ID> I <ID> <PARENTOUV> <LISTEDCL> <PARENTFER>

<LISTEDCL> ::= <EDCL> I <EDCL> <VIRGULE> <LISTEDCL>

<EDCL> ::= <ID> I <CONSTENT>

<INTEGER> ::= 'INTEGER'

<REAL> ::= 'REAL'

<LOGICAL> ::= 'LOGICAL'

<COMPLEX> ::= 'COMPLEX'

<DOUBLEPRECISION> ::= 'DOUBLEPRECISION'

<DATA> ::= 'DATA'

<LISTLETCHF> ::= <LETCHF> I <LETCHF> <LISTLETCHF>

<LETCHF> ::= <LETTRE> I <CHIFFRE>

<LETTRE> ::= 'A'I'B'I'C'I'D'I'E'I'F'I'G'I'H'I'I'I'J'I'
             'K'I'L'I'M'I'N'I'O'I'P'I'Q'I'R'I'S'I'T'I
             'U'I'V'I'W'I'X'I'Y'I'Z'

<CHIFFRE> ::= '0'I'1'I'2'I'3'I'4'I'5'I'6'I'7'I'8'I'9'

<CARACSPEC> ::= '+'I'-'I'*'I'/'I'.'I','I'$'I'='I' 'I'('I')'
                ''''
```

<PARENTOUV> ::= '('

<PARENTFER> ::= ')'

<LISTID> ::= <ID> I <ID> <VIRGULE> <LISTID>

<ETI> ::= <CONSTENT>

<POINT> ::= '.'

<PLUS> ::= '+'

<MOINS> ::= '-'

<STAR> ::= '*'

<VIRGULE> ::= ','

<EGAL> ::= '='

CALCULATEUR
CAMPANULE

CROUZET

NOTICE D'UTILISATION

DU

FORTRAN CAMPANULE

SOUS LE SYSTEME SYCAMP

S O M M A I R E

1. PRESENTATION.

Le noyau FORTRAN du système SYCAMP sur calculateur CAMPANULE permet :

- l'analyse conversationnelle d'un programme FORTRAN,
- la génération d'un programme binaire,
- l'exécution d'un tel programme.

Chaque tâche (compilation et exécution) est réalisée par un processeur indépendant, activé par SYCAMP sur une commande de l'utilisateur.

2. INTRODUCTION.

2.1. Superviseur SYCAMP :

Le superviseur SYCAMP réalise l'enchaînement des travaux en mono-programmation par l'intermédiaire de commandes décrivant les étapes d'un travail.

Le système repose sur l'existence d'une bibliothèque de programmes sur disque, le noyau résidant en permanence en mémoire, n'assurant que les fonctions essentielles.

Les différents programmes systèmes de FORTRAN ne sont pas résidents ; leur chargement en mémoire et leur activation se font sur une demande de l'utilisateur.

2.2. Etapes d'un travail FORTRAN :

Les étapes d'un travail FORTRAN sont au nombre de deux et dans l'ordre chronologique :

- la compilation du programme-source FORTRAN,
- l'exécution du programme binaire.

2.2.1. La compilation

Cette étape réalise l'analyse conversationnelle du texte FORTRAN et la génération d'un programme binaire exécutable.

2.2.2. L'exécution

Cette étape réalise le chargement en mémoire du programme binaire exécutable, résultat de la compilation et son exécution.

2.3. Environnement d'une étape FORTRAN - Unités symboliques :

La réalisation d'une tâche exige, en général, un certain nombre d'échanges avec l'extérieur. Chacune de ces opérations d'E/S reste à la charge du système et met en jeu un organe périphérique parmi ceux dont dispose une configuration de matériel donnée. Pour assurer une relative indépendance du système par rapport aux périphériques et une souplesse d'utilisation, on a été amené à introduire la notion d'unité symbolique (US)
d'unité fonctionnelle (UF).

Unités symboliques :

Les unités symboliques n'ont d'existence qu'au niveau des programmes. L'environnement d'une étape FORTRAN est un ensemble d'unités symboliques.

Le lien avec un périphérique donné est à la charge de l'utilisateur et se fait à travers une commande d'affectation d'une unité fonctionnelle à une unité symbolique (voir notice descriptive du superviseur SYCAMP).

Unités fonctionnelles :

Les unités fonctionnelles (UF) désignent les fonctions d'un périphérique (voir paragraphes 4.2 et 4.3 du synoptique).

Remarque : à défaut d'une affectation explicite, le supervisuer MUCAMP assure une affectation implicite.

Exemple : l'unité fonctionnelle TC (Télétype Clavier) sera affectée par défaut à l'unité symbolique CDE (CommanDE).

Les opérations d'E/S sont réalisées par l'intermédiaire du moniteur d'E/S MODES. Elles portent sur des unités symboliques.

2.4. Enchaînement des étapes FORTRAN :

Les étapes d'un travail FORTRAN sont activées par l'utilisateur sous le contrôle du superviseur.

Les résultats intermédiaires sont stockés sur disque dans des fichiers gérés par l'utilisateur.

3. DESCRIPTION.

3.1. Les étapes d'un travail FORTRAN :

Le superviseur SYCAMP active toute étape FORTRAN sous contrôle de l'utilisateur.

Le moniteur d'E/S MODES réalise l'interface utilisateur-programme système à travers les unités symboliques qui constituent l'environnement de l'étape FORTRAN.

Il appartient à l'utilisateur de réaliser avant toute étape les affectations US-UF nécessaires.

Son travail terminé, toute étape FORTRAN rend le contrôle au superviseur.

3.1.1. L'étape de compilation du texte FORTRAN

3.1.1.1. Rôle :

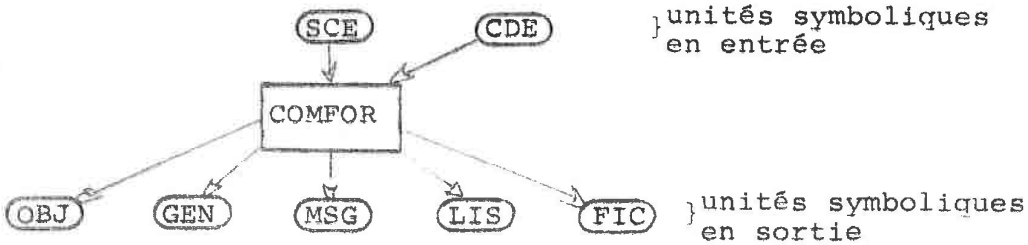
Cette étape réalise l'analyse conversationnelle du programme FORTRAN et la génération d'un programme binaire exécutable.

3.1.1.2. Activation :

L'activation est réalisée par le superviseur sur la commande EXE COMFOR (COMpilation FORtran).

3.1.1.3. Environnement :

L'environnement de cette étape peut être représenté de la manière suivante :



Les unités symboliques sont désignées par des mots-clés reconnus par le superviseur DISCAMP.

Leur signification pour cette étape est résumée ci-dessous :

- SCE : unité de saisie d'un texte FORTRAN existant.
- CDE : unité de saisie des directives de compilation et des créations ou modifications du texte FORTRAN.

. OBJ : fichier disque de travail.
. FIC : fichier disque de génération du programme binaire.
. GEN : unité de génération du texte FORTRAN en cas de création ou de modification.
. LIS : unité de listage du programme source FORTRAN analysé.
. MSG : unité de sortie des messages d'erreurs.

3.1.1.4. Analyse conversationnelle :

Dans cette étape d'un travail FORTRAN, le degré d'interaction système-utilisateur est particulièrement important.

3.1.1.4.1. Information échangées

Les SUPPORTS des échanges d'informations sont ceux affectés au US, SCE, CDE et MSG.

Les INFORMATIONS ECHANGEES sont de trois types :

- les messages d'erreurs signalées par le système sur l'unité symbolique MSG,
- les instructions FORTRAN émises à partir des unités symboliques SCE et CDE,
- les directives de compilation données par l'utilisateur à partir de l'US CDE.

Le FORMAT des informations échangées est celui d'un enregistrement d'au plus 80 caractères, suivi du caractère (Ret) (retour chariot).

En entrée, le caractère (Ret) signale la validation des caractères qui le précèdent .

En sortie, il marque la fin de l'enregistrement.

En entrée, les caractères NULL, LINE FEED et RUB OUT sont ignorés.

On dispose en entrée des caractères d'annulation et avec la signification suivante :

. le caractère ← annule tous les caractères qui le précèdent
. le caractère @ annule le caractère valide qui le précède immédiatement, étant admis que les caractères ← et @ ne sont pas valides.

Principe de l'échange d'informations :

Le dialogue système-utilisateur est du type QUESTION-REPONSE et dépend du mode de conversation.

Modes de conversation :

- Mode_directives_de_compilation

C'est le mode initial à l'activation de l'étape COMFOR. Il permet de demander un changement de mode ou un retour au superviseur MUCAMP.

On en sort, soit par la directive MON (retour au superviseur), soit par une directive de changement de mode.

- Mode_création

Ce mode correspond à l'entrée sur l'US CDE d'instructions FORTRAN en vue de créer tout ou une partie d'un texte FORTRAN.

L'entrée se fait à partir du mode-directives sur une commande explicite d'entrée dans le mode création.

Le retour au mode directive se fait sur une commande explicite de sortie du mode création.

- Mode_analyse

Type 1 : Avec indication d'une plage d'instructions à analyser.

Ce mode permet l'analyse sélective d'instructions d'un source FORTRAN existant, à partir de l'US SCE.

On y entre sur une directive explicite d'entrée dans le mode analyse. On en sort implicitement lorsque la dernière instruction FORTRAN à examiner est traitée.

Ce mode s'accompagne implicitement de la génération sur l'US GEN du texte FORTRAN analysé.

Type 2 : Analyse complète.

Ce mode réalise l'analyse intégrale du fichier source FORTRAN sur l'US SCE

On y entre sur une directive explicite d'analyse complète. On en sort implicitement à la rencontre, dans la source FORTRAN analysée, d'une directive FIN ou SUI.

- Mode_erreur-correction

Il est caractérisé par la saisie à partir de l'US CDE d'une ou plusieurs instructions FORTRAN qui éliminent la ou les causes d'erreurs signalées par l'analyseur.

L'entrée dans ce mode se fait implicitement lorsque le système émet un message d'erreur FORTRAN sur l'US MSG.

La sortie de ce mode se fait sur une demande explicite de l'utilisateur et se traduit par le retour dans le mode précédent.

Une vue synoptique des différents modes et des transitions est donnée en 4.5.

3.1.1.4.2. Directives de compilation

L'utilisateur fournit les directives dans le mode-directives. C'est le mode initial à l'activation de l'étape COMFOR. Il est caractérisé par l'attente d'une directive, ce que le système manifeste par l'impression d'un enregistrement réduit au caractère % sur l'US MSG.

Directive d'entrée dans le mode création :

Syntaxe : la commande se réduit après réception du caractère % émis par le système au simple caractère Ret .

Rôle : demande explicite d'entrée dans le mode création. L'utilisateur soumet à l'analyseur des instructions FORTRAN à partir de l'US CDE.

Les instructions validées par l'analyseur sont ajoutées au fichier source FORTRAN en sortie sur l'US GEN.

Les instructions reconnues comme erronées par l'analyseur font l'objet d'un message d'erreur et sont ignorées.

Avant toute saisie d'une ligne FORTRAN, le système génère sur l'US CDE le numéro de séquence de la ligne FORTRAN dans le fichier source en sortie.

Retour au mode directives :

Il se fait par le simple envoi du caractère Ret après la génération par le système du numéro de séquence de la ligne FORTRAN suivante.

Directive d'analyse intégrale :

Syntaxe : ANA (Ret) .

Rôle : demander l'analyse complète du source FORTRAN disposé sur l'US SCE.

Retour au mode directives :

A la rencontre dans le source FORTRAN de l'un des deux mots clés FIN ou SUI, le système opère le retour dans le mode directives.

Interaction système-utilisateur :

L'utilisateur ne pourra intervenir dans le déroulement de l'analyse que si le système provoque l'entrée dans le mode correction-erreur (sur la détection d'une erreur).

Directive d'analyse sélective :

Syntaxe : ANA [nolig 1] [separ] [nolig 2] (Ret)

où [nolig 1] et [nolig 2] désignent un nombre décimal et [separ] un caractère différent de LINE FEED, RUB OUT, NULL, ← et @ .

Rôle : demander l'analyse du fichier source FORTRAN disposé sur l'US SCE à partir de l'enregistrement de numéro de séquence [nolig 1] inclu jusqu'à l'enregistrement de numéro de séquence [nolig 2] inclu.
Ce mode est accompagné implicitement de la recopie sur le fichier source en sortie des instructions analysées et validées par le système.

Retour_au_mode_directives :

Lorsque l'enregistrement du numéro [nolig 2] est traité, le système provoque le retour dans le mode directives.

Interaction_système-utilisateur :

L'utilisateur ne pourra intervenir dans le déroulement de l'analyse que si le système opère, sur détection d'erreurs, la transition vers le mode erreur-correction.

Directive FIN :

Syntaxe : MØN (Ret)

Rôle : signaler la fin de l'étape COMFOR, opérer le retour au superviseur.

3.1.1.5. Structure d'un texte source FORTRAN :

3.1.1.5.1. Structure d'un_programme_FORTRAN_

Un programme FORTRAN est constitué d'une ou plusieurs unités FORTRAN. La première unité est obligatoirement le programme principal, les autres étant des sous-programmes FUNCTION ou SUBROUTINE.

Une unité se compose d'une suite d'ordre FORTRAN écrits sur une ligne.

Une unité se termine sur une directive END.

3.1.1.5.2. Structure d'un_texte_source FORTRAN_

Le texte source analysé est composé d'une suite de lignes FORTRAN d'au plus 72 caractères et se terminant par le caractère Ret .

On peut y insérer les directives SUI.

Le texte FORTRAN doit se terminer par la directive FIN.

3.1.1.5.3. Directives_intégrées_dans le_source FORTRAN

Elles sont au nombre de deux et occupent toute une ligne.

Directive SUI :

Syntaxe : SUI (Ret)

Rôle : à la rencontre dans le source FORTRAN de la directive SUI, le système opère le passage du mode création ou analyse au mode directives.
L'utilisateur peut alors changer de mode de conversation.

Directive FIN :

Syntaxe : FIN (Ret)

Rôle : la rencontre de la directive FIN signale la fin du programme FORTRAN.
S'il y a des références non satisfaites à des sous-programmes, la directive FIN provoque l'émission de la liste des sous-programmes non définis.
La directive FIN entraîne le retour au superviseur MUCAMP.
Cette directive doit nécessairement être précédée de la directive FORTRAN END.

3.1.1.5.4. Syntaxe d'une ligne FORTRAN

Le découpage d'une ligne FORTRAN tel qu'il est défini par les normes n'est pas très adapté à la saisie à partir d'un clavier de télétype.

Aussi a-t-on introduit une nouvelle syntaxe.

Plusieurs cas sont à envisager.

Cas 1 : l'instruction n'est pas étiquetée :

- le texte de l'instruction peut commencer en position 1.
  Exemple :

```
G O  T O  2 O
1 2 3....
```

Cas 2 : l'instruction est étiquetée :

- l'étiquette doit commencer obligatoirement en position 1,
- le texte de l'instruction peut suivre immédiatement.
  Exemple :

```
1  8  C  O  N  T  I  N  U  E
1  2  3  ....
```

Cas 3 : la ligne FORTRAN est une ligne commentaire :

- la position 1 doit contenir le caractère "*",
- le texte du commentaire suit.
  Exemple :

```
*   E X E M P L E
1 2 3 ...
```

Pour le reste, la syntaxe est celle définie par les normes

NF Z 65-110 - Novembre 1967.

3.1.1.6. Diagnostic et traitement des erreurs FORTRAN :

3.1.1.6.1. Diagnostic_

A l'analyse du texte FORTRAN, on distingue plusieurs types d'erreurs suivant les messages émis par le système.

Type 1 : le message émis par le système est "ERRSTX".
L'instruction en cause ne respecte par la syntaxe de FORTRAN.

Type 2 : le message émis par le système est :

ERR FOR [num]

Il s'agit d'une erreur appartenant à l'un des trois groupes suivants :

- erreurs de type sémantique,
- dépassement des ressources du compilateur,
- dépassement des ressources du calculateur.

Les numéros et les causes d'erreurs sont indiqués dans le paragraphe 4.3.

Type 3 : le message émis par le système est :

ERR END

suivi d'une liste d'étiquettes.
A la rencontre de la directive END, il reste des étiquettes non définies.

3.1.1.6.2. Traitement_

A la détection de l'erreur, la lecture est commutée sur l'US CDE et l'utilisateur peut effectuer les corrections voulues.

3.1.2. L'étape d'exécution

3.1.2.1. Rôle :

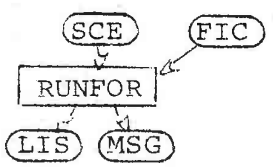
L'étape RUNFOR lance l'exécution du programme binaire utilisateur. C'est aussi un outil de mise au point. Il détecte les situations d'anomalies internes et émet des messages les décrivant.

3.1.2.2. Activation :

Elle se fait sous contrôle du superviseur MUCAMP par la commande EXE RUNFOR.

3.1.2.3. Environnement de l'étape :

L'environnement de cette étape est décrit ci-dessous :



} unités symboliques en entrée

} unités symboliques en sortie

L'affectation des unités symboliques pour cette étape est la suivante :

. SCE : unité de saisie d'un fichier autre qu'un fichier disque
. FIC : unité symbolique disque contenant le binaire du programme à exécuter
. LIS : unité de sortie d'un fichier imprimante
. MSG : unité de sortie des messages d'erreurs.

Les affectations US-UF nécessaires doivent être faites sous contrôle du superviseur DISCAMP avant l'activation de l'étape EXEFOR.

3.1.2.4. L'outil de mise au point MAPFOR :

Le système MAPFOR est intégré dans le programme binaire absolu. Il détecte les anomalies internes du programme, comme la division par zéro.

Un message d'erreur émis par le système MAPFOR sur l'US MSG est de la forme :

ERR EXE [num]

où [num] est un numéro caractérisant l'erreur.

La liste des erreurs est donnée en 4.4.

L'émission d'un tel message provoque l'arrêt du programme de l'utilisateur et le retour au superviseur.

3.1.2.5. Diagnostic et traitement des erreurs d'E/S :

Diagnostic :

Les erreurs d'E/S se répartissent en trois catégories.

Les défauts périphériques

Ce sont des erreurs physiques. Par exemple :

- panne d'un périphérique,
- erreur de transmission,
- périphérique à l'arrêt,
- périphérique pas prêt, ...

Les erreurs d'affectations

Elles sont dues, soit à l'absence d'une affectation d'US-UF,
soit à une mauvaise affectation.
Par exemple, l'affectation LIS Lr est erronée car LIS étant une unité symbolique en sortie, l'unité fonctionnelle LR (lecteur rapide) ne peut convenir.

Les erreurs logiques d'E/S

Ce sont les erreurs de programmation.
Les impossibilités de réaliser une opération d'E/S sont dues :

. soit à la taille d'un fichier : - en lecture : rencontre de la fin de fichier
- en écriture : débordement de fichier

. soit à l'incompatibilité entre le type des données lues et celui des variables d'E/S.

Exemple : soit le programme

```
      INTEGER      A
      .
      .
      READ (1,100) A
100   FORMAT (14)
```

et la donnée 18.43 E + 10 sur l'US numéro 1.
La donnée est réelle et la variable A à affecter est entière.

. soit à l'incompatibilité entre les spécifications de format et le type des variables d'E/S.

Exemple : soit le programme

```
      INTEGER   A
      .
      .
      READ (1,100)A
100   FORMAT (F8.3)
      .
      .
```

et la donnée 1465.852 sur l'US numéro 1.
Il y a compatibilité entre la donnée et la spécification de format F8.3, mais non entre la spécification flottante F8.3 et le type entier de la variable A.

La liste des erreurs est donnée en 4.4.

Traitement :

La détection d'une erreur provoque l'émission d'un message sur l'US MSG de la forme :

ERR EXE [num]

où [num] est le numéro de l'erreur.

Une erreur entraîne l'arrêt du programme utilisateur et le retour au superviseur.

3.2. Enchaînement des étapes FORTRAN :

Les étapes COMFOR et RUNFOR d'un travail FORTRAN sont activées par l'utilisateur sous contrôle du superviseur MUCAMP.

Elles doivent s'enchaîner dans cet ordre.

Le résultat de l'étape COMFOR est un programme binaire. Le rôle de l'étape RUNFOR est de charger ce programme binaire en mémoire et de l'exécuter.

4. SYNOPTIQUE.

4.1. Tableau récapitulatif des étapes :

Dans l'ordre chronologique.

NOM	FONCTION	ENVIRONNEMENT
COMFOR	Compilation : analyse de texte FORTRAN génération du programme binaire	SCE CDE GEN LIS MSG FIC OBJ
RUNFOR	Exécution : chargement du programme binaire exécutable exécution	MSG CDE FIC éventuellement : SCE LIS

4.2. Tableau récapitulatif des directives :

ETAPE	DIRECTIVE	ROLE
COMFOR	ANA sans paramètres	entrer dans le mode analyse complète du texte FORTRAN sur l'US SCE.
Compilation	ANA avec paramètres	entrer dans le mode analyse sélective définie par les paramètres
	(Ret)	si mode courant = mode directives, alors entrée dans mode création ; si mode courant = mode création, alors retour dans mode directives ; si mode courant = mode correction/erreur, alors retour dans mode précédent.
	MON	rendre le contrôle au superviseur
	AMO	générer une amorce de ruban
EXEFOR	pas de directives	

4.3. ERREURS FORTRAN.

NUMERO	RAISON
1	déclaration de type ou de genre concernant un nom de segment ou de sous-programme.
2	déclaration de type ou de genre concernant le sous-programme FUNCTION ou SUBROUTINE courant.
3	déclaration de type ou de genre concernant une variable qui a déjà une affectation mémoire et incompatible avec l'ancienne.
4	double déclaration d'un tableau.
5	déclaration incompatible avec une déclaration antérieure.
6	déclaration concernant une variable paramètre ou commune qui a déjà une affectation mémoire.
7	déclaration concernant un nom de segment ou de sous-programme.
8	dans une instruction EQUIVALENCE il y a une variable indicée faisant référence à un tableau non encore défini.
9	mise en équivalence impossible car deux au moins des variables ont déjà une affectation mémoire.
10	référence à un tableau non encore dimensionné.
11	taille d'un tableau supérieure ou maximum admis.
12	la borne variable d'un tableau n'est pas une variable paramètre.
13	la borne variable d'un tableau n'est pas une variable simple.
14	un tableau commun ne peut pas être de taille variable.

NUMERO	RAISON
15	trop de segments communs.
16	un paramètre formel est répété dans la liste des paramètres formels. paramètre formel incorrect.
17	une étiquette de COMMON a déjà été utilisée par ailleurs à d'autres usages.
18	un identificateur de tableau n'est pas indicé.
19	utilisation incorrecte d'un identificateur de segment ou de sous-programme.
20	expression arithmétique à opérandes complexes et réels double précision.
21	l'objet à assigner n'est pas une variable.
22	nom du sous-programme non valide.
23	le type, le genre ou l'implantation de l'indice de boucle est incorrect.
24	indice de boucle utilisé dans une boucle plus externe.
25	nombre maximum d'imbrications de boucle dépassé.
26	étiquette de fin de boucle déjà définie.
27	mauvaise imbrication de boucles.
28	double définition d'un sous-programme externe.
29	absence de la ligne END.
30	dans une instruction DATA une variable indicée ne peut pas avoir une borne variable.
31	constante invalide.
32	le nombre de valeurs initiales est inférieur au nombre de variables à initialiser.
33	le nombre de variables à initialiser est inférieur au nombre de valeurs initiales.

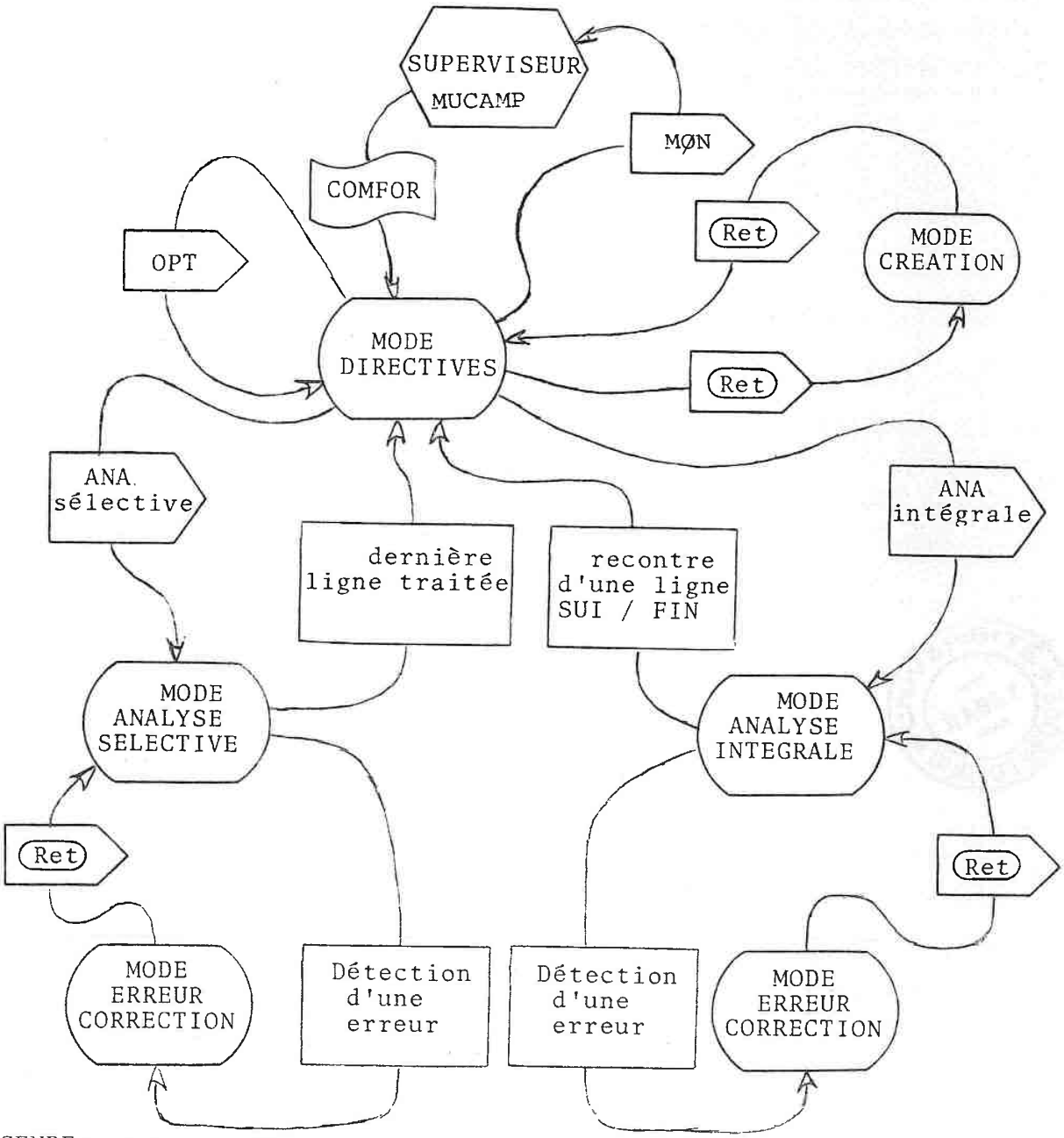
NUMERO	RAISON
34	l'instruction de branchement renvoir à la même ligne.
35	l'expression à tester n'est pas booléenne.
37	affectation invalide car expression non booléenne.
38	affectation invalide car expression non complexe.
39	affectation invalide car variable non complexe.
40	absence de l'instruction STOP dans le programme.
41	absence de l'instruction RETURN dans un sous-programme.
47	répétition d'un identificateur incorrecte.
48	trop de termes dans la déclaration.
49	mauvais cadrage d'une instruction en FORTRAN standard.
50	étiquette trop longue.
51	le nom du sous-programme n'est pas valide.
52	opérande non booléen dans une expression booléenne.
53	constante entière trop grande.
54	mauvaise utilisation d'un identificateur de sous-programme.
55	une instruction FORMAT doit être étiquetée.
56	instruction non admise dans le programme principal.
57	une instruction non exécutable ne doit pas être étiquetée.
58	l'instruction ne peut pas être atteinte.
59	l'instruction de déclaration du sous-programme est absente.

NUMERO	RAISON
60	double définition d'étiquette.
61	dans l'expression de relation l'un des termes est complexe et l'autre réel double précision.
62	étiquette de COMMON invalide.
63	lans l'expression de relation les deux termes n'ont pas le même type.
64	une expression de relation entre complexes ou logiques ne peut pas utiliser d'autres opérateurs de relations que .NE. et .EQ.
65	l'une des valeurs initiale ou finale ou l'incrément de la variable indicée n'est pas entier.
66	marque de ligne suite non attendue.
67	nombre maximum de lignes suites dépassé.
68	ligne suite attendue.
69	expression booléenne à termes non booléens ou erreur de syntaxe.
70	un indice d'une variable de tableau n'est pas entier.
71	dans une instruction READ, la liste des variables d'E/S ne peut pas être vide.

4.4. Erreur sous RUNFOR :

LIBELLE	RAISON
ERR EXE 1	division par zéro.
ERR EXE 2	débordement de tableau.
ERR EXE 3	nombre de paramètres transmis différents du nombre de paramètres attendus.
ERR EXE 4	le type d'un paramètre transmis est différent du type attendu.
ERR EXE 5	le genre d'un paramètre transmis est différent du genre attendu.
ERR EXE 6	fin de fichier en entrée.
ERR EXE 7	débordement de fichier en sortie.
ERR EXE 8	incompatibilité entre le type d'une variable d'E/S et la valeur à lire.
ERR EXE 9	incompatibilité entre le type d'une variable d'E/S et une spécification de format.
ERR EXE 10	dans une spécification de format, incompatibilité entre la longueur de la zone et le nombre de chiffres de la partie fractionnaire.
ERR EXE 11	incompatibilité entre la longueur de la zone et la nature de la spécification de format.
ERR EXE 12	caractère incorrect en entrée.
ERR EXE 13	dépassement de capacité format E, en entrée.
ERR EXE 14	rencontre de la fin de l'enregistrement sans spécification de passage à l'enregistrement suivant.

4.5. Diagramme des modes de conversation et des transitions sous COMFOR :



LEGENDE

- Commande sous MUCAMP
- Directive de l'utilisateur sous COMFOR
- Evènement du système COMFOR
- Mode de conversation
- Transition

NOM DE L'ETUDIANT : MERSDORF JEAN-CLAUDE

NATURE DE LA THESE : DOCTORAT DE SPECIALITE EN INFORMATIQUE option "système"

VU, APPROUVE

& PERMIS D'IMPRIMER

NANCY, le 16/9/1976

LE PRESIDENT DE L'UNIVERSITE DE NANCY I

M. BOULANGE