UNIVERSITE DE NANCY I

U. E. R. DE MATHEMATIQUES

mise en oeuvre de la méthode de programmation déductive

thèse

présentée pour l'obtention du diplôme de docteur-ingénieur en informatique

par

bernard huc

soutenue le 31 mai 1977

JURY : Président M. C. PAIR

Examinateurs M. M. GRIFFITHS

M. J.C. DERNIAME

Mme F. BELLEGARDE

UNIVERSITE DE NANCY I
U. E. R. DE MATHEMATIQUES

mise en oeuvre de la méthode de programmation déductive

thèse

présentée pour l'obtention du diplôme de docteur-ingénieur en informatique

par

bernard huc

soutenue le 31 mai 1977

JURY : Président M. C. PAIR

Examinateurs M. M. GRIFFITHS

M. J.C. DERNIAME

Mme F. BELLEGARDE

A mes Parents,

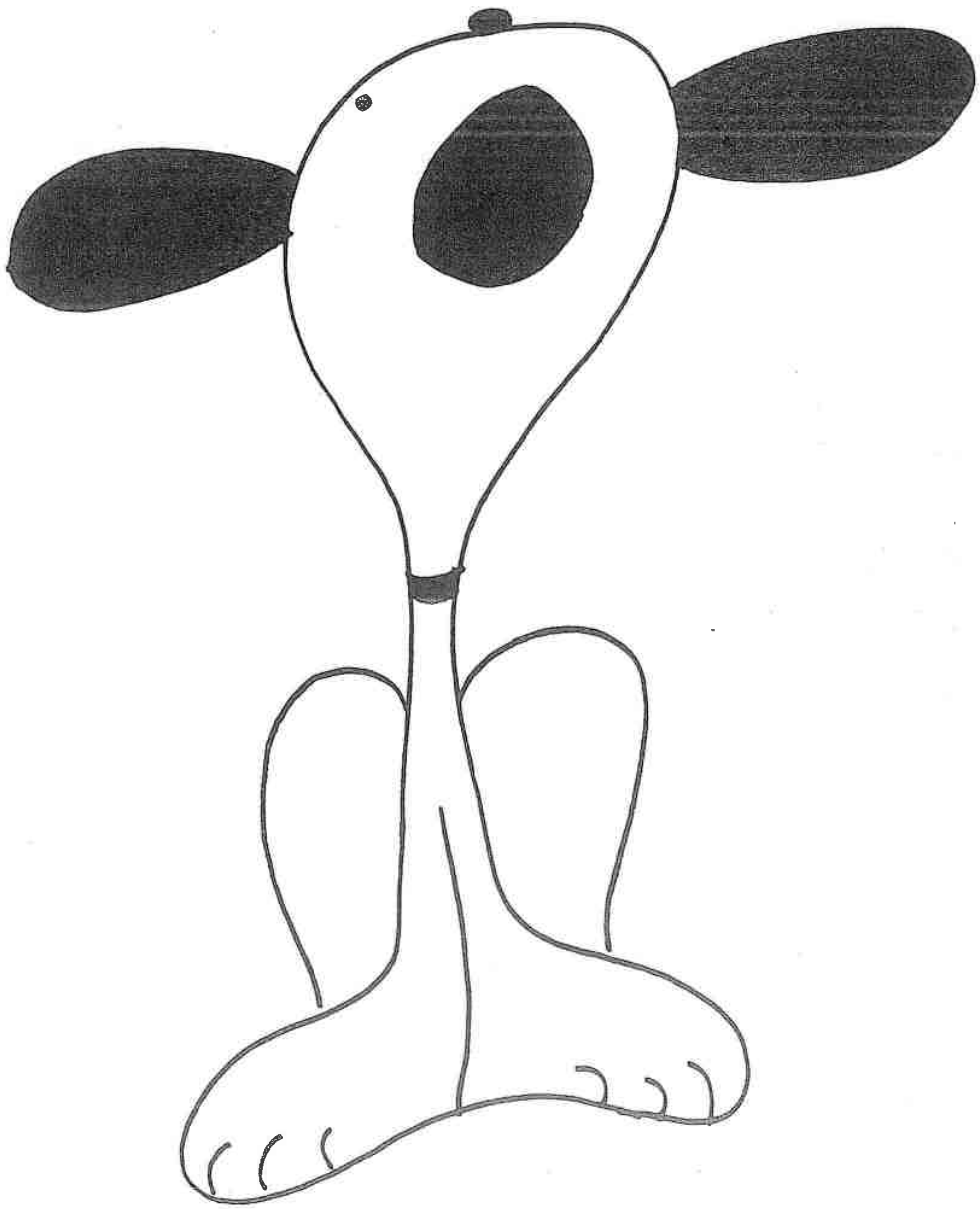
A Dominique

Je remercie tout d'abord Monsieur PAIR, Président de l'Institut National Polytechnique de Lorraine, pour les conseils et le soutien qu'il n'a cessé de me prodiguer et à qui je dois ce sujet.

Je tiens aussi à remercier Monsieur GRIFFITHS, Professeur à l'IUT d'informatique de Nancy, Monsieur DERNIAME, Maître de Conférences à l'Université de Nancy I et Madame BELLEGARDE, Maître-Assistant à l'IUT d'Informatique de Nancy, d'avoir bien voulu prendre connaissance de mon travail et participer au Jury.

J'exprime toute ma gratitude aux utilisateurs de la première version de SNOOPY dont les remarques pertinentes m'ont permis d'améliorer mon travail et en particulier Monsieur et Madame QUERE.

J'adresse également tous mes remerciements à Madame FERRARI, Mme GAUTIER, Mme DUCLOY et Mle LE MARECHAL pour la gentillesse dont elles ont fait preuve dans la réalisation matérielle de cette thèse.



sommaire

I

la méthode de programmation déductive

RESUME

Ce chapitre a pour but de montrer l'évolution de la programmation ces derniers temps. On y présentera quelques méthodes de programmation actuellement utilisées. A partir de l'étude d'une démarche adoptée lors de la résolution des problèmes on introduira la Méthode de Programmation Déductive.

PLAN

1.1. *LE "MOYEN-AGE" DE LA PROGRAMMATION*

Pendant très longtemps, la mise en oeuvre de la programmation a été très aléatoire. En effet, l'inexistence de méthode d'analyse et l'utilisation de langages de programmations mal adaptés ne facilitaient pas la tâche du Programmeur.

L'écriture d'un programme de taille importante était une expédition pleine de traquenards ; seule une longue expérience et le recours à de multiples astuces permettaient au programmeur de venir à bout des problèmes qui lui étaient posés. Encore fallait-il prévoir une période de tests plus ou moins longue pour réparer (le plus souvent par des verrues) les "petits oublis" et les "petites erreurs". Après ce processus, on arrivait généralement à un programme qui tournait "presque" bien... On n'avait plus alors qu'à espérer que le programme n'aurait pas à être modifié. En effet, une modification un peu trop profonde risquait de changer involontairement certaines autres parties du programme. A tel point que pour mener à bien une modification importante, il était souvent plus prudent de réécrire un nouveau programme.

Le tableau ci-dessus peut sembler quelque peu sombre et caricatural, mais je pense qu'il est quand même assez proche de la vérité. On pourrait comparer cette période au "Moyen-Age" de la Programmation.

Depuis quelques années, il semble qu'une volonté quasi-unanime des Informaticiens veuille faire progresser la science de la Programmation et que grâce à cet effort nous soyons entrés,en quelque sorte, dans la "Renaissance" de la Programmation.

A quoi est dû ce changement ? En effet, depuis assez longtemps, les nombreuses difficultés que soulèvent la programmation ont été recensées.

- Fiabilité des produits obtenus
- Apprentissage de la programmation
- Rentabilité des équipes de programmeurs
- Décomposition des gros travaux
- Modification des programmes existants

a) Fiabilité des produits obtenus

On ne pouvait jamais être sûr de la justesse d'un produit et l'on jugeait que celle-ci était atteinte lorsque l'on constatait une absence d'erreur d'exécution pendant des périodes de tests plus ou moins longues ,où l'on essayait de tester les diverses possibilités du programme.

b) Apprentissage de la Programmation

L'enseignement de la programmation se résumait à l'assimilation de langages de programmation et d'un certain nombre, non exhaustif, de "recettes". Ensuite l'expérience du programmeur lui permettait d'apprendre des astuces (comment utiliser le moins de place possible... en rendant le programme de moins en moins lisible ...)

c) Rentabilité des équipes de programmeurs

Cette rentabilité semblait quelque peu dérisoire, car, même si l'écriture du programme était parfois rapide, la période de tests, et des modifications qu'ils imposaient, était souvent longue et délicate.

d) Décomposition des gros travaux

Ce problème a été pressenti très tôt et on a essayé très vite d'y apporter des solutions plus ou moins heureuses. En effet, la résolution de gros travaux nécessite l'intervention de plusieurs personnes ayant chacune un morceau du travail à effectuer, cette

décomposition pose de nombreux problèmes de synchronisation et de cohérence. Les méthodes de décomposition initialement utilisées manquaient bien souvent de rigueur et entraînaient parfois des découpages "en dépit du bon sens".

e) Modification des programmes existants

Cette tâche était très délicate. En effet, les programmes étaient truffés d'astuces, (souvent compréhensibles par l'auteur uniquement) et la mise en oeuvre d'une modification risquait bien souvent d'entraîner des changements imprévus dans les autres parties du programme.

On a très rapidement cherché à surmonter ces diverses difficultés. Malheureusement, les moyens utilisés pour cela n'étaient pas assez méthodiques et étaient encore loin de simplifier la tâche des programmeurs.

1.2. *LA "RENAISSANCE" DE LA PROGRAMMATION*

Très rapidement on s'est rendu compte de la nécessité de créer des langages "mieux adaptés" à la résolution des problèmes. FORTRAN, qui s'il est à l'heure actuelle un peu renié, est ne l'oublions pas une des premières étapes dans cette voie !!

Depuis l'esprit créateur des informaticiens a donné naissance à bien des langages. Chaque année voit l'apparition de nouveaux langages [1] . A chaque fois le nouveau venu est le langage "idéal" !! Ces langages sont soit des langages originaux (SIMULA, LISP, SNOBOL, Pascal, PL/1, Algol ...) soit des améliorations des langages déjà existants (PYGOL, où l'on a cherché à créer un langage hybride de PL/1 et d'Algol 60 [2] afin de ne prendre que le meilleur de chacun,

FORTRAN structuré, où l'on a essayé d'adapter le langage à la programmation structurée en adjoignant au FORTRANIV Standard de nouvelles constructions syntaxiques du genre IF .... THEN.... ou REPEAT, ...). [3]

Tous ces langages peuvent être rangés dans deux catégories :

- les langages spécialisés, ayant pour vocation d'être utilisés dans des applications particulières (SIMULA, LISP, COBOL, SNOBOL, ALGOL LO, EUCLID [4])

- les langages "généraux" , ayant pour ambition d'être utilisables pour des types de problème très variés (PL/I, Pascal, Algol 68, ...)

Mais on s'est rendu compte bientôt que l'amélioration de la Science de la Programmation ne dépendait pas seulement de l'utilisation de langages "parfaits", mais aussi et surtout de l'utilisation de méthodes de programmation. Depuis quelques années, on a vu apparaître de nombreuses méthodes. Dans le paragraphe suivant nous allons décrire rapidement trois méthodes de programmation actuellement utilisées, afin d'en découvrir les qualités mais aussi les défauts.
Ces trois méthodes sont les suivantes :

- le langage d'organigrammes [5]
- la méthode L.C.P. (1) [6]
- la programmation par raffinement successif (aussi appelée programmation structurée) [7][8]

La plupart des méthodes descendantes utilisées à l'heure actuelle s'apparentent pour la plupart à l'une des méthodes citées ci-dessus. Un autre type d'amélioration où sont, à l'heure actuelle, faits beaucoup d'efforts est la mise au point et la vérification de programmes déjà écrits [9] [10]

(1) Langage de Construction de Programmes

1.3. *ETUDE DE QUELQUES METHODES DE PROGRAMMATION*

1.3.1. Les langages d'organigramme

Depuis très longtemps, les organigrammes ont été utilisés. En effet, la lisibilité d'un organigramme est indéniable ; de plus celui-ci permet une très bonne visualisation de l'exécution d'un algorithme. Cette visualisation est d'ailleurs un outil précieux pour la mise au point de programmes.

D'outil de représentation qui était leur rôle initial, les organigrammes ont vu leur utilisation se structurer progressivement. Cette structuration a été à l'origine de ce que l'on pourrait appeler des langages d'organigrammes. [5]

Cette évolution a amené à compléter les primitives initiales par un certain nombre de primitives plus élaborées ; ces nouvelles primitives étant construites à partir des premières. D'autre part certaines règles d'utilisation de ces primitives ont été mises au point. Pour faciliter la modularité des programmes, on a autorisé certains "pavés" à ne plus contenir des instructions mais seulement une indication indiquant le travail à y accomplir ;

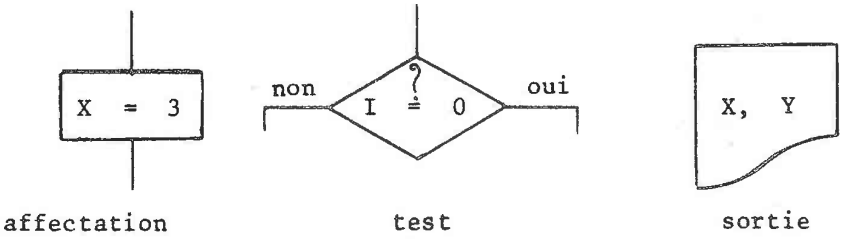


Fig. 1 Quelques primitives simples des organigrammes

dans un second temps ces pavés sont à leur tour représentés par des organigrammes plus détaillés.

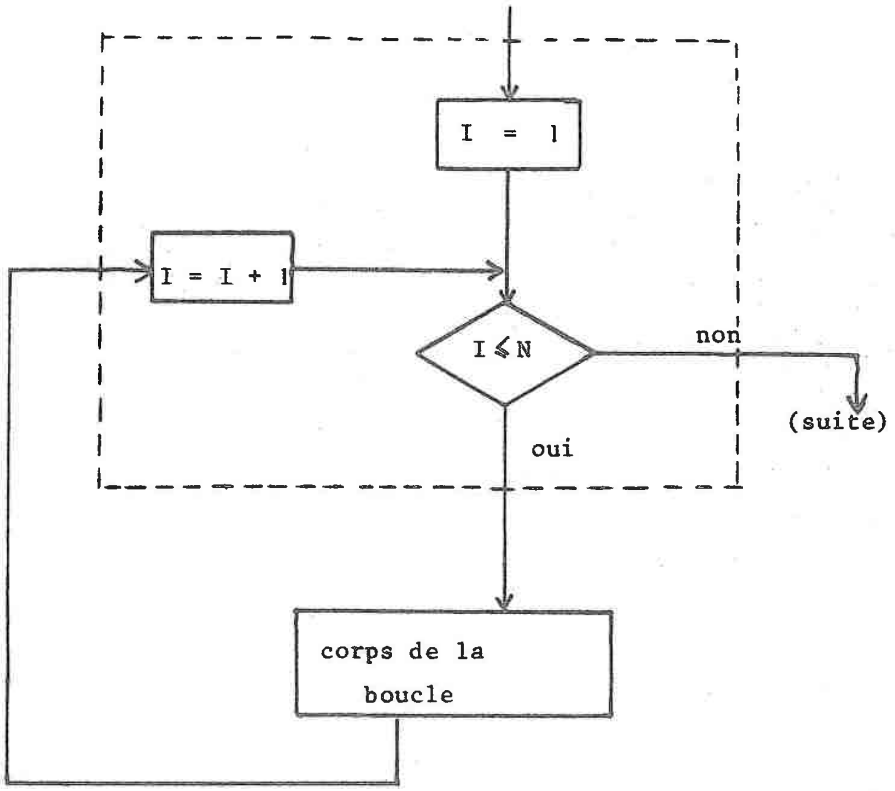


Fig. 2 Une primitive élaborée (boucle avec nombre connu d'itération)

La méthode d'élaboration de l'organigramme reste malgré tout assez hasardeuse et le langage d'organigrammes reste principalement un langage descriptif. Certains apports ont été faits à ces langages d'organigrammes afin justement d'en donner des méthodes de construction [11].Ces méthodes de construction reposent principalement sur deux techniques :

- éclatement d'un organigramme ou encore construction descendante de l'organigramme.

- simplification de l'organigramme après chaque éclatement.

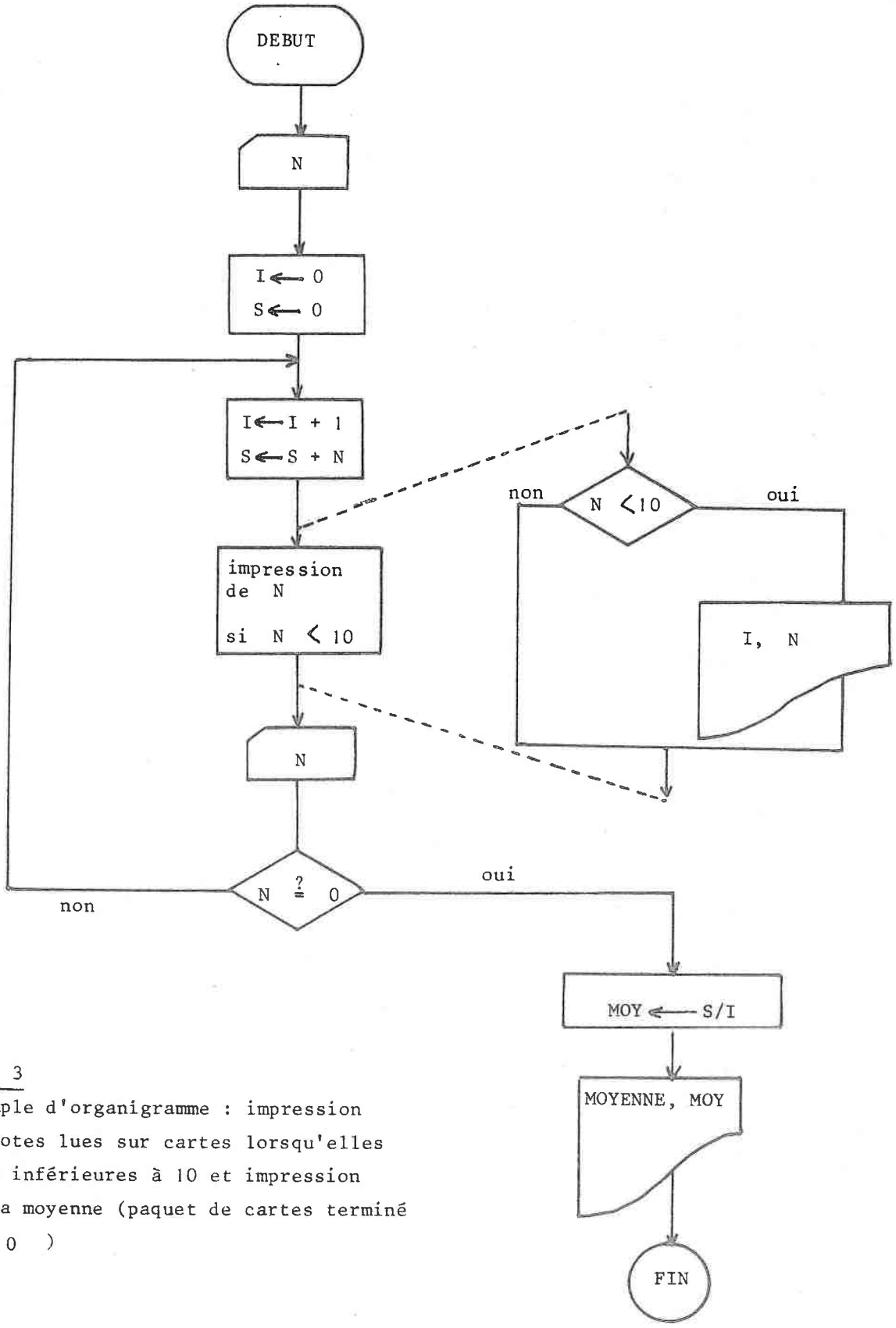


Fig. 3

Exemple d'organigramme : impression de notes lues sur cartes lorsqu'elles sont inférieures à 10 et impression de la moyenne (paquet de cartes terminé par 0)

1.3.2. La Méthode L.C.P.

Cette méthode est un exemple de méthode de programmation destinée à une utilisation particulière (en l'occurence, la gestion)[6]

L'idée de départ est de ne pas favoriser les entrées, les sorties ou les procédures de traitement mais de chercher à élaborer un programme en tenant compte de ces trois types d'informations. L'ensemble des traitements est en effet considéré en tant que fichier d'informations au même titre que ceux des données et des résultats.

Cette méthode est une méthode descendante et une partie importante de la construction du programme est consacrée à la hierarchisation des divers fichiers d'informations (entrées, sorties, traitement).

La mise en oeuvre de cette méthode se fait en trois étapes :

a) définir la structure hiérarchique des résultats

b) définir la structure hiérarchique des données en tenant compte des résultats et des traitements.

c) organiser le programme à partir des entrées et le contrôler par les sorties.

L'organisation du programme se fait elle-même en deux phases :

a) définir la structure hiérarchisée du programme. Cette définition aboutit à une définition du programme en tant qu'ensemble ordonné de séquences logiques.

b) organisation détaillée (niveau des instructions).

On appelle séquence logique tout l'ensemble d'instructions du programme exécutées le même nombre de fois au même endroit du programme.

On trouvera dans la figure 4a un exemple de hiérarchisation du programme déjà traité dans le paragraphe précédent.

On peut vérifier la structure hiérarchique en remarquant qu'il y a application de chacun des ensembles dans tous les niveaux supérieurs (cf Fig. 4b).

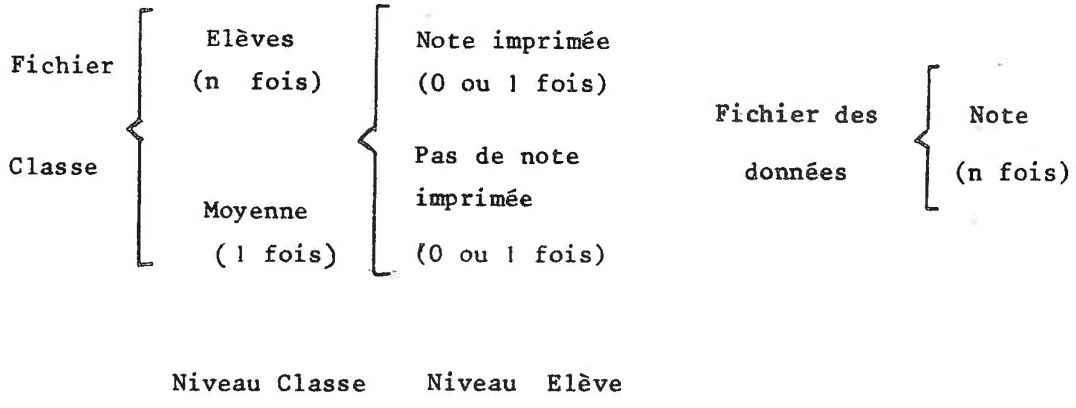


Fig. 4a Structure hiérarchisée d'un problème (résultats, données)

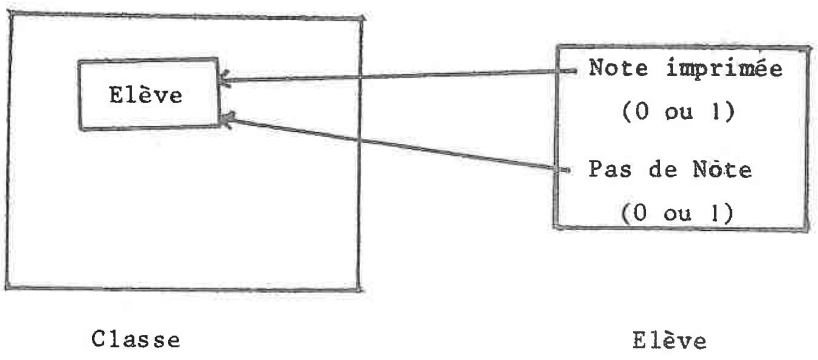


Fig. 4b Application d'un sous ensemble de niveau supérieur dans un autre.

L'organisation détaillée du programme est donc entreprise lorsque les divers ensembles d'informations ont été décomposés en sous ensembles élémentaires. Cette décomposition est accomplie à l'aide de règles de subdivisions appropriées. Les deux types de structure élémentaires utilisés sont :

- la structure alternative
- la structure itérative

L'organisation détaillée s'effectue en établissant tout d'abord les listes d'instructions par catégories (lectures, calculs, etc...). Ces listes sont élaborées dans un ordre bien défini en précisant pour chaque instruction à quelle séquence logique elle appartient.

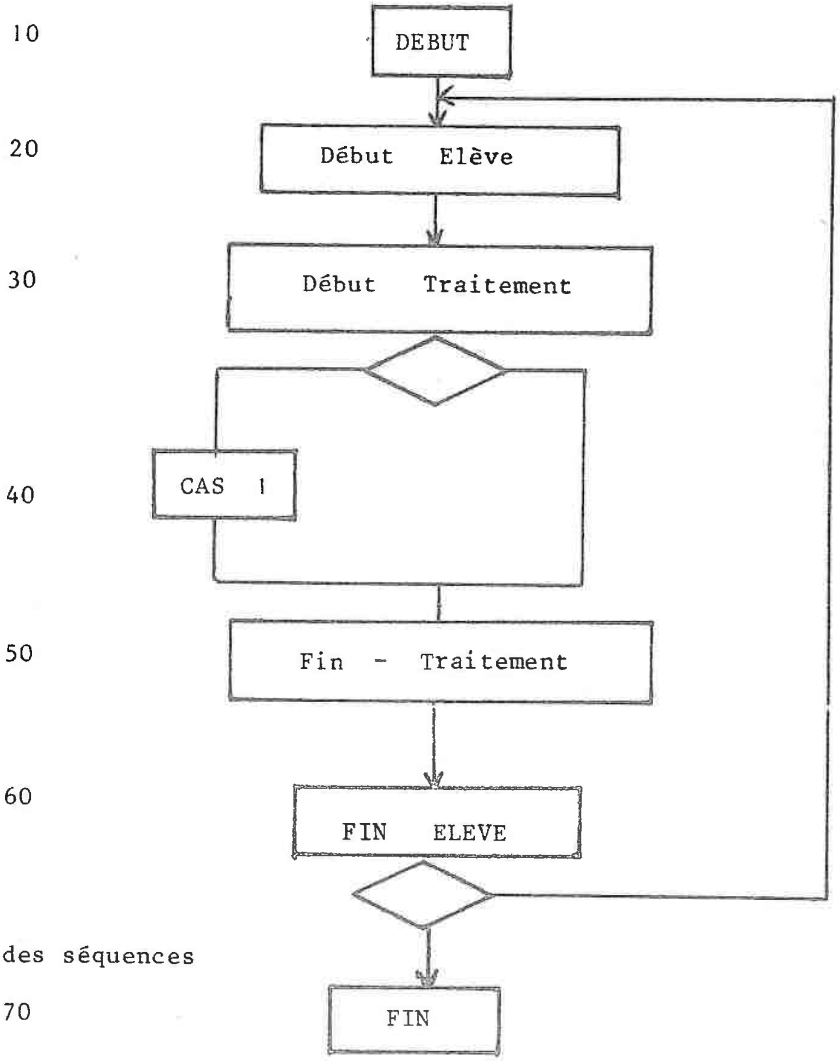


Fig.4c.
Organigramme des séquences logiques.

L'ordre dans lequel sont établies les listes d'instructions est le suivant :

a) liste des lectures

b) liste des branchements : elle est simple à obtenir car les branchements figurent sur l'organigramme des séquences logiques.

c) préparation des branchements. Cette préparation est faite en tenant compte des données effectivement disponibles par les lectures.

d) liste des calculs : il ne faut pas oublier les éventuelles mises à zéro des zones où seront effectués des traitements itératifs.

e) liste des sorties.

Après avoir vérifié l'application des ensembles d'instructions dans l'ensemble des séquences logiques, on établit la liste ordonnée des instructions par ensemble de données à traiter. L'ordre de ces listes est le suivant :

a) préparation des branchements

b) préparation des calculs et calculs

c) préparation des sorties et sorties

d) lectures

e) branchements

10 - lecture de la première note 60 - lecture d'une note	}	lectures
30 - Si N $>$ 10 , 50 60 - Si N $\neq$ 0 , 20	}	branchements
10 - Initialisation de I à 0 10 - Initialisation de S à 0 20 - Calcul de I (+) 20 - Calcul de S (+) 70 - Calcul de Moy (÷)	}	calculs

40 - Impression de N
70 - Impression de Moy } Impressions

Fig. 4d Liste des instructions par catégorie

Lorsque ce travail est accompli , il ne reste plus avant de passer au codage des instructions, qu'à vérifier que chaque sortie est programmée dans la séquence adéquate. On s'aide pour cela du tableau de description des sorties (fig. 4a).

Cette méthode a une grande qualité : grâce à la hiérarchisation des différents fichiers et au contrôle de leur interdépendance, elle permet d'écrire des programmes structurés et assez fiables. On pourrait toutefois émettre quelques critiques :

- Si le principe de hiérarchisation est valable, son application est par moment trop rigide.

- L'établissement des listes d'instructions par catégorie a tendance à cacher un peu au programmeur la vision globale du problème et la manière dont elle est menée peut conduire à oublier certaines instructions.

- Cette méthode, très utilisable lors de la résolution de certains problèmes de gestion (rupture, etc...), est un peu trop spécialisée et s'adapte très difficilement à la résolution de problèmes de types différents.

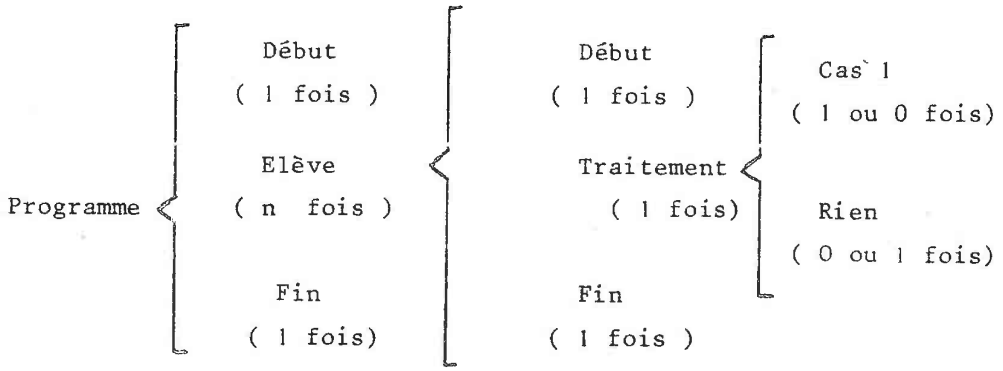


Fig. 4e Structure du programme

```
10  - { I ← 0
      { S ← 0
      { lire N

20  - { I ← I + 1
      { S ← S + N

30  -   Si  N > 10 , 50

40  -   écrire  N

50  - { lire  N
      { Si  N ≠ 0, 20

70  - { Moy ← S - I
      { écrire Moy
```

Fig. 4f Liste ordonnée des instructions par séquence.

1.3.3. La Programmation Structurée

Nous allons maintenant parler de la programmation structurée (ou par raffinements successfis) [7] [8]. De même que la précédente, cette méthode est une méthode descendante.

Le principe en est de décomposer progressivement la résolution du problème en sous-problèmes élémentaires. Ces décompositions successives s'arrêtent lorsque tous les sous-problèmes sont décrits d'une manière élémentaire par les instructions du langage de programmation utilisé. De plus on vérifie à chaque décomposition que le programme obtenu effectue bien le travail demandé. Pour cela, on peut utiliser un système de preuves formelles. Chaque morceau de la décomposition est vérifié à l'aide d'assertions. Ci-dessous est représentée l'analyse du problème de la moyenne à l'aide de la programmation structurée (le langage utilisé est PASCAL).

```
var   moy  :  real ; n, s : integer ;
  begin
      Calcul de la Somme des notes
      et impression des notes
      inférieures à 10
       moy  : =  S/n ;
      write ln ('moyenne' , moy)
end.
```

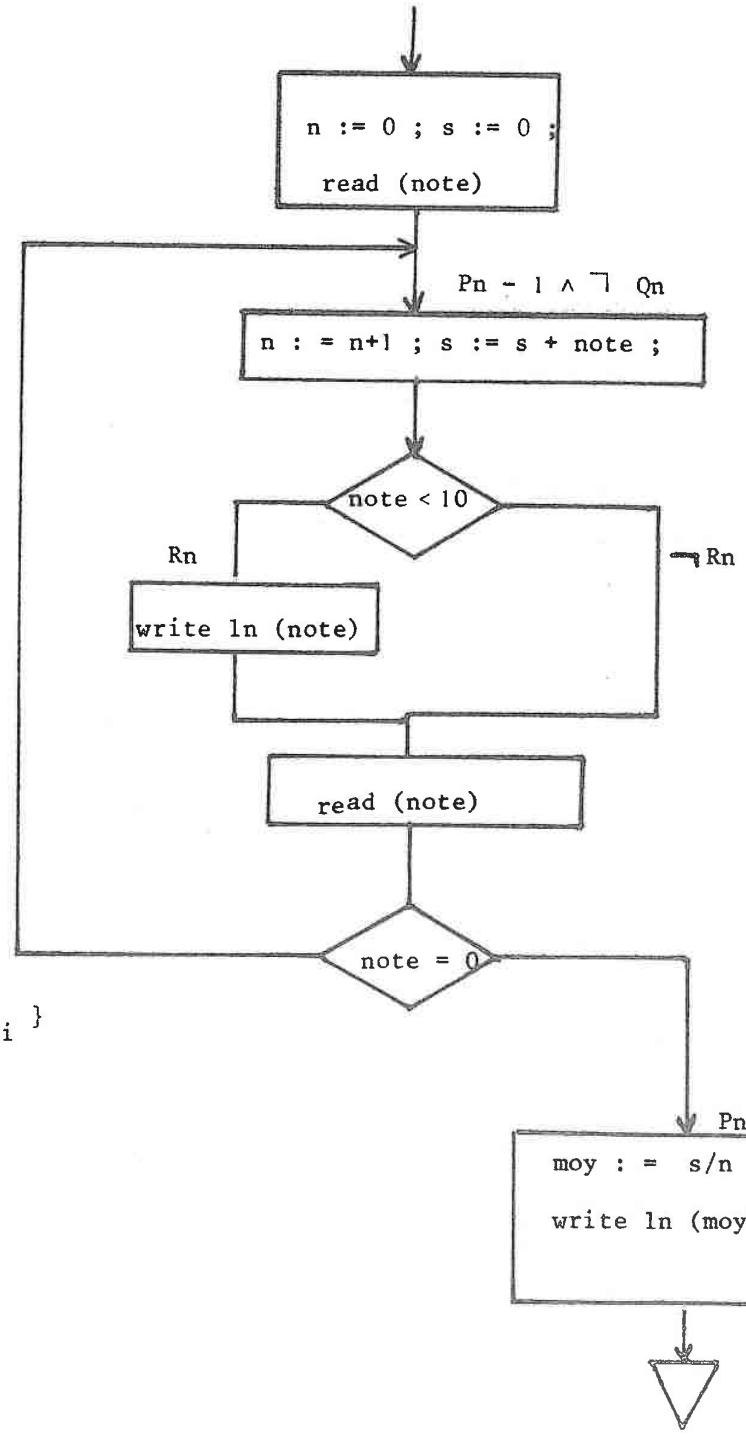
5a

```
var   moy : real ; n, s, note : integer ;
  begin
   n : = 0 ; s : = 0 ; read (note) ;
  repeat { s_n = Σ_{i=0}^{n-1} note_i  ∧ note_n ≠ 0}
    n : = n + 1 ; s := s + note ;
    impression de la note si elle est
          < 10 ;
  read (note)
  until   note = 0 ;
  { s_n = Σ_{i=0}^{n} note_n   ∧ note_{n+1} = 0 }
  moy : = s/n ;
  write ln ('moyenne' , moy)
end.
```

5b

```
var  moy  :  real ; n, s, note : integer ;
  begin
   n : = 0 ; s : = 0 ; read (note) ;
    repeat  {s_n = Σ_{i=0}^{n-1} note_i  ∧ note_n ≠  0 }
      n : = n + 1 ; s : = s + note ;
  if note < 10 then write ln (note) { note< 10 }
               else   { n  ≥ 10 } ;
  read (note)
until  note = 0 ;
   { s_n = Σ_{i=0}^{n} note_n  ∧  note_{n+1} = 0 }
moy  : = s/n
write ln ('moyenne' , moy)
```

5 c



$Pn = \{s_n = \sum_{i=0}^{n} note_i \}$

$Qn = \{note_n = 0 \}$

$R_n = \{note_n < 10 \}$

5 d

Fig. 5 Résolution du problème de la moyenne par raffinement successif

1.3.4. Qualités et défauts des méthodes de programmation

Quels sont les apports de ces méthodes et de ces nouveaux langages à la programmation ? Ont-ils résolu tous les problèmes ? Si non quels sont ceux qui restent encore à résoudre.

Il est indéniable que cet effort dans le renouveau de la programmation a porté ses fruits :

a) le fait d'utiliser une méthode ne peut que faciliter la coordination au sein des équipes de programmeurs et l'enseignement de la programmation.

b) les programmes et les analyses obtenus sont devenus plus lisibles et structurés. Cette structuration même permet à l'analyse d'être beaucoup plus modulaire et par la même plus aisée à modifier.

c) les diverses méthodes employées aboutissent à des programmes plus fiables car plus simples à tester et à prouver. En effet, en parallèle avec ces méthodes se sont développées des méthodes de preuves de programmes élaborées. On cherche à ne plus vérifier les programmes par de longues périodes de tests, mais en cours d'écriture.

d) la plupart des méthodes distinguent nettement trois types de traitement :

- le traitement séquentiel
- le traitement conditionnel
- le traitement itératif

De par ces distinctions, les programmes obtenus sont plus "propres" notamment par la suppression de nombreux GOTO intempestifs.

Cependant, certains obstacles n'ont pas encore été supprimés :

a) dans certains cas l'utilisation de certains langages de programmation comme outils d'analyse risque de géner le programmeur.

b) toutes ces méthodes et tous ces nouveaux langages de programmation lient dès le début de l'analyse la résolution du problème à son exécution. Cette introduction prématurée de la notion de dynamique dans l'analyse géne souvent les programmeurs :

- utilisation délicate des affectations et des variables
- contraintes imposées par la recherche de l'ordre des instructions lors de la définition du problème

Le problème des affectations préoccupe certains informaticiens, et après la chasse aux GOTO il semble que depuis quelques temps, la chasse aux affectations et aux variables soit ouverte !!!

Nous allons dans le paragraphe suivant présenter une méthode de programmation dont la démarche est inspirée par une réflexion sur la méthode de résolution d'un problème.

1.4. *LA METHODE DE PROGRAMMATION DEDUCTIVE*

1.4.1. La démarche adoptée lors de la résolution d'un problème

Nous allons essayer de dégager une méthode de programmation à partir de l'observation des différentes étapes de la résolution d'un problème informatique [12, 13, 14].

Ces diverses étapes sont au nombre de quatre :

a) Recherche d'un énoncé du problème.

Un problème étant donné, il faut en premier lieu, chercher à en faire apparaître clairement le but, ou, en d'autres termes, donner une définition éventuellement informelle, mais toujours précise des résultats.

Il est difficile de donner une démarche précise pour mener à bien cette première étape. En effet, le langage employé pour définir le résultat est très lié au type de problème que l'on cherche à résoudre (problème de gestion, problème lexicographique, problème arithmétique, etc...)

b) Explication de l'énoncé

Dans un second temps, il faut donner une définition explicite des divers résultats. On entend par définition explicite, une définition formelle permettant de définir un résultat à l'aide d'un ou plusieurs nouveaux résultats intermédiaires qui seront introduits à l'occasion.

$x = a * b + 27$ (1)

$y : y_0 = 0, 3$; pour 1 de 1 jqa 10 répéter $y_1 = y_{1-1} * 0.5$ (2)

Fig. 6 Exemples de définitions explicites

Ces résultats intermédiaires devront à leur tour être définis de la même manière. Il faudra tout d'abord formuler des énoncés pour chacun de ces résultats, puis en donner des définitions explicites. Lorsque tous les résultats (initiaux et intermédiaires) sont définis explicitement, on a obtenu une définition explicite du problème. Cette définition est en fait un ensemble, non ordonné, de définitions explicites.

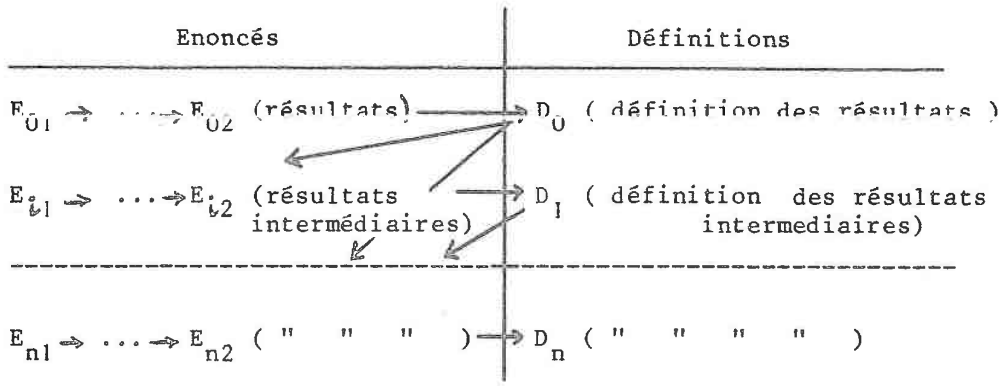


Fig. 7 Définition d'un problème à partir de son énoncé E_{01}

(Ei_1 et Ei_2 désignent respectivement pour le i ème résultat le premier énoncé formulé et l'énoncé auquel on a abouti par transformation avant explication).

L'ensemble des énoncés fournit donc une sorte de lexique donnant pour chaque résultat une définition (formelle ou informelle). La résolution du problème va donc consister en un va et vient entre ce lexique et l'ensemble des définitions explicites. Celle-ci sera terminée lorsque tous les énoncés auront été explicités.

Lexique		Définitions	
Résultat	: impression du nom et du salaire d'un employé	résultat	: écrire (nom, salaire)
Nom	: de l'employé	nom	= donnée
Salaire	: de l'employé	salaire	= nh x sh
nh	: nombre d'heures	nh	= donnée
sh	: salaire horaire (15F)	sh	= 15

Fig. 8a Exemple de définition d'un problème simple

c) Obtention d'un algorithme par l'ordonnancement des définitions.

Lorsque le problème est entièrement défini, il faut, pour obtenir un algorithme, préciser dans quel ordre devront être exécutées les différentes définitions. Les définitions avaient été faites de manière statique ; l'ordonnancement transforme l'ensemble des définitions formelles en une liste de définitions formelles (instructions). Le mécanisme de l'ordonnancement est très simple : une définition ne peut être exécutée que si tous les objets qu'elle utilise ont été définis auparavant. Ce réordonnancement est comme, on peut s'en apercevoir, une étape indépendante de la précédente.

Il faut noter que la manière d'ordonner certains sous ensembles de définitions n'est pas unique. Dans certains cas, cette liberté d'ordonnancement peut être restreinte par des contraintes extérieures au problème (ordre des lectures, ordre des écritures).

```
nom        =    donnée
nh         =    donnée
sh         =    15
salaire    =    nh  x  sh
écrire ( nom, salaire )
```

Fig. 8b Un des réordonnancement possible pour les définitions

Par exemple dans l'exemple 8b, l'ordre des trois premières définitions est indifférent tant que l'on n'a pas imposé un ordre dans la lecture du nom et du nombre d'heures de l'employé.

d) Traduction de l'algorithme dans un langage de programmation.

Une dernière étape consiste à traduire l'algorithme obtenu dans un langage compréhensible par l'ordinateur. La traduction est simple à obtenir. Elle sera faitede manière systématique. Les déclarations du programme seront obtenues à partir du lexique où figurent les définitions informelles des résultats. On peut remarquer que la traduction des définitions formelles peut être omise si ces dernières sont déjà exprimées dans un langage connu de l'ordinateur.

```
      INTEGER  NOM
      REAL NH, SH, SALAIRE
      NAMELIST NOM, NH
C  LECTURE DE NOM ET NH
      INPUT
      SH = 15.0
      SALAIRE = NH * SH
      OUTPUT NOM, SALAIRE
      STOP
      END
```

```
var  nom : alfa ;
     sh, nh, salaire : real ;
begin

     read (nom, nh) ; sh:= 15.0 ;
     salaire := sh * nh ;
     write (nom, salaire) ; write ln
end.
```

Fig. 8c Exemples de traductions systématiques

1.4.2. Les caractéristiques d'une telle démarche

Une caractéristique fondamentale de cette méthode est de définir les résultats de manière déductive. En effet, les objets ne sont introduits, puis définis (d'abord dans le lexique et ensuite explicitement) que lorsqu'ils ont été utilisés. Cette démarche permet de connaître à tout moment quels sont les objets à définir ; d'autre part, elle facilite la tâche du programmeur qui n'a plus à "prévoir" les objets dont il aura besoin pour la résolution de son problème.

Une autre caractéristique de cette démarche, qui est liée d'ailleurs à la précédente, est de définir le problème d'une manière totalement statique. En effet, les aspects dynamiques ne sont introduits dans la résolution que lors de la phase d'ordonnancement. Ce caractère statique a un grand avantage : lorsque l'on défini un objet, on n'a pas à se préoccuper de l'état des divers autres objets utilisés dans le problème.

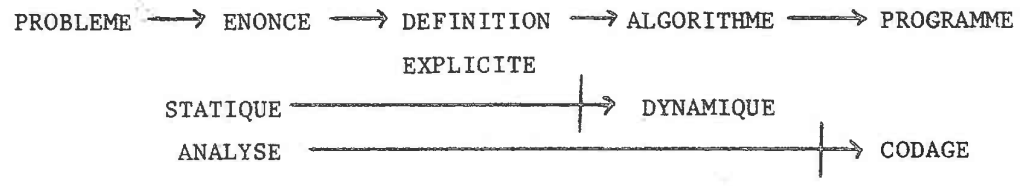


Fig. 9 La Résolution Déductive d'un problème

Une troisième caractéristique est d'avoir pour chaque objet utilisé dans la résolution deux types de définitions :

- une définition formelle ou informelle (le lexique) expliquant clairement la signification de cet objet. Chaque définition informelle du lexique est l'énoncé d'un sous-problème que l'on cherche à résoudre.

- une définition explicite qui est obtenue à chaque fois à partir de l'énoncé correspondant par une transformation et permet d'introduire de nouveaux objets et donc de nouveaux énoncés dans le lexique.

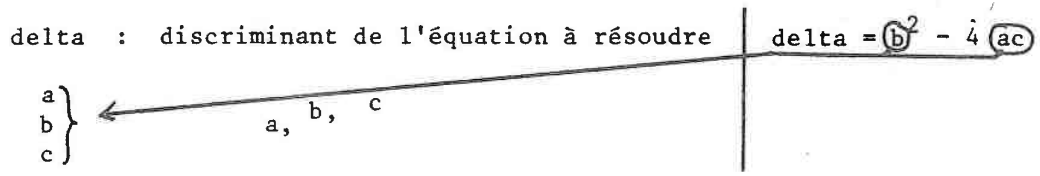


Fig. 10 Les deux types de définition

Cette démarche permet d'autre part de séparer nettement l'analyse du problème et le codage de l'analyse. Cette séparation évite au programmeur les contraintes imposées par le langage de programmation qu'il veut utiliser et qui ne sont pas justifiées par l'analyse (GOTO, déclarations en tête de programme, etc ....)

En résumé, cette démarche :

- est déductive
- est statique
- donne pour chaque objet utilisé deux types de définitions (une formelle et une informelle)
- sépare nettement analyse et codage.

Une troisième caractéristique est d'avoir pour chaque objet utilisé dans la résolution deux types de définitions :

- une définition formelle ou informelle (le lexique) expliquant clairement la signification de cet objet. Chaque définition informelle du lexique est l'énoncé d'un sous-problème que l'on cherche à résoudre.

- une définition explicite qui est obtenue à chaque fois à partir de l'énoncé correspondant par une transformation et permet d'introduire de nouveaux objets et donc de nouveaux énoncés dans le lexique.

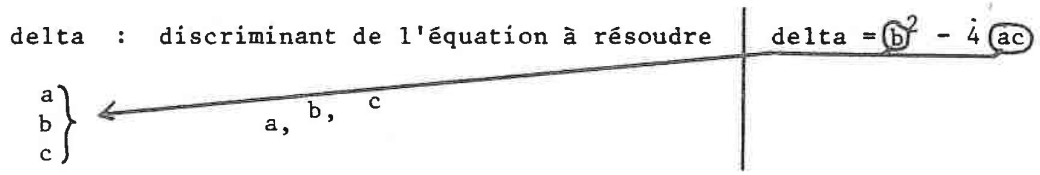


Fig. 10 Les deux types de définition

Cette démarche permet d'autre part de séparer nettement l'analyse du problème et le codage de l'analyse. Cette séparation évite au programmeur les contraintes imposées par le langage de programmation qu'il veut utiliser et qui ne sont pas justifiées par l'analyse (GOTO, déclarations en tête de programme, etc ....)

En résumé, cette démarche :

- est déductive
- est statique
- donne pour chaque objet utilisé deux types de définitions (une formelle et une informelle)
- sépare nettement analyse et codage.

1.4.3. Le langage de formalisation des définitions

Pour mettre en oeuvre une méthode de programmation déductive, il faut disposer d'un langage de formalisation des définitions qui nous aide à transformer les énoncés du lexique en définitions explicites. Dans les cas simples, les définitions formelles définissent un ou plusieurs objets d'une manière explicite à l'aide d'un ou plusieurs objets simples.

$$x_1 , x_2 , x_3 , \ldots, x_n = f (y_1 , \ldots, y_p) \quad (1)$$

La fonction f peut avoir des significations très différentes suivant les cas : expression arithmétique, définition de "donnée" etc... On trouvera, ci-après, quelques exemples de définitions simples.

(2) x, y = donnée

(3) impr = 'MOYENNE' , MOY

(4) x = 2 * y - 3 * a/sqrt (z) + 5.3

Dans l'exemple (3) la liste des objets définis est composée d'un seul élément "impr" qui est en fait un objet externe correspondant à une ligne d'imprimante.

Un cas particulier fondamental de ce type de définition est la définition conditionnelle où la fonction f est définie d'une manière conditionnelle. Par exemple la définition de la densité d'une variable aléatoire X qui suit une loi uniforme sur [a, b]

(5) densité = si x ε [a, b] alors 1/(b - a) sinon 0

La fonction f peut aussi être une fonction définie dans une autre partie de l'analyse. Par exemple :

(6) x = intégrale (a, b)

Mais dans certains cas, on ne peut résoudre les problèmes en utilisant uniquement des objets simples. De nombreux problèmes en effet, nécessitent l'utilisation d'objets plus élaborés : les listes.

Par exemple :

- impression des salaires d'une entreprise (7)
- impression d'une table des carrés (8)
- calcul d'une moyenne de 100 notes (9)

Ces considérations nous amènent à introduire des définitions itératives :

res : pour tous les employés faire imprimer-un-salaire (10)

res : pour i de 1 à 1000 faire impr = i, i^2 (11)

Dans le cas (10) le résultat n'a pu être explicité immédiatement. Le résultat à obtenir est une liste de salaires ; le "module" imprimer-un-salaire définit donc un des éléments de la liste. La définition des modules sera faite elle aussi de manière déductive. On sera donc amené à définir chaque module d'abord informellement dans un lexique, puis à formaliser cette définition. La définition informelle du premier module sera donnée par l'énoncé du problème.

lexique des identificateurs	définitions formelles	lexique des modules
résultat : liste de n couples (nom, salaire) n : nombre d'employés	résultat : répeter n fois SAL n = donnée	SAL : imprime 1nom et 1 salaire
	SAL	
nom : nom de l'employé salaire : de l'employé nh : nombre d'heures	impr = nom, salaire nom, nh = donnée salaire = nh x 15	

Fig. 11 Impression des salaires de n employés

Dans certains cas (par exemple (8) et (9)) les listes ainsi définies et utilisées sont des suites mathématiques. Certaines de ces suites sont définies de manière récurrente ; par exemple dans le problème (9) la somme des 100 notes est définie par :

$$\begin{cases} Somme_i = Somme_{i-1} + Note_i \\ Somme_0 = 0 \end{cases}$$

Il sera donc essentiel au niveau des définitions formelles des termes de suites récurrentes de pouvoir distinguer un terme (somme$_i$) du terme précédent (somme$_{i-1}$). Cette distinction permet de faire disparaître au niveau de l'analyse la notion de variable et donc de lui conserver son caractère statique.

terme initial	U_0	U_d
terme courant	U_i	U
terme précédent	U_{i-1}	$\overline{U}$
terme final	$U_{/n}$	U_f

Fig. 12 Notations utilisées pour les suites récurrentes

Si nous prenons l'exemple (9), l'analyse du problème sera la suivante :

-résultat : impression de la moyenne -moy : moyenne des 100 notes - Somme : suite { $Somme_{i-1}$ +N ; $Somme_0$ = 0	résultat = moy moy = $somme_f$ /100 $Somme_f$: INI ; répéter 100 fois T	- INI : définit $Somme_d$ - T : définit
	T	
- n : note lue	Somme = Somme + N N = donnée	
	INI	
	$Somme_d$ = 0	

Fig. 13 Impression de la moyenne de 100 notes

Dans de nombreux cas, le nombre d'éléments de la liste construite par une itération n'est pas connu à priori (calcul d'une racine carrée avec une précision 0.001, arrêt sur carte blanche, etc...). On voit donc apparaître ici, la nécessité d'avoir deux sortes de définitions itératives :

- les définitions itératives avec nombre connu d'itérations
- les définitions itératives avec condition d'arrêt.

Si le choix d'un formalisme pour la définition dans le premier cas ne pose pas de problème, dans le second cas, par contre, le choix n'est pas aussi simple.

Nous reviendrons sur ce choix dans le chapitre II.

1.4. *CONCLUSION*

On pourrait en guise de conclusion, essayer de voir si les objectifs définis dans le paragraphe 41 sont atteints.

a) fiabilité

Le principe même de la démarche, s'il est scrupuleusement respecté permet d'affirmer que la construction même des définitions en constitue une preuve [12].

b) apprentissage

Le fait même d'utiliser une méthode ne peut être que favorable à l'enseignement.
Il est faux d'affirmer que l'utilisation de la méthode de programmation déductive supprime toute initiative au programmeur en le guidant pas à pas. En effet, la démarche déductive ne fait que faciliter la tâche du programmeur, en séparant nettement les périodes où l'intelligence humaine est nécessaire des autres (voir aussi le chapitre IV et V).

c) rentabilité des équipes de programmeurs

Elle est directement liée à l'obtention rapide de programmes fiables.

d) modularité

La construction déductive avec l'obtention d'une arborescence de modules ne peut que favoriser le découpage logique des problèmes.

e) modification

La lisibilité des analyses, et la suppression des astuces inextricables, permet de mener à bien très facilement des modifications sur les analyses obtenues.

Il semble donc qu'une méthode de programmation déductive soit bien adaptée à la résolution des problèmes. Une telle méthode est utilisée depuis quelques années à Nancy. Elle est notamment utilisée dans l'enseignement depuis 3 ans pour des publics assez variés (DEUG, IUT, MIAGE, Grandes Ecoles) [14],[15]

- res 1 : liste des notes < 10 - res 2 : moyenne des notes - S (suite) somme des notes $\begin{cases} S_i = S_{i-1} + \text{note} \\ S_0 = 0 \end{cases}$ - N (suite) nombre de notes $\begin{cases} N_i = N_{i-1} + 1 \\ N_0 = 0 \end{cases}$ - Moy moyenne	1 3 2	Sf, Nf, res1 : INI ; jqa fin répéter ELEVE res 2 = 'moyenne',moy moy = Sf / Nf	- INI - ELEVE :- définit Si, N_i - imprime la note si elle est < 10
		ELEVE	
- note : élément de la suite note_i = donnée - note d'un élève	1 2 3 5 4	S = S + note N = N + 1 impr : si note < 10 alors impr = note fin = (note = 0) note = donnée	
		INI	
	1 2 3 4	Sd = 0 Nd = 0 Note_d = donnée fin_d = faux	

Fig. 14 Résolution du problème de la moyenne et de l'impression des notes inférieures à 10 par la méthode déductive (cf. aussi fig. 3 - 4 - 5).

II

snoopy, le langage de la méthode déductive

RESUME

Ce chapitre présente le langage SNOOPY.

Pour les définitions itératives, on expliquera les raisons qui ont fait choisir une forme plutôt qu'une autre. On expliquera aussi pourquoi les procédures ne sont pas implémentées à l'heure actuelle dans SNOOPY.

PLAN

2.1. *PRESENTATION*

Comme on l'a vu dans le chapitre précédent, la résolution d'un problème par la Méthode Déductive peut se décomposer en plusieurs étapes, les deux dernières étant :

- l'ordonnancement des définitions dans chaque module
- la traduction des définitions ordonnées dans un langage de programmation.

La réalisation de ces dernières étapes est automatisable. (cf. 1.4.1 c et d). Donc une analyse non ordonnée peut être considérée comme un programme et par là même, peut être compilée.

SNOOPY a été créé pour permettre d'écrire, de compiler, d'ordonner et de traduire des analyses déductives. Les principaux rôles de SNOOPY sont les suivants :

- vérifier la cohérence des définitions explicites et des lexiques
- ordonner les définitions à l'intérieur de chaque module, tout en vérifiant la consistance de l'ensemble des définitions d'un même module
- traduire ces définitions réordonnées dans un langage de programmation.

L'utilisation de SNOOPY dans l'enseignement de la programmation nous a conduit à une certaine souplesse dans sa conception :

- ordonnancement automatique ou non, partiel ou total
- type implicite éventuel
- choix entre divers langages objets pour la traduction

Ce chapitre est destiné à présenter le langage SNOOPY. Il ne doit pas être considéré comme un manuel d'utilisation de ce langage. On trouvera un tel manuel en annexe A. On trouvera, de plus, un exemple de programme en annexe B.

2.2. *ORGANISATION DES PROGRAMMES*

2.2.1. Remarques préliminaires

SNOOPY est un langage à mots réservés. La liste des mots réservés est donnée dans l'annexe A. D'autre part, les blancs sont des caractères significatifs : une unité syntaxique (identificateur, opérateur, etc...) ne peut contenir de caractères blancs.

Un programme SNOOPY est divisé en plusieurs modules. Chaque module sert à définir un ou plusieurs résultats. Chaque module est de plus divisé en trois parties, présentées de gauche à droite sur la feuille de programmation :

- le lexique des résultats intermédiaires introduits dans le module
- la table des définitions
- le lexique des identificateurs de modules où figurent les modules introduits.

La séparation entre deux modules est faite par une ligne spéciale indiquant le nom du nouveau module et commençant par le caractère *.

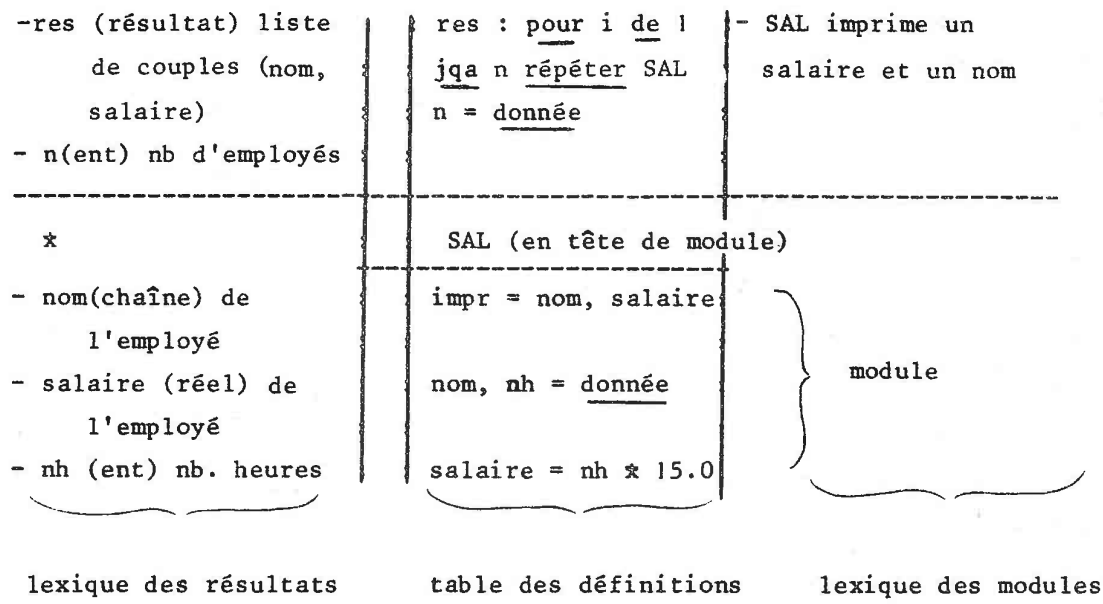


Fig. 1 Exemple de programme simple

2.2.2 Lexique des identificateurs

Tous les identificateurs employés dans l'analyse doivent être référencés dans le lexique lors de leur première utilisation. Une référence dans le lexique a la syntaxe suivante :

- ident (type) description

ident est l'identificateur
type est le type de l'identificateur ident
description permet de décrire d'une manière formelle ou informelle ident. Cette description peut éventuellement tenir sur plusieurs lignes du lexique. Cette partie doit être établie avec le plus grand soin. En effet, non seulement elle permet au programme de progresser dans la résolution en vérifiant la correction de ses définitions explicites, mais en plus, ce lexique peut servir de documentation du programme obtenu.

- DELTA (REEL) discriminant de l'équation

$$ax^2 + bx + c$$

- U (REEL) suite définissant $\sqrt{a}$

$$\begin{cases} U_i = \frac{1}{2}\left(U_{i-1} + \frac{a}{U_{i-1}}\right) \\ U_0 = 2 \end{cases}$$

- NOM (CHAINE) de l'élève

Fig. 2 Exemple de lexique des identificateurs

< référence dans le lexique >::= -<identificateur simple> <type >
< description >

< type >::= (<type simple>)|(<type simple >TAB< bornes>)|(TAB < bornes>)|^

<type simple >::= ENT | CAR | BOOL | REEL | CHAINE | RESULTAT

<bornes > ::= < entier sans signe> |<entier sans signe >, <bornes>

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

<description > ::= <caractère> |<caractère> <description >

Fig. 3 Description d'une référence dans le lexique des résultats

Les différents types utilisables sont décrits dans le paragraphe 2.3.

Le type peut éventuellement être omis dans la déclaration si celui-ci correspond au type implicite qui a été choisi pour le programme (cf. §2.2.5).

2.2.3. La table des définitions

Dans cette partie figurent les définitions explicites obtenues à partir des énoncés du lexique.

Le programmeur peut indiquer, s'il le désire, devant chaque définition un numéro d'ordre correspondant à l'ordre d'exécution de la définition lors de l'exécution du programme. Ces numéros d'ordonnancement seront pris en compte si le programmeur a choisi l'option d'ordonnancement manuel (cf. §2.2.5) sinon ils pourront être remplacés par tout caractère autre que blanc. Une définition peut tenir sur une ou plusieurs lignes, toutefois, une unité syntaxique ne devra jamais être coupée par une fin de ligne.

```
| 2 | x = a + b *∂x
| 3 | y : si  x > 0 alors y = a sinon calcul
| 1 | a, b  : si ∂x = 0 alors a, b  = donnée
                          sinon module
```

Fig. 4 Exemple de table des définitions

2.2.4. Le lexique des modules

Dans ce second lexique sont décrits les divers modules utilisés dans l'analyse. Ces modules, rappelons-le, sont utilisés pour définir les résultats dont la définition ne peut être explicitée simplement (résultat d'une récurrence, d'un choix, etc...). Tous les modules utilisés dans le programme doivent être référencés dans ce lexique. La forme d'une référence est la suivante :

- ident : description

ident est l'identificateur du module

description donne une définition informelle mais précise des fonctions du module ident. De même que pour le lexique des résultats, cette description doit être aussi précise que possible, et ce afin de documenter le programme. Cette description peut tenir sur une ou plusieurs lignes.

- INI : initialisation du calcul de la masse salariale
- CALCAL : définit y quand x est négatif ou nul

Fig. 5 Exemple de lexique des modules

2.2.5. Options de programmation

Lors de l'écriture d'un programme, l'utilisateur peut, s'il le désire, utiliser certaines options destinées à lui faciliter la tâche. Certaines de ces options ont été introduites notamment pour faciliter l'utilisation de SNOOPY dans l'enseignement. Les diverses options sont :

- choix du langage objet (actuellement Pascal et FORTRAN)
- ordonnancement automatique ou non
- listage optionnel du programme objet
- choix d'un type implicite (entier ou réel)
- choix des zones, de la feuille de programmation, utilisées pour la représentation des diverses parties de l'analyse (lexiques, définitions).

Ces options sont données par le programmeur en tête de programme sur une carte paramètre (cf. annexe A.9.2).

2.2.6. Portée des identificateurs de résultats

Un identificateur peut être utilisé, dans l'analyse, dans le module où il a été référencé dans le lexique, ainsi que dans tous les modules qui sont utilisés dans ce module.

De même, tout identificateur référencé dans le lexique doit être défini dans le module correspondant (sauf si sa définition est remontée dans une initialisation).

Une récapitulation complète des régles d'utilisation des identificateurs est donnée dans l'annexe A paragraphe 7.

2.3. *LES DIVERS OBJETS UTILISES*

2.3.1. Types simples

Actuellement, SNOOPY permet l'utilisation de six types simples. Ces six types sont les suivants :

a) Entier (ENT)

b) Réel (REEL)

c) Booléen (BOOL). Les identificateurs de ce type peuvent prendre deux valeurs : *VRAI* ou *FAUX*

d) Caractère (CAR). Les divers caractères utilisables sont donnés dans l'annexe A.

< constante caractère > :: = ' < caractère différent de ' > ' | ''''

Fig. 6 Syntaxe de la constante caractère

e) Chaîne (CHAINE). Une chaîne de caractères est une suite de caractères comprise entre deux apostrophes.

< constante chaîne > :: = '< suite de caractères > '

< suite de caractères > :: = < caractère de chaîne><suite> |<caractère de chaîne>

< caractère de chaîne > :: = < caractère différent de ' > |''

Fig. 7 Syntaxe de la constante chaîne

f) Résultat (RESULTAT). Ce type permet de donner un nom aux résultats externes en rendant ainsi l'analyse plus lisible.

Lorsque l'analyse doit être traduite dans un langage donné, les types utilisables dans le programme SNOOPY sont ceux qui sont autorisés dans le langage objet utilisé. Un seul type fait exception à cette régle, c'est le type RESULTAT. En effet, les identificateurs de ce type n'apparaissent pas dans la traduction.

- K (ENT) RANG DU NOMBRE DANS LE VECTEUR V
- LISTE (RESULTAT) LISTE DE LIGNE AVEC SUR CHAQUE LIGNE UN NOM D'ELEVE ET SA MOYENNE

Fig. 8 Exemple de déclaration de types simples

2.3.2. Type Tableau

On peut aussi utiliser en SNOOPY des objets structurés de type tableau. Les tableaux peuvent avoir une ou deux dimensions. Le type des éléments d'un tableau est l'un des six types simples énumérés dans le paragraphe 2.3.1.

<tableau >:: = <identificateur simple>

<élément de tableau >:: = <identificateur simple> <indiçage >

<indiçage >:: = [indices] | (indices)

<indices >:: = <expression entière >|< expression entière >, <expression entière >

Fig. 9 Représentation des tableaux

2.4. *LES DEFINITIONS EXPLICITES*

2.4.1. Présentation

Dans le chapitre précédent, nous avons vu que suivant l'objet à définir, on pouvait distinguer deux types de définitions :

- les définitions non itératives pour les objets simples
- les définitions itératives pour les listes

Nous avons d'autre part distingué un cas particulier important des définitions non itératives : les définitions conditionnelles. Dans ce paragraphe, nous avons séparé les définitions en trois groupes :

- les définitions simples
- les définitions conditionnelles
- les définitions itératives

2.4.2. Les définitions simples

2.4.2.1. Définition explicite simple

Elle a la forme suivante :

a = e

a est l'objet défini, c'est un identificateur simple ou indicé

e est l'expression par laquelle est défini explicitement a

La syntaxe de l'expression est très libre ; elle dépend du langage objet utilisé et, par la même, du type de problème traité. On peut donc remarquer que, lorsqu'aucune traduction n'est demandée, la syntaxe de ces expressions est très libre . La seule contrainte syntaxique est que les opérateurs ne doivent pas être formés à l'aide de lettres (par exemple : .LT. en FORTRAN).

Il existe toutefois, une exception à cette règle ; en effet, trois opérateurs logiques ($\neg$, | et $\wedge$) peuvent être remplacés par trois identificateurs réservés (NON, OU et ET). Cette particularité a été introduite pour faciliter l'utilisation de SNOOPY dans l'enseignement. En effet, les trois opérateurs NON, ET et OU sont plus faciles à manipuler pour des débutants que leurs correspondants symboliques.

```
X = 3 * A - 2 * B * B

Y = ( A ET X <= 2 ) OU B

Y = ( A ∧ X <= 2 ) | B

PHRASE = 'LE'. NOM1. VERBE. 'LE'. NOM2
```

Fig. 10 Exemples de définitions explicites simples

<définition explicite simple> ::= <identificateur > = <expression>
<identificateur > ::= <identificateur simple> |<élément de tableau>

Fig. 11 Description de la définition explicite simple

2.4.2.2. Définition de résultat externe

Elle a la forme suivante :

$r = l_1, \ldots, l_n$

r est soit un identificateur de type résultat, soit l'identificateur standard IMPR. L'utilisation de IMPR évite d'avoir à référencer le résultat externe défini dans le lexique.

$l_1, \ldots, l_n$ est la liste de sortie où l_i est soit un identificateur simple ou indicé, soit une chaîne de caractères.

```
LIGNE = 'SALAIRE' , SAL , 'F'
IMPR = T (5, 3 * I), X, Y, 'RESULTAT'
```

Fig. 12 Exemples de définitions de résultats externes

<définition de résultat externe> ::= <partie gauche >= <partie droite>

<partie gauche >::= IMPR | <identificateur>

<partie droite > ::= <élément de liste externe> | <élément de liste externe> , <partie droite>

<élément de liste externe> ::= <identificateur> | <constante chaîne>

Fig. 13 Description de la définition de résultat externe

2.4.2.3. Définition par lecture

Elle a la forme suivante

$l_1, \ldots, l_n$ = DONNEE

$l_1, \ldots, l_n$ est la liste d'entrée, où l_1 est un identificateur simple ou indicé

```
A, B, C [3] = DONNEE
```

Fig. 14 Exemple de définition par lecture

<définition par donnée >::= <liste d'identificateurs > = DONNEE

<liste d'identificateurs >::=< identificateur >|<identificateur > ,

<liste d'identificateurs>

Fig. 15 Description de la définition par lecture

2.4.3. Définition Conditionnelle

Elle a la forme suivante

x_1 , ..., x_n : INI ; si cond 1 alors CAS 1 ;

cond 2 alors CAS 2 ;

.....

cond n alors CAS n

sinon T

x_1 , ..., x_n est la liste des objets définis où x_i est un identificateur simple ou indicé.

INI est un module ou une définition simple permettant de remonter certaines définitions communes aux diverses branches de la conditionnelle . Ce module peut être omis.

CAS 1, .., CAS n, T sont des modules ou des définitions simples où sont définis x_1 ,..., x_n.

Cond 1,..., Cond n sont des expressions booléennes.

Lors de l'exécution INI est toujours exécuté. Les diverses expressions booléennes sont ensuite évaluées jusqu'à ce que l'on en trouve une dont la valeur est VRAI. Le module correspondant est alors exécuté. Si aucune des expressions booléennes n'a la valeur VRAI, le module T est alors exécuté. La partie sinon T peut être omise.

```
RES      :  SI    DELTA > 0     ALORS   REEL  2 ;
                  DELTA = 0     ALORS   REEL  1
                                SINON   IMPR  = 'IMAGINAIRE'

Z  :  IN ; SI     R  =  1       ALORS   Z  =  2 ;
                  R  =  2       ALORS   Z  =  6 ;
                  R  =  3       ALORS   CALCUL

X, Y  :  SI  A  > 0  ALORS  T1  SINON  T2

IMPR  :  SI  SALAIRE  <  SMIC  ALORS  IMPR  =  NOM , SALAIRE
```

Fig. 16 Exemplesde définitions conditionnelles

<définitions conditionnelles > ::= <entête> SI < partie alors> <partie sinon >

<entête >::= < but > : < initialisation >

<initialisation >::= <définition interne> ; | ∧

<but> ::= IMPR | <liste d'identificateurs >

<définition interne >::= <définition explicite simple> | <définition par lecture>|
<définition de résultats>|< identificateur simple >

<partie alors >::= <clause> |<clause >; <partie alors >

<clause > ::= <expression booléenne> ALORS <définition interne >

<partie sinon >::= ∧ |SINON< définition interne>

Fig. 17 Description de la définition conditionnelle

2.4.4. Définition itérative

2.4.4.1. Définition POUR

Elle a la forme suivante :

x_1 , ... , x_n : INI ; pour i de p jqa q répéter MOD

x_1 ,..., x_n sont les suites définies par l'itération ;

x_i est un identificateur simple ou indicé

i est l'indice de l'itération

p et q sont des expressions à résultats entiers.

INI est le module d'initialisation de l'itération. Il sert à "remonter" certaines définitions qui avaient été placées tout d'abord dans le corps de l'itération (cf. fig. 19) et à initialiser les suites récurrentes. Ce module peut être remplacé par une définition simple ou être omis.

MOD est un module ou une définition simple définissant x_1 ,..., x_n

Lors de l'exécution, INI est toujours exécuté. Ensuite p et q sont évalués. Si p est inférieure ou égale à q, le module MOD est alors exécuté q - p + 1 fois, à la jème fois, i à la valeur p + j - 1. Si p est inférieure à q, le module MOD n'est pas exécuté.

<définition POUR> ::= <entête> POUR <identificateur simple > DE

<expression entière >JQA <expression entière> REPETER <définition interne>

Fig. 18 Description de l'itération POUR

- liste (résultat) des couples (nom,salaire)	2	liste, s : INI ; pour i de 1 jqa n répéter T	- T imprime un nom et un salaire et redéfinit la masse salariale
	3	MS = 'MASSE = ' , s	
- MS (résultat) ligne où figure la masse salariale de l'usine	1	n = donnée	
			- INI : initialise s
- S (réel) suite $\begin{cases} s_i = s_{i-1} + sal \\ s_0 = 0 \end{cases}$			
- n (ent) nbre d'employés			
		T	
- nom (chaîne) de l'employé	5	impr = nom, sal	
- sal (réel) salaire (nh x sh)	1	nom = donnée	
	3	sal = nh * sh	
- nh (ent) nbr. d'heure	2	nh = donnée $ nom	
- sh (réel) salaire horaire (dans INI)	4	s = @ s + sal	
		INI	
	1	s = 0	
	2	sh = 15.0	

Fig. 19 Exemple d'utilisation d'un POUR

2.4.4.2. Définition JQA

2.4.4.2.1. Description de la définition

Elle a la forme suivante :

$x_1, \ldots, x_n$: INI ; jqa fin répéter T

$x_1, \ldots, x_n$ est la liste des objets à définir où x_i est un identificateur simple ou indicé.

INI : est le module d'initialisation dont les fonctions sont les mêmes que dans la définition POUR. Une des suites qui sera notamment initialisée dans ce module est la suite "fin".

fin est un identificateur booléen permettant lors de l'exécution de contrôler l'arrêt de l'itération.

T est un module où sont définis x_1 , ... x_n et fin.

Lors de l'exécution, l'initialisation est toujours exécutée. La valeur de fin est ensuite évaluée et le module T est itéré jusqu'à ce que fin prenne la valeur VRAI. Donc, si fin a la valeur VRAI après l'initialisation, T n'est pas exécuté.

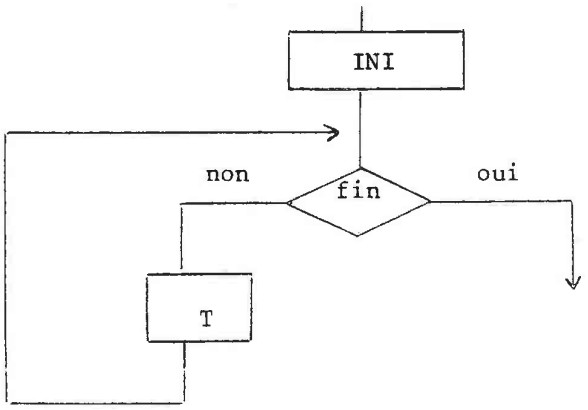


Fig. 20 Schéma d'exécution de la définition JQA

-sol (résultat) valeur de X et de $\sqrt{X}$ - X (réel) valeur dont on cherche la racine - RAC (réel) suite $\begin{cases} RAC_i = \frac{1}{2}(RAC_{i-1} + \frac{X}{RAC_{i-1}}) \\ RAC_0 = 1 \end{cases}$	3 1 2	sol = 'RACINE DE', X, RAC X = donnée RAC : I ; jqa arrêt répéter CAL $ X	- CAL calcul RAC_i et arrêt (arrêt qd $\lvert X^2-RAC\rvert < \epsilon$) - Initialise RAC et arrêt
		CAL	
- X2 (réel) X au carré (ds I) - EPS (réel) précision (ds I)	1 2	RAC = (∂ RAC + X/∂ RAC)/2 arrêt = ABS (RAC-X2)< EPS	
		I	
	1 2 3 4	RAC = 1 arrêt = faux EPS = donnée X2 = X * X	

Fig. 21 Exemple d'utilisation de la définition JQA

<définition JQA> ::= <entête >JQA <identificateur simple >REPETER

<définition interne>

Fig. 22 Description de la définition JQA

2.4.4.2.2. *Justification du choix de la forme de la définition*

Depuis sa création, la Méthode de Programmation Déductive a vu la forme de ses itérations connaître de nombreux changements. Ces diverses modifications ont été imposées par le malaise que créait tant dans l'enseignement que dans la pratique, l'utilisation d'une forme plutôt qu'une autre.

Ce malaise est surtout dû à la difficulté de placer la définition de l'arrêt et au moment de l'exécution son évaluation à l'intérieur de l'itération. Cette difficulté a déjà été ressentie lors de l'utilisation d'autres méthodes, notamment la programmation structurée où elle est directement liée à l'absence de GO TO [16].

Nous allons présenter les diverses formes qui ont été déjà utilisées dans la Méthode Déductive, en donnant pour chacune, ses avantages et ses inconvénients [17] .

a) INI ; répéter T : le module T contient une définition de l'arrêt. Lors de l'exécution, l'évaluation de l'arrêt est faite en fin de boucle. Malgré sa simplicité, cette forme a un gros défaut : le module T est toujours exécuté au moins une fois.

b) INI ; répéter (A ; T) : le module T sert à définir les objets définis par l'itération et le module A définit l'arrêt. Lors de l'exécution, le module T n'est exécuté que lorsque la condition d'arrêt calculée dans A a la valeur FAUX. Cette forme quoique plus agréable, présente cependant quelques inconvénients :
   - dans certains cas, le module T est vide (lorsque la définition de l'arrêt nécessite la définition des objets définis par l'itération).
   - l'utilisation de deux modules est lourde

c) utilisation simultanée des solutions a et b :

Cette solution qui permet de définir l'arrêt "au meilleur endroit" dans tous les cas pose des problèmes au programmeur, quant au choix de la forme qu'il doit utiliser pour une définition donnée (notamment lors de l'utilisation de la Méthode Déductive dans l'enseignement).

d) INI ; jqa fin répéter T : c'est la forme présentée dans le paragraphe précédent. Elle présente un avantage certain : l'utilisation de l'arrêt est banalisée(fin est un identificateur comme les autres), d'autre part, il ne se pose plus de problème de choix, comme dans les solutions b et c. Un inconvénient toutefois, certains problèmes donnent lieu à des initialisations assez lourdes où sont reprises certaines définitions du module itéré. Ce cas se produit lorsque le traitement du premier élément d'une liste est particularisé (par exemple : traitement d'une liste avec arrêt sur un élément donné, cet élément n'étant pas traité, le premier élément de la liste peut de plus vérifier la condition d'arrêt). Cet inconvénient est largement compensé par la simplicité d'utilisation de cette définition.

e) INI ; répéter T jqa fin : cette solution est identique à la précédente, à la seule différence, que l'identificateur d'arrêt est évalué lors de l'exécution, après les calculs du module T. Cette solution évite l'initialisation de l'identificateur d'arrêt et l'alourdissement du module d'initialisation. Par contre, comme dans la solution a le module T est toujours exécuté au moins une fois.

La solution d qui a été retenue est caractérisée, en résumé, par deux avantages importants :

- banalisation de l'arrêt
- simplicité d'utilisation

4.4.3. Remarque sur l'ordonnancement des définitions

Dans certains modules, la mise en oeuvre d'un ordonnancement automatique peut donner lieu à des ambiguïtés, ordre des lectures par exemple. Pour pallier à cette difficulté, certaines définitions pourront être suivies de "listes d'ordonnancement" qui indiqueront, pour une définition, les résultats qui devront être calculés avant celle-ci, lors de l'exécution.

```
a : si x > 0 alors a = b sinon T $ c

c = b * b

b = donnée
```

Fig. 22 Liste d'ordonnancement : c doit être calculé avant l'exécution de la conditionnelle car le module T utilise celui-ci.

<liste d'ordonnancement > ::= $< liste d'identificateurs >

Fig. 23 Description de la liste d'ordonnancement

2.5. *LES SOUS-PROGRAMMES ET LES FONCTIONS*

A l'heure actuelle SNOOPY n'autorise pas l'utilisation et l'écriture de sous-programmes élaborés par le programmeur. En effet, seules quelques fonctions standards sont utilisées (cf. annexe A). Cette lacune n'est pas due à un problème posé par l'implémentation des fonctions et des sous-programmes, mais à l'absence d'une syntaxe satisfaisante pour la définition de ceux-ci, dans la Méthode de Programmation Déductive, au moment où l'implémentation de SNOOPY a été réalisée.

Depuis une forme satisfaisante pour les procédures a été trouvée, mais celle-ci n'a pas encore été implémentée.

TTT

le compilateur snoopy

RESUME

Ce chapitre présente le fonctionnement et la structure du compilateur SNOOPY [18, 19]

PLAN

3.1. *LES FONCTIONS DU COMPILATEUR*

3.1.1. Vérifications syntaxiques

Un des rôles les plus importants du compilateur SNOOPY est de vérifier syntaxiquement l'analyse. Cette vérification doit s'effectuer à deux niveaux bien distincts :

a) Vérification lexicographique des diverses parties de l'analyse (lexiques et définitions)

b) Vérification liée à l'utilisation correcte de la méthode déductive. Le nombre des contrôles que doit réaliser le compilateur dans ce domaine est assez important (cf. annexe A7). Il faut en effet vérifier que :
   - tous les résultats du lexique ont été bien définis ;
   - chaque module a bien servi à définir les résultats qu'il était censé définir ;
   - tous les modules utilisés figurent dans le lexique des modules et réciproquement ;
   - les identificateurs de résultat n'ont été utilisés que là où ils pouvaient l'être (cf. 2.2.6.) ;
   - les résultats définis par récurrence ont été bien initialisés (cf. 2.4.4.).

3.1.2. Ordonnancement des définitions explicites

Après avoir été analysé syntaxiquement, chaque module doit être correctement réordonné si cela est possible et si l'utilisateur désire utiliser l'option d'ordonnancement automatique (cf. 2.2.5.).

3.1.3. Traduction de l'analyse obtenue

Lorsque la vérification et l'ordonnancement de l'analyse se sont terminés avec succès, il reste encore à la traduire dans le langage évolué

choisi par le programmeur. A l'heure actuelle, deux langages-objets sont disponibles : FORTRAN et Pascal.

3.1.4. Fourniture de renseignements concernant l'analyse

Le dernier rôle du compilateur, rôle à ne pas négliger si l'on veut que le compilateur soit facilement utilisable, est de fournir à l'utilisateur un certain nombre de renseignements concernant l'analyse. Ces informations sont les suivantes :

a) Ordre des définitions à l'intérieur de chaque module ou, si le réordonnancement n'a pu être mené à bien, message indiquant cet échec.
b) Liste des symboles utilisés dans l'analyse, c'est-à-dire :
   - liste des résultats utilisés, avec, pour chacun, son type et son encombrement en mémoire.
   - liste des modules utilisés, avec, pour chacun, la catégorie à laquelle il appartient (itération, etc...), son niveau d'imbrication (le module principal a le niveau 0) et éventuellement le module d'initialisation qui lui correspond.
c) Message d'erreur. Ces messages d'erreur doivent être aussi compréhensibles que possible et doivent permettre une localisation aisée de chaque erreur (spécification du numéro de la ligne ou de la définition où a été détectée l'erreur). En plus des erreurs lexicographiques, ces messages doivent signaler toutes les entorses faites aux règles exposées plus haut (cf. 3.1.1.b).
d) Impression du jeu d'essai. Ce renseignement n'est malheureusement pas fourni par la plupart des langages existants. Il semble pourtant que cette information permette de vérifier la validité du jeu d'essai et aussi de pouvoir suivre plus aisément le déroulement du programme.
e) Impression du programme-objet

On trouvera en annexe B un exemple de programme SNOOPY.

3.2. *LA STRUCTURE DU COMPILATEUR*

3.2.1. Les tâches assumées par le compilateur

Le paragraphe précédent nous a amenés à distinguer trois tâches à l'intérieur du compilateur (vérification, ordonnancement et traduction). Nous examinerons successivement ces trois tâches dans le paragraphe 3.3. Nous allons tout d'abord présenter les structures de données utilisées par le compilateur et leur représentation en mémoire.

3.2.2. Structures de données utilisées par le compilateur

3.2.2.1. Le lexique des résultats

En plus du nom et du type de chaque identificateur de ré-résultats (intermédiaires ou finaux), le lexique doit contenir un certain nombre de résultats (intermédiaires ou binals), de renseignements permettant de vérifier la cohérence de l'analyse. Ces renseignements sont les suivants :

- Ensemble des modules où peut être utilisé le résultat
- Connaissance des modules où le résultat doit, ou ne doit pas, être défini et, par là-même, des modules où il a déjà été défini. En effet, un résultat
  - + doit être défini dans le module où il apparaît pour la première fois dans le lexique.
  - + doit être défini dans le module itératif d'une définition itérative lorsqu'il apparaît dans l'en-tête de cette définition.
  - + doit être défini dans au moins un des modules conditionnels d'une définition conditionnelle lorsqu'il apparaît dans l'en-tête de cette définition.
  - + peut être défini dans le module d'initialisation d'une définition où il apparaît dans l'en-tête.
- Connaissance du module définissant le résultat si celui-ci est défini par récurrence.

3.2.2.2. Le lexique des modules

En plus du nom de chaque module, le lexique doit contenir les renseignements suivants :

- Type du module (itératif, conditionnel, initialisation).
- Initialisation correspondante (éventuellement).
- Ensemble des modules d'initialisation où peuvent être remontées les définitions de ce module.
- Etat de définition (déjà défini ou pas) et d'utilisation (déjà utilisé ou pas) du module.
- Connaissance du module où a été utilisé le module.
- Niveau du module.

3.2.2.3. Les définitions explicites

Les renseignements qui doivent être conservés pour les définitions explicites sont les suivants :

- Type de la définition (itération, lecture, etc...)
- Modules utilisés par la définition avec leurs rôles (ce rôle est fonction de la position de chaque module à l'intérieur de la définition)
- Expressions arithmétiques et booléennes utilisées dans la définition.
- Identificateurs de résultat utilisés dans la définition (identificateur d'arrêt du JQA et indice du POUR).
- Liste d'identificateurs pour les entrées-sorties.
- Résultats définis par la définition. Ce renseignement n'est conservé que durant l'analyse de la définition.

Certains autres renseignements sont nécessaires, par exemple pour l'ordonnancement automatique. Nous y reviendrons plus tard.

3.2.2.4. Autres informations nécessaires

Certains autres renseignements doivent aussi être conservés. Par exemple, les messages d'erreurs qui ne sont édités qu'en fin d'analyse ou encore l'ordre des définitions de chaque module après l'ordonnancement automatique. Ces renseignements sont moins importants pour la compréhension du fonctionnement du compilateur et ne seront donc pas détaillés.

3.2.3. Représentation des structures de données

3.2.3.1. Le lexique des résultats

Pour représenter le lexique des résultats et le lexique des modules, on utilise des tableaux. Le langage utilisé est Pascal (cf. 3.4.1.). Ce langage permet d'avoir des types d'élément de tableau très variés (ensemble, alphanumérique, etc...). Pascal n'autorise pas les tableaux à bornes variables et la dimension de ceux-ci a donc été arbitrairement fixée à 64. Chaque résultat et chaque module est donc identifié par le rang où il apparaît dans la table des symboles. Lors de la rencontre d'un symbole déjà utilisé, la recherche du rang est faite de manière associative.
Soit X le ième résultat de la table des symboles :

SYMBOLE [I] contient l'identificateur X
TYPTAB [I] contient le type de X
OK [I] est l'ensemble des modules où peut être utilisé X
ADEFINIR [I, J] donne l'état de définition de X dans le jème module. Cet état est l'un des quatre suivants :

- NON X n'a pas à être défini dans le module n° j
- DEJA X a déjà été défini dans le module n° j
- DOIT X doit être défini dans le module n° j
- PEUT X peut être défini dans le module n° j

Ce quatrième état est utilisé dans trois cas :

+ quand le module j est une initialisation.
+ quand le module j est une branche d'une conditionnelle. Dans ce cas, en effet, X doit être défini dans au moins une branche de la conditionnelle. En effet, par mesure de simplification, SNOOPY autorise à omettre les définitions méthodologiquement obligatoires mais qui n'apportent rien à la définition du problème dans les diverses branches des définitions conditionnelles (cf. fig.

min, max : si x < ∂ min alors { min = x ; max = ∂ max
x > ∂ max alors { max = x ; min = ∂ min

1 a forme rigoureuse de la méthode

```
MIN, MAX : SI  X < @ MIN ALORS MIN = X ;
               X > @ MAX ALORS MAX = X
               1 b syntaxe SNOOPY autorisée
```

fig. 1 : exemple de définition conditionnelle simplifiée

+ quand X est un résultat de type tableau. En effet, un tableau peut être défini plusieurs fois dans le même module (différents éléments du tableau).

TABIN [I] indique les modules d'initialisation où peut être remontée la définition de X.

AREMONTER [I] indique si X doit être ou non remonté dans un module d'initialisation.

INIREC [I] indique, dans le cas où X est défini par récurrence, s'il a été ou non correctement initialisé.

ITER [I] indique, dans le cas où X est défini par une itération, par quel module il est défini.

3.2.3.2. Le lexique des modules

Soit T le jème module de l'analyse :

TABLE [J] contient l'identificateur T.

J ∈ INITAB indique si T est ou non une initialisation.

J ∈ SITAB indique si T est ou non la branche d'une conditionnelle.

ITERATION [J] indique si T est ou non un module itératif.

CORINI [J] indique éventuellement l'initialisation qui correspond à T.

INITA [J] est l'ensemble des modules d'initialisation où peuvent être remontées les définitions de T.

TABLUTILE [J] indique si T a déjà été utilisé.

DEJADEFINI [J] indique si T a déjà été défini.

SURTAB [J] indique le module où T a été utilisé.

NIV [J] donne le niveau de T.

NBDEF [J] donne le nombre de définitions de T.

3.2.3.3. Les définitions explicites

Chaque définition conduit à un enregistrement qui est généré au moment de l'analyse de cette définition. Pour accéder à ces définitions,

on utilise un tableau du pointeur LISTRUCT. LISTRUCT [I,J] est donc un pointeur sur l'enregistrement représentant la Ième définition du Jème module. En plus du type de la définition (POUR, JQA, SI, ENTREE, SORTIE, DEF. EXP.) l'enregistrement contient les renseignements liés à la définition. Suivant le type de la définition, ces renseignements ne sont pas les mêmes et donc l'enregistrement généré n'aura pas la même structure interne (cf. fig. 2).

JQA	numéro du module d'initialisation	numéro de l'identificateur d'arrêt	numéro du module itéré

ENTREE	liste des résultats définis

fig. 2 : exemple d'enregistrements générés pour les définitions JQA et ENTREE(lecture)

Lors de la compilation de la définition, le tableau VARNUM contiendra les numéros des identificateurs de résultats définis par la définition.

3.2.4. Les fichiers utilisés par le compilateur

Pour accomplir son travail, le compilateur utilise cinq fichiers de caractères :

DEFINITION permet de conserver les définitions au cours de l'analyse d'un module.

MESSAGE permet de conserver les divers messages relatifs aux erreurs détectées en cours de compilation.

DONNEE contient le jeu d'essai.

SOURCE contient la traduction de l'analyse correcte dans le langage évolué choisi par l'utilisateur.

ERRFILE est le fichier contenant la liste des messages d'erreurs.

3.3. *LA VERIFICATION DE L'ANALYSE*

3.3.1. Principe général de la compilation

Du fait de la structure des programmes SNOOPY, la compilation ligne par ligne aurait posé de gros problèmes. En effet, à part les lignes commentaires et les en-têtes de module, une même ligne est composée de trois parties distinctes (cf. 2.2.1.). De plus, chacune de ces parties n'est généralement qu'un morceau d'une entité syntaxique.

```
-X(ENT) RACINE DE |08| X:SI A > 0 ALORS T  | - T DEFINIT X
ENTIERE                        SINON XX    | QUAND A > 0
```

lexique gauche — définitions — lexique droit

fig. 3 : les trois parties composant une ligne.

Chaque module est donc analysé en deux temps :

a) Analyse ligne par ligne au cours de laquelle s'effectue la compilation des lexiques (cf. 3.3.2.). Pendant cette analyse, les définitions sont rangées une par une dans un fichier où les blancs non significatifs sont supprimés. Cette analyse permet aussi de vérifier éventuellement la cohérence des numéros d'ordonnancement (le numéro d'ordonnancement doit être un entier positif inférieur ou égal au nombre de définitions du module et différent des numéros déjà utilisés dans le module).

b) Analyse des définitions une par une. Pendant cette analyse, les définitions sont compilées (cf. 3.3.3.) et une vérification est effectuée sur la cohérence entre les identificateurs utilisés et les lexiques.

Cette méthode de compilation, tout en facilitant l'analyse, évite aussi les problèmes posés par les références en avant. En effet, un identificateur ne peut être utilisé sans avoir été auparavant référencé dans un lexique. L'absence de référence en avant facilite la vérification de la cohérence entre les lexiques et les définitions.

On trouvera en annexe D.1. l'algorithme déductif global de la compilation d'un programme.

3.3.2. Compilation des lexiques

3.3.2.1. Le lexique des identificateurs

Après avoir reconnu un nouveau symbole correct, quatre opérations sont effectuées :

a) Le symbole est rangé dans SYMBOLE.
b) Le type est compilé et, s'il est correct, rangé dans TYPTAB.
c) Le tableau d'état de définition des résultats est modifié (ADEFINIR). Le résultat doit être défini dans le module en cours d'analyse.
d) L'ensemble des modules où peut être utilisé le résultat (élément du tableau OK) est initialisé avec le module en cours d'analyse et, éventuellement, son initialisation.

3.3.2.2. Le lexique des modules

Après avoir reconnu un nouvel identificateur de module, trois opérations sont effectuées :

a) Le symbole est rangé dans TABLE.
b) On indique que le module doit être défini à l'aide du tableau booléen DEJADEFINI.
c) On indique que le module doit être utilisé dans le module en cours d'analyse (TABLUTILE).

3.3.3. Compilation des définitions d'un module

3.3.3.1. Présentation

La compilation d'une définition se fait en trois étapes :

- compilation de l'en-tête de la définition et reconnaissance du type de la définition ;
- compilation du corps de la définition ;
- vérification de la cohérence de la définition.

3.3.3.2. Compilation de l'en-tête

Le compilateur reconnaît tout d'abord la liste des résultats. Ces résultats sont rangés dans le tableau VARNUM. A la suite de la liste de résultats, trois cas peuvent se produire :

- Le compilateur rencontre le caractère ':'. Cans ce cas, il recherche à reconnaître une initialisation s'il y en a une, puis suivant le mot réservé qu'il rencontre, il reconnaît le type de définition auquel il a affaire (SI, POUR, JQA).
- Le compilateur reconnaît la chaîne '=DONNEE'. Dans ce cas, il reconnaît la définition par lecture.
- Le compilateur reconnaît le caractère '=' non suivi de 'DONNEE'. Dans ce cas, il a affaire à une définition explicite simple ou à une écriture.

Dans ces trois cas, le compilateur génère un nouvel enregistrement pointé par un élément de LISTRUCT. Cet enregistrement a la structure correspondant au type de la définition reconnue.

3.3.3.3. Compilation du corps de la définition

3.3.3.3.1. Définition simple

Dans le cas d'une impression ou d'une lecture, la compilation ne pose pas de problème. Dans le cas d'une définition explicite simple, le compilateur ne fait pas une analyse profonde de la syntaxe de l'expression mais il se contente de reconnaître les identificateurs afin de pouvoir vérifier s'ils peuvent être utilisés (tableau OK). Cette absence d'analyse des expressions est due à la grande liberté autorisée dans la syntaxe de celles-ci (cf. 2.4.2.1.).

3.3.3.3.2. Définition structurée

Le compilateur vérifie la syntaxe de la définition et à chaque fois qu'il rencontre un module, il effectue cinq opérations. Soit T le module rencontré, i son numéro d'ordre et j le numéro du module en cours d'analyse. Les opérations effectuées sont les suivantes :

a) Vérification que T n'a pas encore été utilisé (TABLUTILE [I])
b) Si la définition comporte une initialisation, son numéro d'ordre est mis dans CORINI [I] .
c) L'ensemble des initialisations où peuvent être remontées les définitions de T est construit :
   INITA [I] := INITA [J] + CORINI [I]
d) Le type du module est mémorisé (SITAB et ITERATION).
e) Le niveau de T est calculé : NIV [I] := NIV [J] + 1

Dans certains cas, un module peut être remplacé par une définition simple. Le compilateur génère alors un module supplémentaire ne contenant qu'une définition et celle-ci est compilée immédiatement.

3.3.3.4. Vérification de la cohérence de la définition

Lorsque la définition est complètement analysée, des vérifications de cohérence et des modifications sont effectuées à l'aide du tableau VARNUM qui, rappelons-le, contient les résultats définis par la définition :

a) Les résultats doivent être définis dans les divers modules de la définition à l'exception de l'initialisation où leur définition est facultative. Modification du tableau d'état ADEFINIR.
b) Les résultats doivent avoir le droit d'être définis dans le module en cours d'analyse (ADEFINIR).
c) Les résultats ne doivent plus être définis dans le module en cours d'analyse à l'exception des tableaux (ADEFINIR).

3.3.4. Vérifications effectuées à la fin d'un module

A la fin de chaque module, certains contrôles et certaines modifications sont effectués :

a) Vérification que tous les modules du lexique ont été utilisés (TABLUTILE).
b) Vérification que tous les résultats qui devaient être définis l'ont été. Si certains n'ont pas été définis et que l'ensemble des initialisations

du module (INITA) n'est pas vide, on indique que ces résultats doivent être remontés (AREMONTER) et où ils peuvent être remontés (TABIN).

c) Modification des ensembles du tableau OK. Les résultats utilisés dans le module peuvent être utilisés dans tous les modules qui y ont été utilisés.

3.3.5. Vérifications effectuées à la fin de l'analyse

A la fin de l'analyse, quatre types de recherche sont effectués.

a) Recherche des résultats non définis (élément de ADEFINIR égaux à DOIT).
b) Recherche des résultats qui auraient dû être remontés et qui ne l'ont pas été à l'aide du tableau AREMONTER.
c) Recherche des résultats définis par récurrence qui n'ont pas été initialisés à l'aide du tableau INIREC.
d) Recherche des modules qui n'ont pas été définis à l'aide du tableau DEJADEFINI.

3.3.6. L'ordonnancement automatique

3.3.6.1. Présentation

Si l'utilisateur a choisi l'option d'ordonnancement automatique, à la fin de chaque module, les définitions de celui-ci sont réordonnées automatiquement. La méthode de réordonnancement est, rappelons-le, assez simple : une définition ne peut être exécutée que lorsque tous les objets qu'elle utilise ont déjà été définis. Le problème est légèrement compliqué par la présence de résultats définis par récurrence. En effet, il faut toujours utiliser la valeur précédent une suite récurrente avant d'en calculer une nouvelle.
Le tri qui va être utilisé pour mener à bien cet ordonnancement est donc défini comme suit :
soit x et y deux identificateurs ;
soit $\cup$ la relation entre identificateurs telle que $x \cup y$ signifie la définition de x utilise y.

Il s'agit d'un tri topologique pour le graphe dont les points sont les identificateurs et la relation Γ définie par :

$x \Gamma y \Leftrightarrow (y \cup x) \vee (x \cup \partial y)$.

3.3.6.2. Le principe du tri

Pour réaliser ce tri, on associe à chaque définition du module trois ensembles :

a) GAUCHE [i] qui est l'ensemble des résultats définis par la ième définition.
b) DROITE [i] qui est l'ensemble des résultats utilisés par la ième définition.
c) VDROITE [i] qui est l'ensemble des résultats utilisés par la ième définition mais dont la définition a été faite lors d'une itération précédente.

Les ensembles DROITE et VDROITE ont des rôles différents. En effet, une définition ne peut être ordonnée tant que son ensemble DROITE n'est pas vide et tant que des éléments de son ensemble GAUCHE appartiennent à l'ensemble VDROITE de l'une des autres définitions non encore ordonnées. Ces ensembles sont construits lors de l'analyse de chaque définition.

(1) X, A = DONNEE
DROITE=Ø VDROITE=Ø GAUCHE={X,A}

(2) IMPR=∂X, 'COUCOU', Y, Z
DROITE={Y,Z} VDROITE={X} GAUCHE=Ø

(3) X [I] = 2*Y+∂U-Z
DROITE={Y,Z,I} VDROITE={U} GAUCHE={X}

fig.4 : Exemple d'ensembles associés à des définitions.

D'autre part, pour pouvoir être réalisé, le tri ne doit s'effectuer que sur les objets effectivement définis dans le module. En effet, certains objets définis dans des modules de niveau supérieur peuvent être utilisés et ne doivent pas contrarier le réordonnancement.
Les ensembles DROITE et VDROITE de chaque définition du module seront donc réduits à leurs intersections avec la réunion de tous les ensembles GAUCHE du module.

Cette restriction étant faite, il ne reste plus qu'à effectuer le tri en respectant les règles suivantes :

Règle 1 :
Une définition peut être ordonnée lorsque son ensemble DROITE est vide et que son ensemble GAUCHE a une intersection vide avec la réunion des ensembles VDROITE des définitions non encore ordonnées du module.

Règle 2 :
Lorsqu'une définition a été ordonnée, les éléments de son ensemble GAUCHE doivent être ôtés des ensembles DROITE des définitions non encore ordonnées et son ensemble VDROITE doit être mis à vide afin de ne pas perturber la suite du tri.

L'ordonnancement peut se terminer de deux façons :

a) Favorablement, si toutes les définitions ont pu être ordonnées.
b) Défavorablement, si à un moment de l'ordonnancement on n'a pas pu trouver une définition satisfaisant la règle 1. Dans ce cas, l'ordonnancement est arrêté et donne lieu au message : 'ORDONNANCEMENT IMPOSSIBLE'.

$X = \partial Y + 1$
$Y = \partial X + 3$

fig. 5 : exemple de module où la règle 1 ne peut pas être satisfaite.

La méthode utilisée pour mettre en oeuvre ce tri, à chaque étape, est une méthode de parcours séquentiel de la liste des définitions du module pour trouver les définitions satisfaisant à la règle 1. Cette méthode est apparentée à celle présentée par Knuth [23]. Le nombre de définitions visitées est donc proportionnel à n^2. On aurait pu choisir une méthode plus performante de ce point de vue (nombre de visites proportionnelles au nombre d'arcs du graphe) [24]. Ce choix n'a pas été fait car la diminution du nombre de visites n'est pas justifiée par l'alourdissement qu'elle aurait entraîné sur la structure de données nécessaire (pile) et dans le traitement effectué à chaque visite. De plus, le nombre de définitions n'est jamais supérieur à 20 et est en moyenne de l'ordre de 4 ou 5.

On trouvera, en annexe D.2 une version légèrement simplifiée de l'algorithme utilisé dans le compilateur et écrit de manière déductive.

1 X = @Y + Z - W + 3 + C

2 Y = X + Z + A + D

3 U = @U + 2

4 Z = B + C - 3

5 B, D = DONNEE $ C

6 C : SI @X > 0 ALORS T1 SINON C = @U

6a définitions à ordonner

n° de déf.	GAUCHE	DROITE	VDROITE
1	{X}	{Z, W, C}	{Y}
2	{Y}	{X, Z, A, D}	∅
3	{U}	∅	{U}
4	{Z}	{B, C}	∅
5	{B,D}	{C}	∅
6	{C}	∅	{X, U}

6b ensembles avant l'ordonnancement

DROITE [1] ← {Z, W}

DROITE [4] ← {B}

DROITE [5] ← ∅

VDROITE [6] ← ∅

6c modifications apportées aux ensembles après le choix de la définition 6

fig. 6 : exemple d'ordonnancement automatique.

Dans l'exemple traité fig. 6, la première définition ordonnée est la définition 6, en effet :

(DROITE [6] = ∅) ∧

(∀ i : (i ∈ {1, ..., 6}) ∧ (i ≠ 6) GAUCHE [6] ∩ VDROITE [i] = ∅)

3.3.6.3. Problèmes posés par les tableaux

L'algorithme de réordonnancement échoue dans certains cas où l'ordonnancement ne pose aucun problème lorsqu'il est réalisé manuellement. En effet si dans les mêmes définitions, on utilise plusieurs éléments du même tableau, l'ordonnancement devient très délicat (cf. figure 7). Il faut alors avoir recours à l'ordonnancement manuel du module (cf. annexe A 32).

T [I] = @ T [I + 1]
T [I + 1]= @ T [I + 2]

Figure 7 Exemple de problème posé par les tableaux.

3.3.7. La traduction de l'analyse

3.3.7.1. Principe

Le principe de la traduction est très simple (cf. 141 d). A chaque définition SNOOPY correspond, en effet, une traduction systématique dans le langage objet choisi.

SNOOPY	FORTRAN	PASCAL
A = e	A = e	A : = e
IMPR = liste	OUTPUT liste	writeln (liste)
liste = DONNEE	INPUT	read (liste)
INI ; SI cas 1 ALORS T1 ; cas 2 ALORS T2 ; ..... SINON Tn	{INI} IF(¬ cas 1) GOTO 1 {T1} GOTO n 1 IF(¬ cas 2) GOTO 2 {T2} 2 GOTO n	{INI} if cas 1 then {T1} else if cas 2 then {T2} else .... else {Tn}

	{Tn} n CONTINUE	
INI ; POUR I DE P JQA Q REPETER T	{INI} DO 1 I = P, Q {T} 1 CONTINUE	{INI} for L : = 1 to q do {T}
INI ; JQA FIN REPETER T	{INI} 1 IF(FIN) GOTO 2 {T} GOTO 1 2 CONTINUE	{INI} while (¬ fin) do {T}
Traduction d'un module	Traduction de chaque définition dans l'ordre de l'ordonnancement	begin Traduction de chaque définition dans l'ordre de l'ordonnancement end

Figure 8 Traductions systèmatiques des définitions SNOOPY

3.3.7.2. Mise en oeuvre

3.3.7.2.1. Déclarations du programme

La génération des déclarations ne pose aucun problème. Elle se fera, en effet, directement à partir de la table des synboles.

3.3.7.2.2. Corps du programme

Comme nous l'avons vu, une traduction systématique permet de donner pour chaque type de définition une traduction dans le langage objet choisi.

Le programme de traduction utilisera donc une procédure permettant de traduire une définition.
Le corps de cette procédure sera formée d'un choix permettant suivant le type de la définition à traduire de lui associer la traduction systématique correspondante (cf. figure 8). Cette procédure est récursive. En effet la traduction d'une définition nécessite la traduction d'un ou plusieurs modules et donc d'un certain nombre de définitions.

```
procédure   traduneligne (i, j : integer) ; % jeme définition du ième
module %
     .....
begin
     case  listruct [i, j] ↑ . typdef of
    entrée : ...
    sortie : ...
    jqa    : ...
    .......
     end % case %
end ;
```

Figure 9 Structure globale de la procédure de traduction d'une ligne.

La traduction de l'analyse se résumera donc à traduire les définitions du module principal :

```
for i : = 1  to nbdef [1]  do traduneligne (1, i)
```

On peut ramarquer que la structure du programme ne dépend pas du langage objet utilisé. La conception d'un traducteur est donc simple. Il suffit en effet de programmer la traduction systématique de chaque type de définition (cf. figures 8 et 10)

```
jqa : begin
      for k : = 1 to nb def [initi] do traduneligne  (initi, k)  ;
      writeln ('WHILE (NOT', symbole [arret] ,') DO BEGIN') ;
      for k : = 1 to nbdef [tab] do traduneligne  (tab, k)  ;
      writeln ('END % WHILE %')
      end ;
```

initi est le n° du module d'initialisation
tab est le n° du module itéré
arret est le n° de l'identificateur d'arrêt

Figure 10 Traduction de la définition JQA en Pascal.
(légèrement simplifiée)

3.4. *REALISATION DU COMPILATEUR*

3.4.1. Langage utilisé pour écrire le compilateur

Le langage utilisé pour écrire le compilateur SNOOPY est Pascal. Ce choix a été fait pour diverses raisons :

- facilité de représentation des structures de données complexes (une des raisons qui a fait écarter FORTRAN)
- lisibilité des programmes obtenus
- langage procédural donc idéal pour écrire un compilateur (facilités lors des modifications)
- rapidité à l'exécution
- possibilité de gestion dynamique de l'espace de travail
- encombrement relativement faible en mémoire (principale raison qui a fait écarter PL/1 qui occupe 96 pages mémoire minimum sous SIRIS 8 contre 60 pour Pascal).

3.4.2. Structure en recouvrement du compilateur

Afin d'optimiser la place occupée par le compilateur en mémoire, nous avons décidé de donner au compilateur une structure en recouvrement. Pour ce faire, trois des plus grosses procédures qui ne sont jamais utilisées simultanément (Traductions et Ordonnancement) ont été écrites et compilées séparément. Lors de l'exécution, une seulement de ces trois procédures est présente en mémoire. (cf. figure 11)

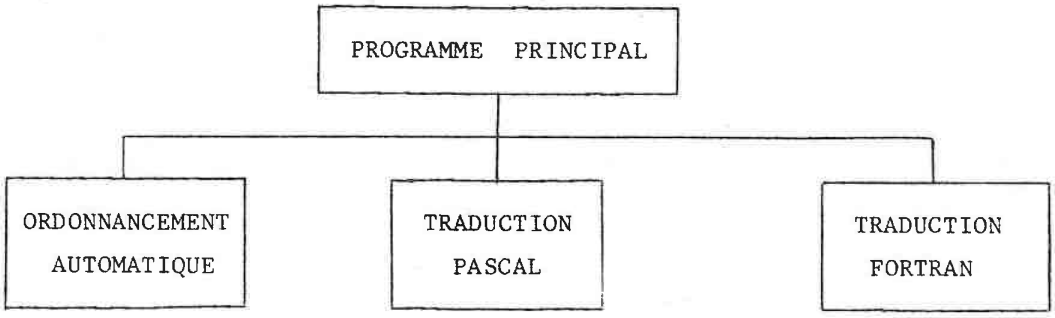


Figure 11 Arbre de recouvrement du compilateur

IV

un système d'aide à la construction d'analyses déductives

RESUME

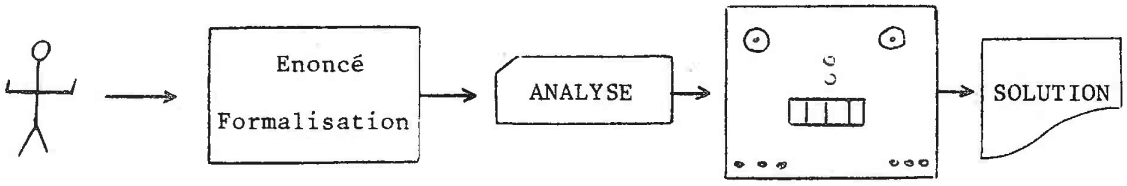
Ce chapitre montre tout d'abord la nécessité d'avoir un programme conversationnel pour aider les utilisateurs lors de l'élaboration d'analyses déductives. On décrit ensuite les diverses fonctions que doit accomplir un tel programme d'aide.

PLAN

4.1. *NECESSITE D'UNE VERSION CONVERSATIONNELLE.*

La Méthode de Programmation Déductive a été introduite pour faciliter la tâche du programmeur en l'aidant au niveau de l'énoncé et de la formalisation du problème.

Le langage SNOOPY ainsi que son compilateur ont été créés afin d'établir une continuité entre l'analyse déductive et le programme obtenu. Pour mettre en oeuvre SNOOPY, on s'est appuyé sur le fait que l'ordonnancement et la traduction de l'analyse pouvaient être automatisés et que ces deux étapes de la résolution étaient nettement séparées des deux précédentes (Enoncé et Formalisation).



(Compilation, ordonnancement, traduction)

Fig. 1 La construction d'un programme avec SNOOPY

La phase délicate de la résolution reste malgré tout l'écriture de l'analyse. On pourrait donc se demander si l'ordinateur ne pourrait contribuer à aider le programmeur dans cette phase. Pour cela nous allons recenser les diverses tâches accomplies au cours de l'analyse et qui pourraient être automatisées du fait de leur aspect systématique :

a) La construction du lexique des identificateurs est une tâche simple et systématique. A chaque fois qu'un nouvel identificateur est utilisé dans une définition formelle, il doit être référencé dans le lexique du module correspondant.

b) La construction du lexique des modules est faite d'une manière similaire à celle du lexique des identificateurs.

c) Le choix de certaines constructions syntaxiques peut être fait d'après certains critères bien déterminés (par exemple choix pour une itération entre un POUR et un JUSQU'A).

d) La vérification du bon emploi des identificateurs et de leur définition est aisée à faire du fait de la méthode d'analyse.

D'autre part, certaines modifications des définitions formelles et des lexiques peuvent intervenir en cours de résolution :

a) Fusions de définitions(itératives ou conditionnelles)

b) Remontées de définitions dans les modules d'initialisation

c) Transformations des définitions formelles

Le but d'un système d'aide à la construction d'analyse déductive serait donc de seconder le programmeur dans ces diverses tâches en le dégageant ainsi de nombreuses obligations syntaxiques et des lourdeurs nécessitées par les modifications de l'analyse. Le programmeur n'aurait plus alors qu'à aborder les tâches les plus délicates mais aussi les plus intéressantes et qui nécessiteront toujours l'intervention humaine :

a) Descriptions informelles des identificateurs

b) Définitions des résultats

c) Formalisation des définitions

Il est évident qu'un tel système d'aide au programmeur serait assez différent de celui qui a été présenté dans les chapitres précédents.

En effet, les tâches automatisables et celles qui ne le sont pas sont ici intimement mêlées. Un tel système devra être conversationnel.

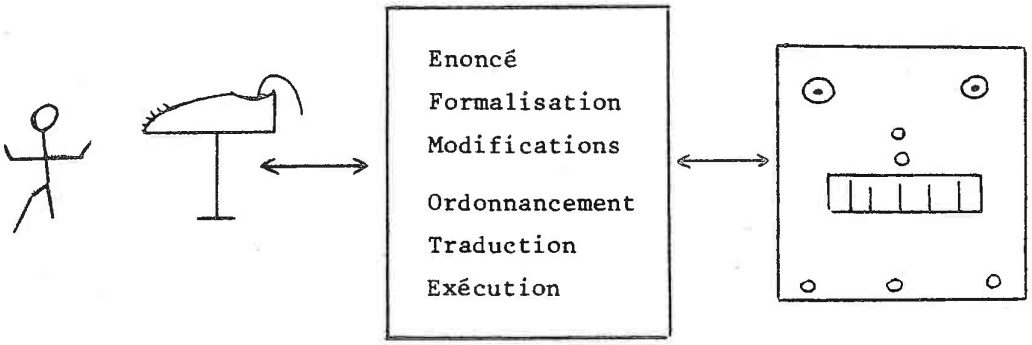


Fig. 2 Un système d'Aide à la Construction d'Analyses Déductives

4.2. *LES FONCTIONS QUE DOIT ACCOMPLIR UN SYSTEME D'AIDE A LA CONSTRUCTION D'ANALYSES DEDUCTIVES*

Lorsqu'un programmeur écrit une analyse déductive, quelles sont les diverses tâches qu'il aborde ? On peut citer les principales :

- Remplir les lexiques
- Formaliser les définitions
- Modifier l'analyse
- Relire ce qui a déjà été écrit
- Ordonner un module
- Traduire l'analyse
- Faire une exécution simulée d'une partie de l'analyse pour en vérifier l'exactitude.

Ces divers travaux peuvent en fait être groupés en deux classes bien distinctes de tâches :

- Les tâches liées à la construction de l'analyse
- Les tâches qui facilitent la construction et la mise au point de l'analyse.

De même, un Système d'Aide à la Construction d'Analyses Déductives comportera deux parties bien distinctes :

- Un Constructeur qui aidera l'utilisateur à écrire l'analyse

- Un Moniteur qui permettra à l'utilisateur de réaliser des tâches aussi variées que :

  + lister des définitions
  + modifier l'analyse
  + ordonner un module
  + exécuter une partie de l'analyse en conversationnel
  + etc.... etc...

Dans les paragraphes suivants, nous allons décrire plus en détail ces diverses fonctions.

4.3. *LE CONSTRUCTEUR*

4.3.1 Présentation

Pour donner une description des diverses tâches qui doivent être accomplies par le constructeur, il est bon d'avoir présent à l'esprit quelles sont les diverses constructions qui sont utilisées dans une analyse déductive. On peut en distinguer quatre :

a) Le lexique des identificateurs (la table des symboles) qui contient tous les renseignements concernant chaque identificateur utilisé dans l'analyse (type, description, etc ...)

b) Le lexique des modules qui contient les renseignements concernant chaque module (nombre de définitions, description, etc ....)

c) Les modules, qui sont composés de définitions formelles et qui sont utilisés pour définir un ou plusieurs résultats

d) Les définitions qui utilisent des identificateurs et des modules afin de définir un ou plusieurs résultats.

Nous allons essayer de voir plus en détail de quelle manière sont construits ces différents objets et quels sont les renseignements qu'il est bon de conserver au sujet de chacun d'eux. Dans le rencensement de ces renseignements, nous dinstinguerons ceux qui sont apportés par le programmeur de ceux qui peuvent être collectés automatiquement.

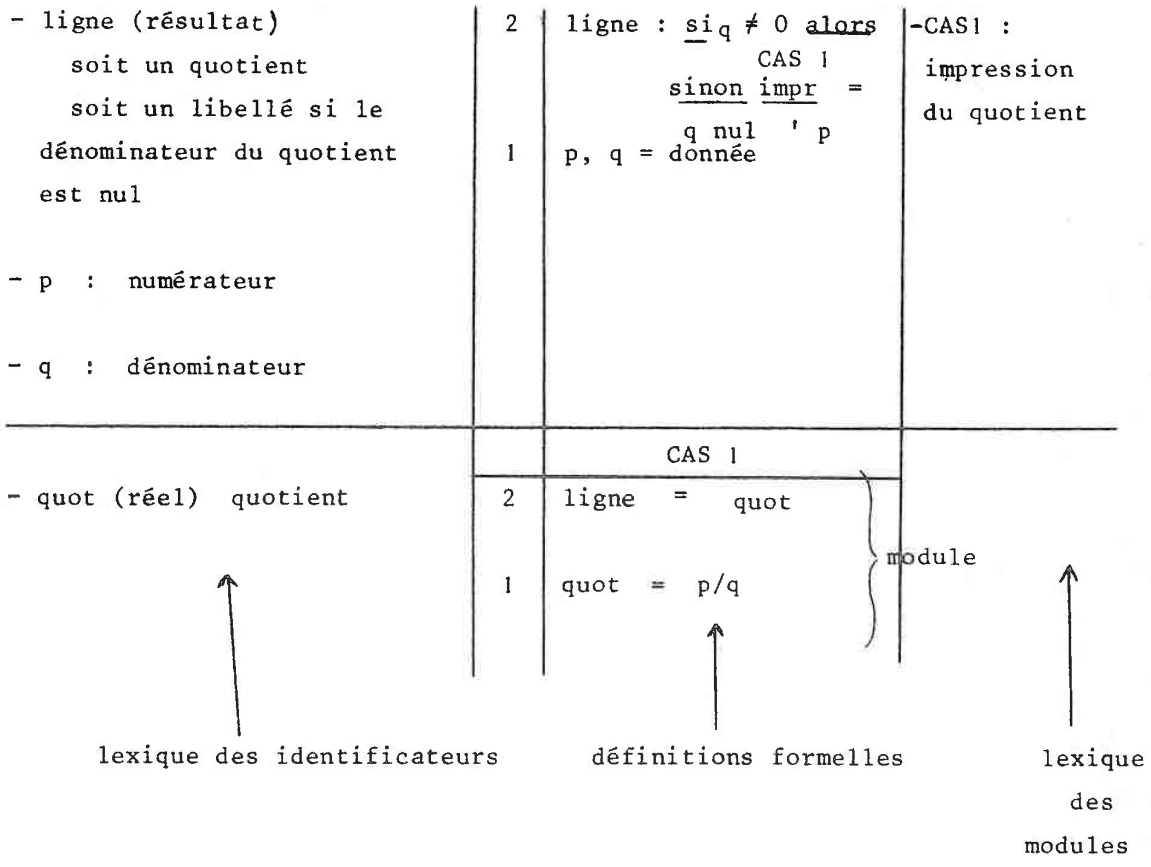


Fig. 3 Les diverses parties de l'analyse déductive

4.3.2 Le lexique des identificateurs

4.3.2.1 Principe de construction

Ce lexique permet de connaître pour chaque identificateur un certain nombre d'informations. Le mécanisme de sa construction est très simple : chaque fois qu'un nouvel identificateur est utilisé dans une définition formelle, il doit être ajouté au lexique des identificateurs.

4.3.2.2 Informations contenues dans le lexique

4.3.2.2.1. Informations fournies par le programmeur

Certains renseignements concernant l'identificateur sont fournis par l'utilisateur. Ces types de renseignements sont au nombre de deux :

- Définition informelle de l'identificateur
- Type de l'identificateur

4.3.2.2.2. Informations connues automatiquement

D'autres renseignements, au contraire, peuvent être déduits automatiquement à partir de la place et de l'utilisation de l'identificateur dans le lexique :

- Connaissance du module où est utilisé un identificateur pour la première fois
- Portée d'un identificateur (celui-ci pourra être utilisé dans tous les modules qui ont été introduits dans le module où l'identificateur a été utilisé pour la première fois)
- Mode de définition d'un identificateur (définition simple ou par une suite)

ident.	Type	Description	Module 1ère utilisation	Champs	Mode
ligne	Résultat	impression du quotient ou de "q nul"	Module 1	Module 1 CAS 1	Définition conditionnelle
p	Réel	numérateur	Module 1	Module 1 CAS 1	Définition Simple
q	Réel	dénominateur	Module 1	Module 1 CAS 1	Définition Simple
quot	Réel	quotient p/q	CAS 1	CAS 1	Définition Simple

Fig. 4 Lexique des identificateurs en fin d'analyse (cf. Fig. 3)

4.3.3 Le lexique des modules

4.3.3.1 Principe de construction

Ce lexique permet de connaître pour chaque module un certain nombre de renseignements. Le mécanisme de construction est, là aussi, très simple. A chaque fois qu'un module est utilisé dans une définition formelle, il doit être ajouté au lexique. On peut remarquer que, contrairement aux identificateurs, chaque module ne peut être utilisé qu'une fois dans l'ensemble des définitions formelles.

4.3.3.2. Informations contenues dans le lexique

4.3.3.2.1. Information fournie par le programmeur

La seule information que doit fournir l'utilisateur est une description informelle des fonctions de chaque module.

4.3.3.2.2. Informations connues automatiquement

De même que précédemment la place même où est utilisé un module permet de connaître un certain nombre de renseignements :

- Type de la définition où a été utilisé le module (conditionnelle, itérative, etc...)
- Connaissance éventuelle du module d'initialisation correspondant
- Etat de définition du module (" à définir" , "en cours de définition" , etc ...)
- Résultats qui doivent être définis à l'intérieur du module
- Nombre de définitions contenues dans le module si celui-ci est déjà défini ou s'il est en cours de définition.

nom	description	type de la définition	Init.	Etat	Résultats	nbre de définition
Module 1	---	---	---	Déjà défini	ligne	2
CAS 1	impression du quotient	condition-nelle	aucun	à définir	ligne	?

Fig. 5 Lexique des modules après la définition du module principal (cf. fig. 3)

4.3.4. Les Modules

4.3.4.1. Principe de construction

La construction d'un module se fait en deux étapes. Tout d'abord, on donne une définition formelle de tous les résultats qui doivent être définis par le module. Ensuite il faut définir les divers résultats intermédiaires introduits lors de la première étape. Cette deuxième étape introduisant elle aussi éventuellement des résultats intermédiaires, la construction du module sera terminée quand tous les résultats intermédiaires auront été définis formellement.

4.3.4.2. Informations utiles pour un module

En plus des renseignements fournis par le lexique des modules, il faudra connaître à tout moment lors de la construction, l'ensemble des identificateurs qui doivent être définis dans le module et qui ne l'ont pas encore été. On peut remarquer que contrairement au lexique des identificateurs, le lexique des modules peut être lié directement et simplement à l'utilisation des modules.

Tous les renseignements concernant un module au cours de sa construction pourront donc figurer dans le lexique des modules. Cette différence entre les deux lexiques est simple à expliquer : alors qu'un identificateur peut être utilisé à plusieurs endroits et défini dans plusieurs modules, un module n'est utilisé qu'à un seul endroit et défini qu'une fois et une seule. Cette unicité d'utilisation rend donc la correspondance avec le lexique très avantageuse.

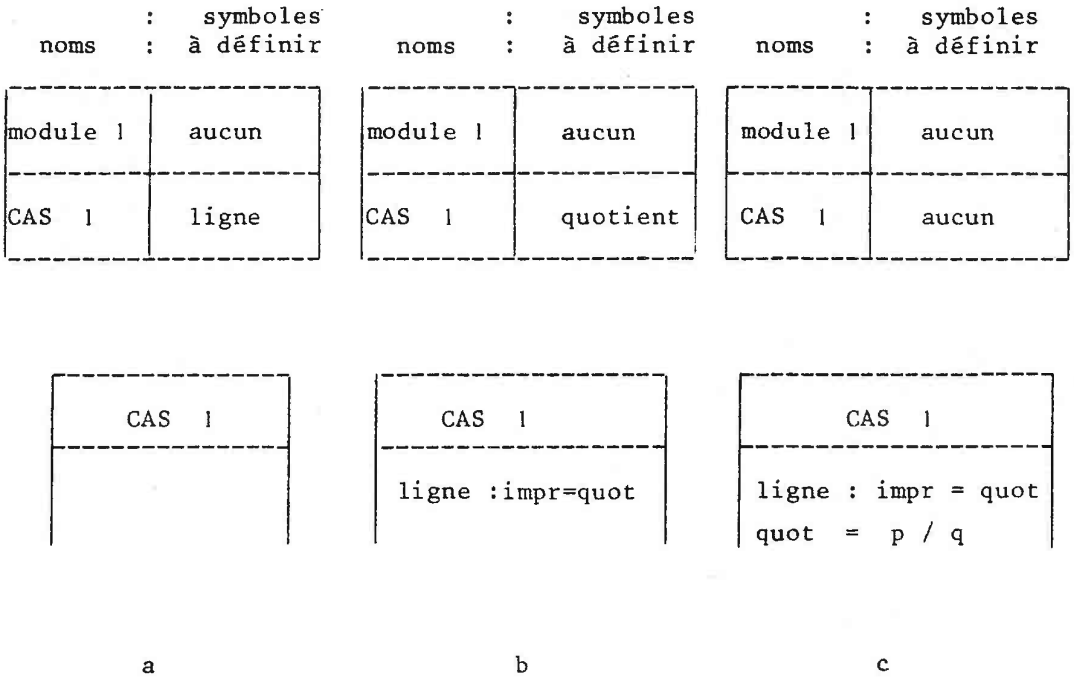


Fig. 6 Evolution du lexique lors de la construction d'un module (cf. fig. 3)

Nous rajouterons donc dans le lexique des modules une information indiquant à un instant donné les identificateurs restant à définir (cf. fig. 6)

4.3.5. Les Définitions

4.3.5.1. Principe de construction

Le programmeur ayant choisi un ou plusieurs résultats (finaux ou intermédiaires) parmi ceux qui doivent être définis ; il doit ensuite choisir par quel type de définition seront définis ces résultats (conditionnelle, itération, etc...). Il faut ensuite que l'utilisateur donne tous les renseignements nécessaires à la définition formelle qu'il a choisi (expression pour une définition simple, modules et conditions utilisés pour une définition conditionnelle, etc...).

On peut remarquer que c'est à ce moment que sont introduits de nouveaux identificateurs et de nouveaux modules et donc que les lexiques se modifient.

4.3.5.2. Informations contenues dans une définition

Tous les renseignements contenus dans une définition sont fournis par le programmeur sous le contrôle du système d'aide. Ils sont principalement de quatre sortes :

- ensembles des identificateurs définis
- ensembles des identificateurs utilisés
- modules utilisés avec leurs fonctions
- type de la définition (écriture, itération, etc...)

```
ligne : si q ≠ 0 alors cas 1
        sinon impr = 'q nul' $ p
```

identificateur défini	:	ligne
identificateurs utilisés	:	p , q
module utilisé	:	cas 1 module conditionnel
type de la définition	:	conditionnelle

Fig. 7 Exemple de renseignements liés à une définition (cf. fig. 3)

4.4. *LE MONITEUR*

4.4.1. Présentation

Le moniteur doit permettre à l'utilisateur de savoir à tout moment où il en est dans l'analyse et de modifier celle-ci. On peut ranger les différentes fonctions dans cinq catégories :

- Visualisation de l'analyse (listage des identificateurs, des définitions, etc...)
- Modification de l'analyse (modification d'une définition, fusion de deux itérations, etc...)
- Ordonnancement des définitions à l'intérieur de chaque module
- Traduction des analyses obtenues dans un langage de programmation existant
- Exécution des analyses (exécution totale ou partielle).

Nous allons voir plus en détail ces différentes fonctions dans les paragraphes suivants.

4.4.2. Fonctions de Visualisation

Cette fonction est essentielle, en effet, la structure conversationnelle de S.A.C.A.D. rend difficile une vue d'ensemble de l'analyse en cours d'élaboration. Cet obstacle n'était pas présent lors de l'utilisation de SNOOPY. Il faut donc pouvoir à tout moment visualiser toute l'analyse ou certaines parties de celle-ci. Ces fonctions de visualisation doivent être très maniables de manière à ne pas trop alourdir l'utilisation de S.A.C.A.D. Les diverses fonctions de visualisation sont les suivantes :

a) listages inconditionnels :

- listage du lexique des identificateurs
- listage du lexique des modules
- listage des définitions formelles de l'analyse

b) listages conditionnels :

- listage des identificateurs restant à définir dans un module
- listage des noms des modules qui ne sont pas encore complétement définis
- listage des définitions d'un module
- listage d'une définition
- listage des renseignements concernant certains identificateurs ou certains modules (le choix des identificateurs ou des modules étant fait selon certains critères : énumération, appartenance au lexique d'un module, ayant un type donné, etc...)

4.4.3. Fonctions de modification

4.4.3.1. Les deux types de modifications

Lors de l'élaboration d'une analyse déductive, deux types de modifications peuvent être mises en oeuvre :

- les modifications liées à la méthode déductive (remontées par exemple de définition dans un module d'initialisation)
- les modifications qui sont nécessaires quelle que soit la méthode de programmation utilisée (correction d'erreurs de conception de l'analyse, modification de la présentation des résultats, etc...)

4.4.3.2. Les modifications liées à la programmation déductive

Il y en a principalement deux :

- les remontées de définition dans des initialisations
- les fusions de définitions similaires (fusion d'itération, par exemple).

4.4.3.3. Les autres modifications

Ces modifications s'apparentent à celles effectuées à l'aide d'éditeurs de textes bien qu'elles soient faites à un niveau logique plus élevé.

Signalons entre autres :

- suppression d'une ou plusieurs définitions
- changement d'une ou plusieurs définitions
- adjonction d'un ou plusieurs résultats
- modification du résultat de l'analyse

4.4.3.4. Liens entre les fonctions de modification et le constructeur

La modification de l'analyse entraîne, dans la plupart des cas, des changements plus ou moins profonds au sein des structures de l'analyse :

- suppression de modules
- apparition d'identificateurs non définis à l'intérieur de modules qui étaient auparavant complets
- etc...

L'utilisation des fonctions de modifications nécessitera la plupart du temps la reconstruction de certaines parties de l'analyse. Il est essentiel que ces reconstructions soient faites sous le contrôle du Constructeur et non pas sous celui du Moniteur. Certaines modifications de l'analyse devront donc être interdites. Par exemple l'adjonction de nouvelles définitions. Ces restrictions éviteront de créer une ambiguïté, quant à la manière d'élaborer l'analyse, dans le choix entre l'utilisation du Constructeur ou du Moniteur.

4.4.4. Fonctions d'ordonnancement

L'ordonnancement des définitions à l'intérieur d'un module sera réalisé d'une manière similaire à celle utilisée dans SNOOPY (cf. chapitre 3). Le programmeur doit pouvoir choisir au niveau de chaque module s'il désire que celui-ci soit ordonné automatiquement ou non.

4.4.5. Fonctions de traduction

La traduction elle aussi sera réalisée d'une manière identique à celle utilisée dans SNOOPY. (cf. chapitre 3).

4.4.6. Fonctions d'exécution

L'exécution pourra être faite, au choix, en différé avec un listing de l'analyse en SNOOPY ou en conversationnel pour suivre l'exécution du programme.

L'exécution conversationnelle pourra éventuellement ne concerner qu'une partie de l'analyse.

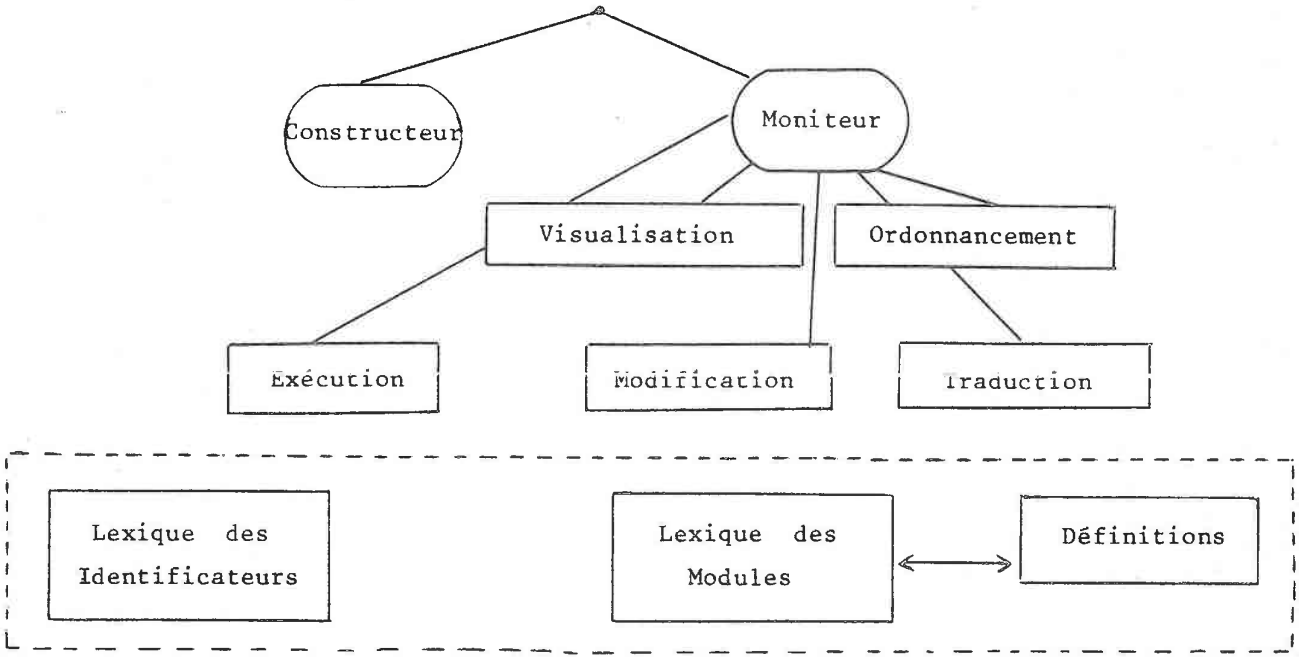


Fig. 8 Schéma récapitulatif du S.A.C.A.D.

V

utilisation de s.a.c.a.d.

RESUME

Ce chapitre présente de manière détaillée les diverses fonctions remplies par le S.A.C.A.D. On trouvera en annexe C, un exemple commenté d'une session S.A.C.A.D.

PLAN

5.1. *GENERALITES*

Le S.A.C.A.D est formé de deux sous-systèmes :

- le Moniteur
- Le Constructeur

Un des deux sous-systèmes est toujours présent en mémoire et durant toute la session, l'utilisateur est sous le contrôle d'un de ces deux sous-systèmes.

Lors de l'initialisation du S.A.C.A.D, le sous-système qui a le contrôle est le constructeur.

L'utilisateur a le choix entre deux options pour l'utilisation du S.A.C.A.D. Cette option est choisie en début de session :

a) Option libre (L) : l'utilisateur dispose d'une assez grande liberté dans l'utilisation du constructeur (choix des identificateurs à définir, etc...) (cf. § 52)

b) Option assistance (A) : le système guide l'utilisateur pas à pas dans la construction de l'algorithme.

A la fin de la session, les renseignements concernant l'analyse sont rangés sur deux fichiers :

- le fichier SOURCE contient l'analyse déductive
- le fichier OBJET contient la traduction de l'analyse dans un langage évolué si celle-ci a été demandée (cf. 534).

5.2. *LE CONSTRUCTEUR*

5.2.1. Présentation

Comme il a été dit dans le chapitre IV, le rôle du constructeur est d'aider l'utilisateur à construire une analyse déductive.

Cette aide se matérialise principalement par la construction automatique des lexiques par le système et par la vérification de la cohérence entre les définitions explicites et les lexiques au fur et à mesure de l'analyse. Cette aide est complétée dans le cas de l'option assistance (cf. 51) par le choix des identificateurs et des modules à définir.

Pour permettre à l'utilisateur d'élaborer son analyse, le constructeur met à la disposition de l'utilisateur deux types de commandes :

- les commandes de construction
- les commandes de fonctionnement

5.2.2. Les Commandes de Construction

Ces commandes permettent de construire l'analyse :

- définitions explicites des résultats (intermédiaires ou externes)
- descriptions des résultats (intermédiaires ou externes) à l'aide d'énoncés (formels ou informels).

Lors de l'analyse, le module en cours de construction est appelé module courant, et les définitions en sont numérotées de un en un à partir de la première.

Toutes les commandes de construction agissent sur le module courant. Elles sont de cinq sortes :

- Définition explicite de résultats (externes ou intermédiaires)
- Description de résultats externes
- Description de résultats intermédiaires
- Description du rôle des modules utilisés dans l'analyse
- Choix d'un module courant

La structure de ces commandes est identique. Elles sont toutes composées d'une chaîne de caractères ne contenant pas le caractère % suivie des deux caractères %(CR). Un ou plusieurs retours-chariot ((CR)) peuvent intervenir à l'intérieur de la commande.

5.2.2.1. Définitions explicites de résultats

Elles ont la même syntaxe que les définitions explicites SNOOPY (Cf. Chapitre II et annexe A). Deux restrictions interviennent cependant :

- tout résultat externe doit avoir un nom (IMPR n'est plus un mot réservé)
- toute définition structurée doit comporter une initialisation (éventuellement elle ne comportera aucune définition).

L'utilisateur doit introduire une définition de résultat à chaque fois que le système le lui demande. La requête formulée par le système n'est pas la même suivant l'option choisie.

a) Option L : le système indique le numéro de la définition à venir et la liste des identificateurs restant à définir. L'utilisateur ayant choisi un ou plusieurs de ceux-ci, le système après avoir donné la description de ceux-ci, attend l'introduction de la définition.

DEFINITION 2 (LIGNE, X, Y)

LIGNE % (CR)

UN NOM ET UN SALAIRE (RESULTAT)

2 : LIGNE = NOM, SALAIRE % (CR)

Fig. 1 Exemple de définition de résultat en mode L
(les parties écrites par le système sont soulignées)

b) Option A : après avoir donné la description du résultat à définir, le système attend l'introduction de la définition de celui-ci

LIGNE : UN NOM ET UN SALAIRE (RESULTAT)

2 : LIGNE = NOM, SALAIRE % (CR)

Fig. 2 Définition de résultat en mode A

En plus des définitions explicites SNOOPY, il existe deux définitions spéciales :

a) La définition par fusion

Elle permet de définir un ou plusieurs résultats à l'aide d'une définition ayant déjà été utilisée dans le module courant. Le nouvel entête de la définition obtenu est formé de celui de la définition utilisée pour la fusion et des résultats définis par la définition par fusion.

La syntaxe de cette définition est la suivante :

liste : FUSION n % (CR)

liste est la liste des résultats (ou le résultat en mode A) définis

n est le numéro de la définition où est réalisée la fusion.

```
DEFINITION  2  (X, Y, Z)
X % (CR)
INCONNUE
8 :  X  =  DONNEE  % (CR)
DEFINITION  3  ( Y, Z)
Y  % (CR)
VALEUR  MAXIMALE
3  :  Y  :  FUSION  2 % (CR)
2  :  X, Y  =  DONNEE
```

Fig. 3 Exemple de définition par fusion

b) Définition par remontée

Cette définition permet de différer la définition d'un ou plusieurs résultats. Ceux-ci devront être définis dans le module d'initialisation indiqué par l'utilisateur (si cela est autorisé). Ce type de définition est employé, entre autre, pour éviter de définir dans une itération un objet qui y est utilisé mais dont la définition est fixe. La syntaxe de cette définition est la suivante :

```
liste  :  REM / module /  % (CR)
```

liste est la liste des résultats (ou le résultat en mode A) définis

module est le nom du module d'initialisation où doit être remontée la définition.

```
SH : SALAIRE  HORAIRE (REEL)
3 : SH : REM / INI /  % (CR)
```

Fig. 4 Exemple de définition par remontée

5.2.2.2. Description de résultats externes

La description des résultats externes est faite dans deux cas :

- au début de la construction du module principal
- lors de la description d'un module itératif définissant des résultats externes. En effet, alors que le résultat externe définit par une itération est une liste, à l'intérieur de l'itération on définit un élément de la liste (cf 5124).

La syntaxe de la description d'un résultat est la suivante :

nom : description % (CR)

nom est l'identificateur de résultat externe.

Plusieurs descriptions peuvent se suivre. La dernière description doit être suivie de '% (CR)'.

Ces descriptions sont faites à la demande du système.

DECRIRE LES RESULTATS EXTERNES

LISTE : LISTE DE COUPLES (NOM, SALAIRE) % (CR)

LIGNE : IMPRESSION DE LA MASSE SALARIALE % (CR)

% (CR)

Fig. 5 Exemple de description de résultats externes

5.2.2.3. Description de résultats intermédiaires

Elle est faite par l'utilisateur, à chaque fois qu'un nouveau résultat est introduit dans l'analyse, à la demande du système.

Sa syntaxe est la suivante :

type, description % (CR)

type est l'un des types SNOOPY (cf.23)

DECRIRE N

ENT, NOMBRE D'EMPLOYES (TITULAIRES (CR)

ET CONTRACTUELS) % (CR)

Fig. 6 Exemple de description d'un résultat intermédiaire

5.2.2.4. Description des fonctions d'un module

Elle permet de décrire les divers rôles que remplit un module utilisé dans l'analyse. Elle est faite à la demande du système lors de l'introduction de chaque nouveau module. Sa syntaxe est la suivante :

description % (CR)

DECRIRE LE MODULE TOTO

DEFINITION DU POURCENTAGE DES VOTES % (CR)

Fig. 7 Exemple de description de module

La description peut éventuellement être complétée en cours d'analyse, à la demande du système (après une fusion par exemple) ou de l'utilisateur (commandes ≠ ADM et ≠ CHM cf 523)

5.2.2.5. Choix d'un module courant

Le choix d'un nouveau module courant est fait à la demande du système ou à la requête de l'utilisateur (commande ≠ CHA cf 523). Ce choix ne peut être fait que lorsque l'option libre a été choisie (cf 51).

L'utilisateur doit donner le nom d'un module utilisé dans l'analyse, et qui n'est pas encore complétement défini. Pour cela, il choisit parmi ceux que lui propose le système.

CHOISIR UN MODULE COURANT (INI, TAB)

TAB % (CR)

CALCULE U ET DEFINIT X

Fig. 8 Exemple de choix de module courant
Le système fournit la description du module choisi

5.2.3. Les commandes de fonctionnement

Les commandes de fonctionnement permettent à l'utilisateur de demander au système certains renseignements concernant l'analyse. Ces renseignements sont importants et, pour la plupart, liés au module courant. L'utilisation de ces commandes évite donc d'alourdir la session en utilisant le moniteur trop souvent . Ces commandes sont au nombre de huit :

a) ≠ AND (CR) annulation de la dernière définition du module courant

b) ≠ ANM (CR) annulation de toutes les définitions du module courant

c) ≠ DEF (CR) impression des définitions déjà écrites du module courant

d) ≠ ADM (CR) adjonction de renseignements à la description d'un module

e) ≠ CHM (CR) changement de la description d'un module. Pour cette commande comme pour la précédente, le système demande à l'utilisateur le nom du module et la nouvelle description.

```
≠ CHM (CR)
MODULE ?
TOTO % (CR)
DESCRIPTION DU MODULE TOTO
IMPRIME UN NOMBRE ET SON CARRE %(CR)
```

Fig. 9 Exemple d'utilisation de ≠ CHM

f) ≠ CHA (CR) changement de module courant (uniquement en mode libre)
g) ≠ MON (CR) appel du moniteur (cf. 53)
h) ≠ FIN (CR) fin de la session. L'analyse est recopiée sur le fichier SOURCE

5.3. *LE MONITEUR*

5.3.1. Présentation

Le Moniteur est le complément indispensable du Constructeur. Il permet à l'utilisateur de faire des manipulations aussi variées que possible sur l'analyse. Ces manipulations vont de l'obtention de renseignements sur l'analyse à l'exécution conversationnelle de certains modules. Ces manipulations ont été regroupées en six tâches. Lorsque l'utilisateur est sous contrôle moniteur, il peut soit avoir accès à l'une de ces tâches, soit quitter le contrôle du moniteur. Les commandes utilisables ont toute la même syntaxe et sont au nombre de huit :

a) CONS (CR) retour au contrôle constructeur
b) FIN (CR) fin de la session. L'analyse est recopiée sur le fichier SOURCE
c) EXEC (CR) accès à la tâche d'exécution
d) ORDO (CR) accès à la tâche d'ordonnancement
e) TRAD (CR) accès à la tâche de traduction

f) MODI (CR) accès à la tâche de modification
g) VISU (CR) accès à la tâche de visualisation
h) DOCU (CR) accès à la tâche de documentation

Lorsqu'une des six tâches a été choisie, l'utilisateur a accès à toutes les commandes relatives à cette tâche. Pour changer de tâche, il lui suffit d'utiliser une des huit commandes citées précédemment.

5.3.2. La tâche d'exécution

Cette tâche permet d'exécuter toute ou une partie de l'analyse en conversationnel. La commande est la suivante :

[/MOD/] i , j (CR)

(les parties entre crochets sont optionnelles)

Cette commande permet d'exécuter les définitions du module MOD dont les numéros d'ordonnancement sont compris entre i et j. La valeur particulière 0 pour i et pour j peut être utilisée, elle désigne respectivement le début et la fin du module. Si le nom du module est omis, le module exécuté est le module principal.

Lorsque le S.A.C.A.D a besoin d'une valeur (donnée ou résultat dont la définition n'a pas encore été totalement écrite) il la demande à l'utilisateur.

```
EXEC (CR)
/TOTO/ 0, 3 (CR)
A ?
12 (CR)
A AU CARRE = 144
```

Fig. 10 Exemple d'utilisation de la commande d'exécution

5.3.3. La tâche d'ordonnancement

Cette tâche permet d'ordonner les définitions d'un module. Elle comporte elle aussi une seule commande. Sa syntaxe est la suivante :

[/MOD/] { A / M } (CR)

(les parties entre accolades indiquent le choix entre plusieurs options)

Le module MOD est ordonné :

- automatiquement, si l'option A est choisie
- manuellement, si l'option M est choisie. L'utilisateur doit alors donner l'ordre des définitions à la demande du S.A.C.A.D.

Un même module peut être réordonné plusieurs fois au cours de la même session.

```
ORDO (CR)
/TOTO/A (CR)
ORDRE : 3 1 2 5 4
/TOTO/ M (CR)
ORDRE ?
3 2 1 5 4 (CR)
```

Fig. 11 Exemple d'utilisation de la tâche d'ordonnancement

5.3.4. La tâche de traduction

Cette tâche ne peut être utilisée que lorsque l'analyse est complète. Elle utilise la commande suivante :

{ PASC / FORT } (CR)

Suivant l'option choisie, l'analyse est alors traduite en FORTRAN ou en Pascal. Le programme obtenu est recopié sur le fichier OBJET. Le programme contenu dans le fichier OBJET est le dernier qui a été généré.

5.3.5. La tâche de modification

Cette tâche permet de modifier certaines parties de l'analyse. Il y a cinq commandes disponibles :

a) R/MOD1/i/MOD2/(CR)

La définition de numéro i du module MOD1 est remontée dans le module MOD2 si cela est autorisé.

b) F [/MOD/] i,j (CR)

Les définitions i et j du module MOD sont fusionnées si cela est autorisé. Les noms des modules utilisés pour la nouvelle définition sont ceux de la définition numéro i. Si MOD est absent, le module concerné est le module principal.

c) D [/MOD/] i, j (CR)

Les définitions i à j du module MOD sont détruites. Si nom est omis, le module concerné est le module principal.

d) S/ NOM1 /{R / M}/ NOM2 / (CR)

L'identificateur de résultat (option R) ou de module (option M) NOM1 est remplacé par l'identificateur NOM2 dans toute l'analyse.

e) M/XXX/{R / M}(CR)

La description de l'identificateur de résultat (option R) ou du module (option M) XXX peut être modifiée.

```
M/TOTO/M (CR)
DECRIRE LE MODULE TOTO
IMPRESSION DU SALAIRE % (CR)
M/X/I (CR)
DECRIRE X
REEL, VALEUR MAXIMALE DE LA PRESSION % (CR)
```

Fig. 12 Exemple de modifications de descriptions

5.3.6. Tâche de visualisation

Cette tâche permet de visualiser certaines parties de l'analyse. Les commandes liées à cette tâche sont au nombre de sept :

a) [/MOD/] { ∅ / D } i, j (CR)

Impression des définitions i à j du module MOD. Si le nom du module est omis, le module considéré est le module principal.

i et j représentent les numéros des définitions (option D) ou les numéros d'ordonnancement des définitions (option ∅). Dans le second cas, les définitions sont listées dans leur ordre d'exécution.

b) [/MOD/] { R / M } (CR)

Impression du lexique des résultats (option R) ou des modules (option M) du module MOD. Si le nom du module est omis, le module considéré est le module principal.

c) L { R / M } (CR)

Impression du lexique des résultats (option R) ou des modules (option M)

d) { R / M } /NOM1,...., NOMN/ (CR)

Impression des parties du lexique des résultats (option R) ou des modules (option M) concernant les identificateurs NOM1,..., NOMN

e) [/MOD/] L (CR)

Impression de tous les renseignements concernant le module MOD (description, lexiques, définitions). Si le nom du module est omis, c'est le module principal qui est considéré.

f) LA (CR)

Impression de toute l'analyse

g) RENS (CR)

Impression de renseignements divers :

- nombre de modules
- nombre de résultats
- temps déjà utilisé
- etc...

5.3.7. Tâche de documentation

Cette tâche permet de connaître des renseignements sur l'utilisation de S.A.C.A.D (commandes, fonctionnement, etc...).

VI

remarques sur

l'implémentation de s.a.c.a.d.

RESUME

Ce chapitre permet de faire quelques remarques sur l'implémentation de SACAD. Pour cela, on y mène un parallèle entre SNOOPY et SACAD. On présentera, de plus, la structure du système.

PLAN

6.1. *SACAD ET SNOOPY*

Le SACAD est utilisé pour élaborer des analyses déductives. Lors de cette élaboration, l'utilisateur se sert d'un langage proche de SNOOPY. Il va donc y avoir certaines similitudes entre l'implémentation du SACAD et celle du compilateur SNOOPY.

D'un autre côté, la structure conversationnelle du SACAD et les possibilités données à l'utilisateur de modifier l'analyse en cours d'écriture vont nécessiter un certain nombre de différences de conception importantes entre SNOOPY et SACAD.

Nous allons dans la suite, recenser rapidement ces ressemblances et ces différences et voir les conséquences entraînées dans l'implémentation de SACAD par sa structure conversationnelle.

6.2. *LES RESSEMBLANCES*

6.2.1. Vérification de la cohérence

La vérification de la cohérence entre les lexiques et les définitions ressemblera, quant à sa mise en oeuvre, à celle qui est utilisée dans le compilateur SNOOPY. Elle sera cependant beaucoup simplifiée par la construction pas à pas de l'analyse. En effet, certaines ambiguïtés qui pouvaient apparaître en SNOOPY lors de la compilation sont ici levées par les réponses de l'utilisateur. Citons entre autres, le problème des remontées qui sont clairement indiquées au système (cf. 522! et 535 a)

6.2.2. Ordonnancement automatique

L'ordonnancement automatique avec SACAD sera mené d'une manière totalement identique à celle utilisée par le compilateur SNOOPY (cf. 336).

6.2.3. Traduction Systématique

La traduction de l'analyse dans un langage évolué sera elle aussi accomplie de la même manière que celle utilisée par le compilateur SNOOPY (cf. 337).

6.2.4. Analyse des définitions explicites

L'analyse syntaxique des définitions explicites sera elle aussi très proche de celle du compilateur SNOOPY. Quelques simplifications interviendront cependant :

- la suppression de l'identificateur externe réservé IMPR facilitera l'analyse de l'entête de la définition
- la connaissance, avant l'analyse, de chaque définition des objets définis (cf. 5221) simplifiera la tâche de l'analyseur
- la présence obligatoire d'un module d'initialisation dans chaque définition structurée (SI, POUR, JQA) permettra de systématiser l'analyse des définitions structurées :

  + analyse de l'entête de la définition
  + analyse de la partie initialisation
  + analyse du corps de la définition

- l'introduction de nouveaux modules sera simplifiée. En effet, alors qu'en SNOOPY le lexique des modules était déjà connu lors de la compilation des définitions explicites, avec SACAD, ce lexique est construit à la suite de l'analyse de chaque définition.

6.3. *LES DIFFERENCES*

6.3.1. Structure interne de l'analyse

La construction progressive de l'analyse et les possibilités de modification autorisées par le système requièrent une structure interne de l'analyse beaucoup plus raffinée et surtout plus dynamique que

celle utilisée dans SNOOPY. En effet, le système doit pouvoir aisément ajouter ou supprimer des modules, des définitions et des résultats. La structure interne sera la suivante :

a) Lexique des résultats.

On utilisera une structure chaînée

b) Lexique des modules

Là aussi, on utilisera une structure chaînée et de plus chaque module comportera un pointeur donnant accès à sa première définition et un pointeur donnant accès à sa dernière définition.

c) Définitions

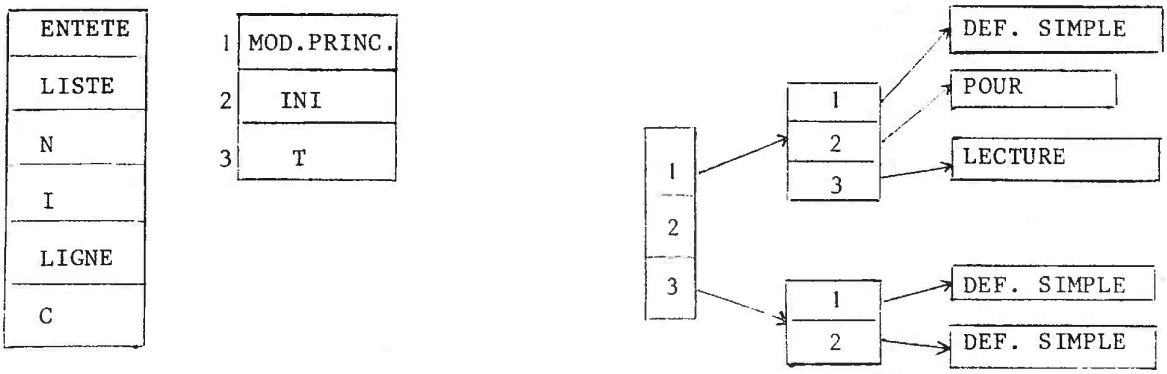
Les définitions d'un même module seront chaînées entre elles de deux manières :

- dans leur ordre d'apparition
- dans leur ordre d'exécution.

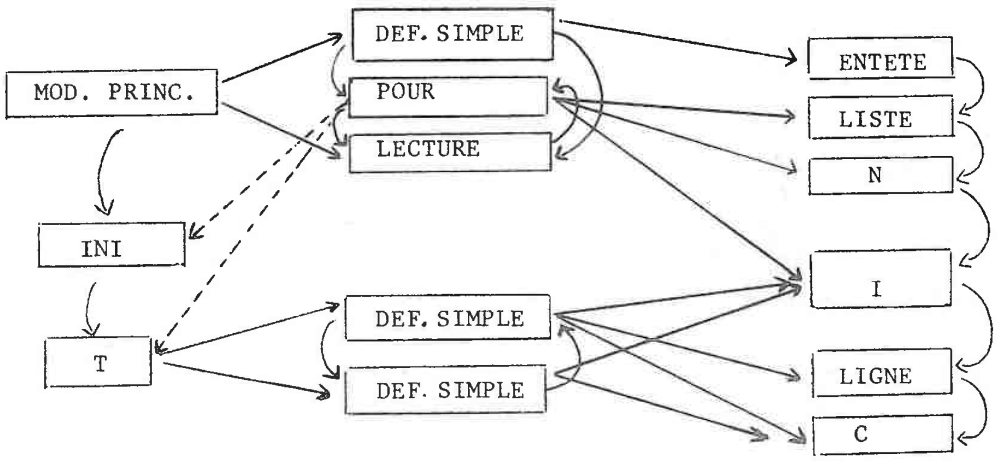
De plus, chaque définition comportera des pointeurs sur les différents modules et les différents résultats utilisés.

-ENTETE (RESULTAT)	1	ENTETE ='TABLE DES CARRES'	- T : IMPRIME I et I^2
-LISTE (RESULTAT) TABLE DES CARRES	3	LISTE : INI ; POUR I DE 1 JQA N REPETER T	
-N(ENT) NOMBRE DE LIGNE DE LA TABLE	2	N = DONNEE	
		T	
-LIGNE (RESULTAT) NOMBRE ET SON CARRE	2	LIGNE = I, C	
-C (ENT) CARRE DE I	1	C = I * I	

1 a Analyse



1b Structure Semi-dynamique (SNOOPY)



1c Structure dynamique (SACAD)

Fig. 1 Structure interne de l'analyse déductive

6.3.2. Tâches de modification et de visualisation

Ces deux tâches sont grandement facilitées par la structure dynamique de l'analyse (cf. 631). Leur mise en oeuvre nécessitera l'écriture d'un genre d'éditeur de texte.

6.3.3. Tâche d'exécution

Cette tâche nécessitera elle, l'écriture d'un interprète. La présence de cette tâche imposera de choisir pour les expressions (arithmétiques et booléennes) une syntaxe rigoureuse et de faire une analyse syntaxique des expression plus détaillée que celle réalisée par le compilateur SNOOPY (cf. 25). En effet, il ne s'agira plus seulement de vérifier la cohérence de la définition, mais aussi de générer un code interne exécutable.

6.4. *STRUCTURE DU SYSTEME*

SACAD est appelé à être utilisé en temps partagé. Il est donc vital pour lui d'occuper le moins de place possible en mémoire lors de son utilisation.

Un gain appréciable de place en mémoire peut être réalisé grâce à la structure modulaire et hiérarchisé du SACAD (cf. chapitre IV). En effet, lors de l'utilisation d'une des tâches de SACAD, les autres tâches n'ont pas besoin d'être en mémoire centrale.

Les différentes tâches de SACAD seront donc utilisées en recouvrement et ce, à plusieurs niveaux :

- niveau Constructeur ou Moniteur
- niveau choix de la tâche du moniteur
- niveau choix du langage de traduction.

Un autre avantage de la structure modulaire du SACAD est la possibilité de lui adjoindre de nouvelles tâches sans avoir à réaliser de grandes modifications dans le système.

Parmi les tâches qu'il serait possible d'adjoindre citons entre autre :

- tâche d'aide à la mise au point d'analyse
- tâche d'optimisation
- adjonction de nouveaux langages de traduction

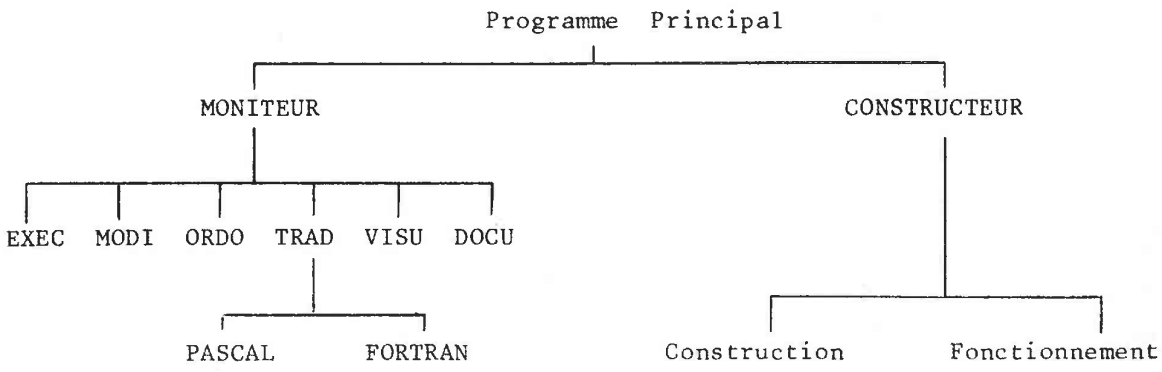


Fig. 2 Arbre de recouvrement du SACAD

conclusion

Ce travail entre dans le cadre des recherches effectuées pour améliorer la construction des programmes.

La Méthode de Programmation Déductive ne doit pas être considérée comme "la méthode idéale", mais comme une des approches possibles, parmi celles qui sont actuellement faites, de la construction de programmes [8, 20, 21, 22]

L'étude que nous avons présentée comporte trois parties.

Dans la première nous avons présenté la Méthode de Programmation Déductive à partir d'une réflexion sur une démarche possible lors de la résolution d'un problème (chapitre I). Cette méthode donne toujours lieu, à l'heure actuelle, à des travaux de recherche au sein du Centre de Recherche en Informatique de Nancy, notamment en ce qui concerne l'emploi de structures de données complexes, son utilisation pour de gros projets, en particulier en informatique de gestion, la mise au point de programmes et leur optimisation ainsi que la validation de son emploi pour l'enseignement.

Dans la seconde partie, nous avons introduit le langage SNOOPY, langage destiné à faciliter la mise en oeuvre de la programmation déductive (chapitre II). Nous avons aussi présenté le compilateur SNOOPY (chapitre III). SNOOPY pourrait être quelque peu amélioré, notamment par l'introduction des procédures (cf 25) et de structures de données plus élaborées (les listes chaînées par exemple). Ce langage a été utilisé cette année dans l'enseignement de l'algorithmique à l'I.U.T. Informatique de Nancy, ainsi qu'à la MIAGE.

La troisième partie, enfin, nous a permis de présenter S.A.C.A.D, un système conversationnel devant permettre une utilisation attrayante et originale de la méthode déductive (chapitres IV, V et VI). La réalisation complète du S.A.C.A.D est prévue et doit permettre notamment d'y faire certaines adjonctions, en particulier, en ce qui concerne la mise au point et l'optimisation de programmes (cf 64).

annexe A

manuel d'utilisation de snoopy

SOMMAIRE

SOMMAIRE

1. PRESENTATION DE SNOOPY

SNOOPY est un langage de construction déductive d'algorithmes. Il a été mis en oeuvre afin de favoriser l'utilisation et l'enseignement de la Méthode de Programmation Déductive.

Le choix a été de ne pas générer un programme objet dans un langage proche de la machine mais de générer un programme écrit dans un langage évolué choisi par l'utilisateur. Ce choix devrait permettre d'utiliser, suivant les problèmes, les constructions syntaxiques adaptées (par exemple, FORTRAN pour le calcul scientifique ; on pourrait envisager de générer du SNOBOL 4 pour un problème de traitement de chaînes de caractères).

SNOOPY a figé la syntaxe de la Méthode de Programmation Déductive en essayant de limiter la rigueur de certaines constructions syntaxiques.

2. NOTATIONS

2.1. Caractères de base

Il y a 59 caractères autorisés :

les lettres A ... Z ;
les chiffres 0 ... 9 ;
les caractères spéciaux : + - * / ( ) $ = ␣ , .
' [ ] : # % | & ∂ < > ;

2.2. Mots réservés

Il y a 22 mots réservés. Ceux-ci ne peuvent pas être utilisés comme identificateurs.

ALØRS	BØØL	CAR	CAS	CHAINE	DE
DØNNEE	ENT	ET (ou &)	FAUX	IMPR	JQA
NØN	ØU (ou |)	PØUR	REEL	REPETER	RESULTAT
SI	SINØN	TAB	VRAI		

2.3. Identificateurs simples

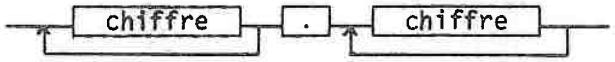
Un identificateur simple est une suite de lettres ou de chiffres commençant par une lettre. Seuls les huit premiers caractères sont significatifs.

2.4. Nombres

La syntaxe des nombres sans signe est l'une des suivantes :

entier sans signe :

réel sans signe :



2.5. Chaînes de caractères

Toute suite de caractères comprise entre deux apostrophes est une chaîne de caractères.
Pour représenter le caractère apostrophe à l'intérieur d'une chaîne, il faut utiliser deux apostrophes consécutives.

Exemples : 'SNØØPY'
'C''EST UN EXEMPLE'

2.6. Utilisation des blancs

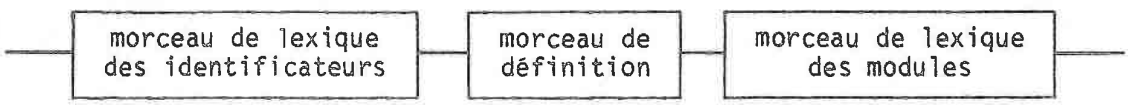
Les blancs compris entre les entités syntaxiques ne sont pas significatifs ; par contre, ils ne peuvent être utilisés à l'intérieur d'une entité syntaxique.

3. STRUCTURE D'UN PROGRAMME

Toute ligne d'un programme correspond soit à un ensemble de trois morceaux (lexiques et définition), soit à un entête de module, soit à un commentaire.

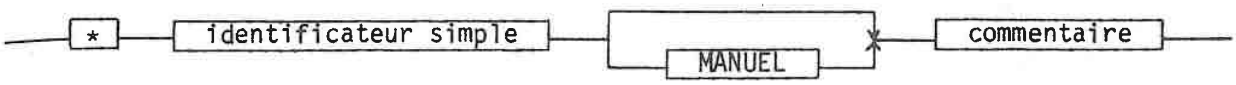
3.1. Ligne normale

Elle se décompose en trois parties :



3.2. Entête de module

L'entête d'un module a la syntaxe suivante :



Le mot MANUEL permet en cas d'ordonnancement automatique d'autoriser un module à être ordonné manuellement (cf. aussi 5.1. et 9.2.). MANUEL n'est pas un mot réservé.

3.3. Commentaire

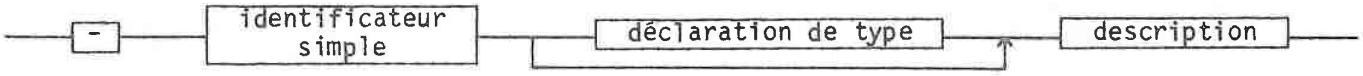
Toute ligne commençant par le caractère $ est un commentaire et est donc ignorée par le compilateur.

4. OBJETS MANIPULES

4.1. Identificateurs

4.1.1. Référence dans le lexique des identificateurs

Tout morceau de lexique commençant par le caractère - est une référence d'identificateur. Sa syntaxe est la suivante :



La description peut tenir sur plusieurs lignes du lexique. La déclaration de type est facultative dans certains cas (cf. 9.2.).

4.1.2. Types simples

Ils sont au nombre de 6. Les opérateurs applicables aux divers types dépendent du langage objet utilisé (cf. 1. et 5.2.1.).

4.1.2.1. Entiers

Exemple : -X (ENT) : premier coefficient de l'équation.

4.1.2.2. Réel

Exemple : -TØTØ (REEL) : racine

4.1.2.3. Booléen

Exemple : -T (BOOL) vrai si x > y

4.1.2.4. Caractère

Exemple : -U (CAR ␣) 1ère lettre

4.1.2.5. Chaînes de caractères

Le nombre maximum de caractères autorisé pour une chaîne dépend du langage objet utilisé (par exemple en PASCAL : au plus 8 caractères).

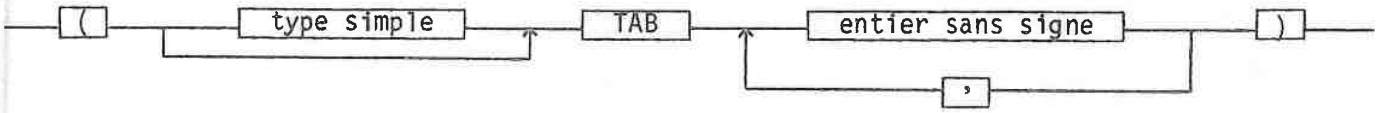
Exemple : - NØM (CHAINE) de l'employé

4.1.2.6. Résultats

Exemple : -LISTE (RESULTAT) des salaires

4.1.3. Type tableau

Les tableaux peuvent posséder une ou deux dimensions. Leurs composantes peuvent être de l'un des 6 types simples dont il est parlé plus haut.
La déclaration de type a la forme suivante :



Exemple : -CARTE (CAR TAB 80) carte lue.

Le type simple peut être omis dans certains cas (cf. 9.2.).

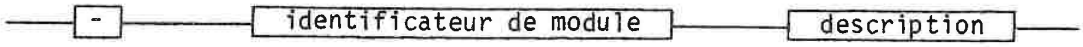
Dans les définitions, les indices de tableaux peuvent indifféremment être notés entre crochets ou entre parenthèses. Dans le cas où le langage objet nécessite l'utilisation des crochets, ceux-ci devront être utilisés.

Note : Pour les installations ne possédant pas les crochets, une notation équivalente est : (. et .). Pour la perforation des cartes, il faut utiliser pour obtenir les crochets ç et !.

Exemples : A [2, I-3] B (.2*X-3.) C (3, 4)

4.2. Identificateurs de modules

Un identificateur de module est un identificateur simple. La référence d'un tel identificateur dans le lexique des modules a la syntaxe suivante :



La description peut tenir sur plusieurs lignes du lexique.

Exemple : -RECH : recherche du PGCD de a et b

5. LES DEFINTIONS

5.1. Numéro d'ordonnancement

Si l'ordonnancement est manuel (cf. 9.2. et 3.2.), toute définition devra être précédée par un entier sans signe dans la colonne réservée à cet effet (cf. 9.2.) indiquant son ordre d'exécution dans le module où elle apparaît. Ce numéro devra être inférieur ou égal au nombre de définition du module et positif.

Exemples :

```
| 12 | X = 3
|  9 | Y : si A = 2 alors Y = 1 sinon Y = 2
```

Dans le cas contraire, le numéro d'ordonnancement pourra être remplacé par un ou deux caractères autres que blanc.

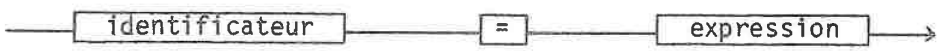
Exemples :

```
|  *  | X = 3
| 99  | Y : si A = 2 alors Y = 1 sinon Y = 2
```

5.2. Définitions simples

5.2.1. Définition explicite simple

Syntaxe :



La syntaxe de l'expression dépend du langage objet et du type de l'identificateur (on peut remarquer que dans le cas où la compilation n'est pas suivie de traduction, cette syntaxe est donc assez libre (cf. 5.4.)).

L'identificateur définit peut-être un identificateur simple ou indicé.

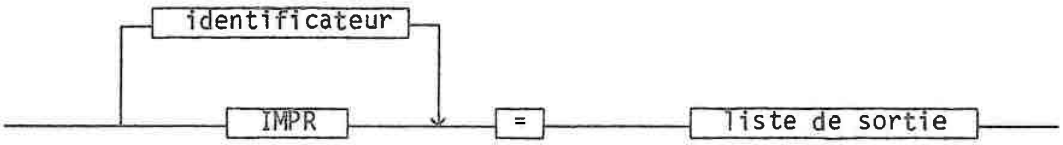
En cas de définition récurrente, la distinction entre deux niveaux consécutifs de la récurrence sur un identificateur est obtenue en précédant le plus ancien du caractère ∂ (ce dernier ne peut pas être utilisé récursivement).

Exemples :

```
x = ∂x + 3 * y
z = 'LE' . nom . verbe
```

5.2.2. Définition externe (impression)

Syntaxe



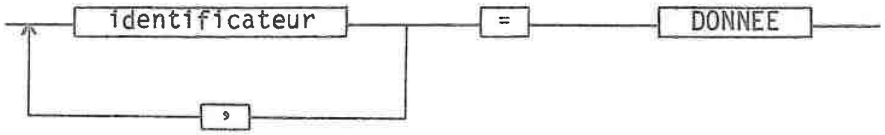
La liste de sortie est une liste d'identificateurs et de chaînes de caractères séparés par des virgules.

Exemples :

```
ligne = 'MOYENNE', MOY
impr = 'x =', x, 'c''est␣tout'
```

5.2.3. Définition par lecture

Syntaxe



5.2.4. Remarques sur les entrées/sorties

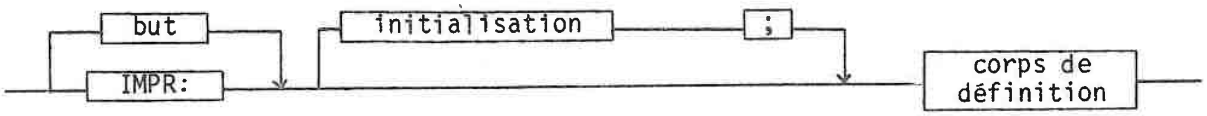
Leur utilisation n'est pas la même suivant le langage objet utilisé. En PASCAL, en plus des possibilités habituelles (format libre, lecture d'entiers ou de réels, écriture d'entiers, de réels, de chaînes, de booléens), il est possible de lire des chaînes (une procédure à cet usage est systématiquement générée dans le programme objet). En FORTRAN, les ordres d'entrée/sortie sont INPUT et OUTPUT.

5.3. Définitions structurées

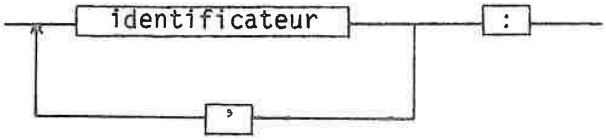
5.3.1. Syntaxe générale

Les définitions structurées sont de deux types : itératives et conditionnelles.

Syntaxe



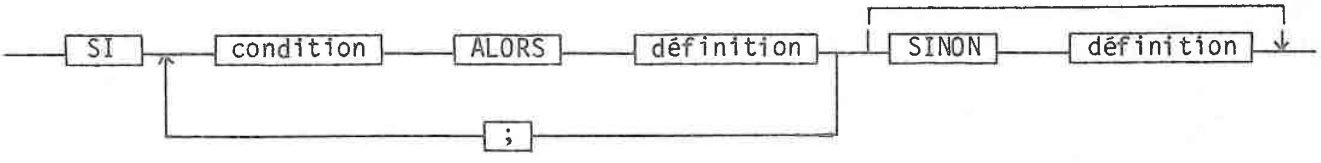
Le but a la forme suivante :



L'initialisation peut être soit un identificateur de module, soit une définition simple.

Lors de l'exécution, l'initialisation est toujours exécutée.

5.3.2. Définitions conditionnelles



Les définitions sont soit des identificateurs de module soit des définitions simples.

Les conditions sont des expressions booléennes (cf. 5.4.). La partie SINON est facultative.

Lors de l'exécution, les conditions sont évaluées dans l'ordre de leurs apparitions jusqu'à ce que l'une d'elles soit vérifiée ou jusqu'à épuisement ; dans le premier cas, la définition correspondante est alors exécutée.

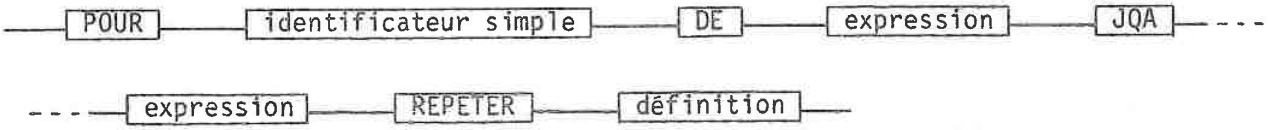
Exemples :

```
res : si
  delta > 0 alors réel 2 ;
  delta = 0 alors réel 1
            sinon impr = 'imaginaire'
x, y : si a > 0 alors x = ∂x + 3
                sinon y = 3
```

5.3.3. Définitions itératives

5.3.3.1. Itération avec nombre d'itérations connu

Syntaxe



L'identificateur simple de la boucle peut ne pas être référencé dans le lexique. S'il y est référencé, il doit être de type entier.

Lors de l'exécution, la définition itérée est répétée pour chacune des valeurs entières de l'identificateur simple comprises entre les deux

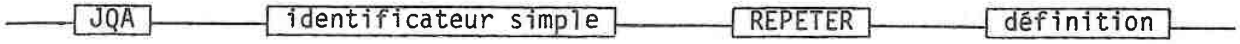
expressions.
La définition est soit un identificateur de module, soit une définition simple.

Exemples :

x : pour i de 1 jqa n-2 répéter x = ∂x + 3
y : INI ; pour j de 2 * x jqa 7 répéter T

5.3.3.2. Itération avec condition d'arrêt

Syntaxe



L'identificateur simple d'arrêt peut ne pas être référencé dans le lexique. S'il y est, il doit être de type booléen.

La définition est soit un identificateur de module, soit une définition simple.
Lors de l'exécution, la définition itérée est exécutée jusqu'à ce que l'identificateur d'arrêt ait la valeur VRAI. Si l'identificateur a la valeur VRAI avant l'exécution de l'itération, la définition itérée n'est donc jamais exécutée.

Exemples :

x : fin = faux ; jqa fin répéter TØTØ
y : INI ; jqa arrêt répéter X

5.4. Remarque sur les expressions

Les expressions arithmétiques ont la même syntaxe que celles du langage objet utilisé.
Les expressions booléennes sont formées à l'aide d'expressions arithmétiques, de parenthèses, des opérateurs booléens (

< > < = > = # & | ET OU

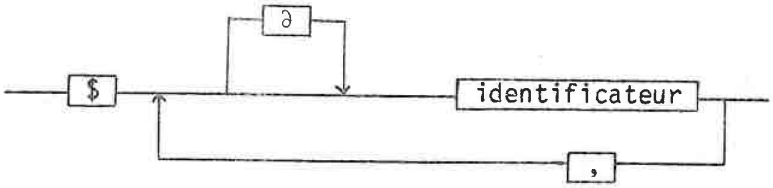
NON) et des constantes booléennes (VRAI, FAUX).

5.5. Liste d'ordonnancement

Dans le cas de l'ordonnancement automatique (cf. 9.2.), toute définition

autre qu'une définition explicite peut être suivie d'une liste d'identificateurs pour aider à l'ordonnancement d'un module.

Syntaxe



Exemple :

impr = x, y $ ∂z, u

6. FONCTIONS STANDARD

Elles sont liées au langage objet utilisé.

Actuellement, les fonctions standard utilisées sont les suivantes :

langage objet / signification	PASCAL	FORTRAN	AUCUN
division entière de a par b	a DIV b(*)	-	-
reste de la division de a par b	a MØD b(*)	MØD (a,b)	MØD (a,b)
$\sqrt{x}$	SQRT (x)	SQRT (x)	SQRT (x)
x^2	SQR (x)	-	-
$\|x\|$	ABS (x)	ABS (x)	ABS (x)
partie entière de x	TRUNC (x)	IFIX (x)	-
conversion réelle de x	-	FLØAT (x)	-
appartenance de a à un ensemble	a IN [...](*)	-	-

Remarque : Une fonction standard ne peut être utilisée comme identificateur.

(*) DIV, MOD et IN ne sont pas en fait des fonctions PASCAL mais sont considérées ici en tant que telles.

7. CONTRAINTES SYNTAXIQUES

7.1. Modules

Règle 1 : Tout module utilisé dans un module doit être référencé dans le lexique correspondant.

Règle 2 : Tout module apparaissant dans le lexique doit être utilisé et défini une fois et une seule.

Remarque 1 : Le nombre maximum de modules (y compris les définitions explicites utilisées dans des définitions structurées) est limité à 40.

Remarque 2 : Le nombre maximum de modules, le nombre de définitions est limité à 20.

7.2. Identificateurs

7.2.1. Portée des identificateurs

Règle : Un identificateur peut être utilisé dans un module lorsqu'il apparaît dans le lexique de ce module ; il peut alors être utilisé dans tous les modules utilisés dans ce même module.

7.2.2. Définition des identificateurs

Règle 1 : Si un identificateur apparaît dans le lexique d'un module, alors il doit être défini dans ce module sauf si sa définition est remontée dans une initialisation.

Règle 2 : Un identificateur ne peut être défini qu'une seule fois dans un module donné (sauf s'il est de type tableau).

Règle 3 : Tout identificateur défini par une définition structurée doit respecter les règles de définition suivantes :

1 - L'identificateur peut être défini dans l'initialisation s'il y en a une.

2 - Si la définition structurée est itérative, l'identificateur doit être défini dans la définition itérée.

3 - Si la définition est conditionnelle, l'identificateur doit être défini dans au moins une des définitions qui la composent.

Remarque 1 : Pour les identificateurs de type résultat, la règle 3 est facultative.

Remarque 2 : Lorsqu'un identificateur est défini de manière récurrente à l'intérieur d'une itération, il doit être défini aussi dans l'initialisation correspondante. En particulier, l'identificateur booléen d'un JQA doit être défini dans l'initialisation de l'itération.

Remarque 3 : Le nombre d'identificateurs est limité à 40.

8. PRESENTATION DES TRAVAUX

La mise en page des programmes (séparation des modules, colonnes de séparation des lexiques, des définitions et des numéros d'ordonnancement, numérotation des lignes) est faite automatiquement.

En cas d'ordonnancement automatique, les définitions de chaque module sont numérotées par ordre croissant et leur ordonnancement est indiqué à la suite du programme.

Une carte du programme est donnée, indiquant :

- pour les identificateurs, leurs types et leurs occupations mémoire ;
- pour les modules, leurs fonctions et leurs niveaux d'imbrication.

En cas d'erreurs syntaxiques, les messages d'erreur et leur significations sont listés à la suite.
Si le programme comporte des données, celles-ci sont listées.

Le programme objet est ensuite listé si le programme est correct et si l'option de listage est positionnée (cf. 9.2.).

9. UTILISATION DE SNOOPY

9.1. Cartes commandes

Les cartes à utiliser pour l'utilisation de SNOOPY sont les suivantes (les cartes entre accolades sont facultatives et dépendent de la présence ou non de données et de l'exécution éventuelle du programme objet).

```
!LIMIT (CØRE, 70)
!EXEC  SNØØPY
 carte paramètre (cf. 9.2.)
  Programme SNOOPY
{ #       }
{ Données }
{ !EXEC nom }
```

"Nom" est le nom de la procédure cataloguée permettant l'exécution du programme objet (actuellement TABLEP pour PASCAL et TABLEF pour FØRTRAN).

En annexe B, se trouvent les détails des procédures SNOOPY, TABLEP et TABLEF.

9.2. Options

Elles sont indiquées à l'aide de la carte paramètres. Celle-ci a le format suivant :

a, b, c, d x y z

9.2.1. Options de compilation (a, b, c, d)

9.2.1.1. Type implicite (a)

La déclaration de type n'est pas nécessaire pour les identificateurs de ce type (cf. 4.1.)

- R réel
- E entier
- X pas de type implicite

9.2.1.2. Langage objet utilisé (b)

- F FØRTRAN
- P PASCAL
- X pas de traduction

9.2.1.3. Ordonnancement (c)

- O automatique
- X manuel

9.2.1.4. Listage du programme objet (d)

- L listage
- X pas de listage

9.2.2. Options de mise en page (x, y, z)

Les trois entiers sans signe x, y et z séparés par des blancs représentent respectivement

- la dernière colonne du lexique des identificateurs
- la dernière colonne de table des définitions
- la dernière colonne de la ligne

Dans le cas où z est supérieur à 80, chaque ligne du programme doit être codée sur deux cartes.

Les colonnes x+1 et x+2 sont utilisées pour l'ordonnancement.

Remarque : $2 < x+2 < y < z < 120$

9.3. Structure du compilateur

SNOOPY est écrit en PASCAL CII (Siris 7/8). La structure du compilateur est la suivante :

- un module principal ;
- des modules séparés utilisés en recouvrement pour l'ordonnancement et les traductions.

ANNEXE A : Exemple de programme

- liste 1 (résultat) : liste des clients du mois de décembre avec le nombre de vignettes restantes et soit le cadeau obtenu, soit un libellé expliquant la raison de l'absence de cadeau
- liste 2 (résultat) : liste des clients ayant droit aux cadeaux surprises avec le nombre de cadeaux surprises pour chaque client
- nbvicl (ent tab 1 000) : donne le nombre de vignettes restant à chaque client

1	liste 1, nbvicl : INI 1 ; jqa fin 1 répéter DECEMBRE
2	liste 2 : pour i de 1 jqa 1 000 répéter AN $ nbvicl

- DECEMBRE : imprime pour un client ayant passé une commande au mois de décembre, son n°, le nombre de vignettes à son actif et soit le cadeau dont il bénéficie soit la raison pour laquelle il n'a pas de cadeau (arrêt par épuisement)
- AN : imprime le n° d'un client et le nombre de cadeaux surprises auxquels il a le droit s'il a plus de 5 vignettes
- INI 1

DECEMBRE

- ligne 1 (résultat) : n° du client, soit n° du cadeau, soit raison (pas de cadeau), nombre de vignettes restantes
- cadeau (bool) : vrai si le client a un cadeau
- nucli (ent) : n° du client
- montant (réel) de la commande
- nucalu (ent tab 10) : n° des cadeaux choisis par le client
- nuca (ent) : n° du cadeau obtenu
- nbvd (ent) : nombre de vignettes après la commande
- x (ent) : nombre de vignettes pour la commande
- nbcad (ent tab 20) : nombre de cadeaux en réserve
- nbvi (ent tab 20) : nombre de vignettes nécessaires pour chaque cadeau

4	ligne 1, nbvicl, nbcad : si cadeau alors T1 sinon T2 $ ∂ nucalu, nuca, ∂ nucli, nbvi
3	cadeau, nuca : INI 2 ; jqa fin 2 répéter RECH $ nbvd, ∂ nucad, nbvi
6	fin 1 = (nucli = 0)
5	nucli, montant, nucalu = donnée
2	nbvd = ∂ nbvicl (∂ nucli) + x
1	x : si ∂ montant ≤ 50 alors x = 1 sinon si ∂ montant ≤ 100 alors x = 4 sinon si ∂ montant ≤ 150 alors x = 15 sinon x = 15 + 10 * (montant - 200)/50)

- T1 : imprime le n° du client, son cadeau et le nombre de vignettes dont il dispose
- T2 : imprime le n° du client, la raison pour laquelle il n'a pas de cadeau et le nombre de vignettes dont il dispose
- RECH : regarde si un cadeau choisi peut être attribué au client
- INI 2

		T 1	
	3	nbcad (nuca) = ∂ nbcad (nuca) - 1	
	2	ligne 1 = ∂ nucli, nuca, nbvicl (∂ nucli)	
	1	nbvicl (∂ nucli) = nbvd - nbvi (nuca)	
		T 2	
- épuise (bool) : vrai si le dernier cadeau choisi est épuisé - lib (chaîne) : raison de l'absence de cadeau	4	ligne 1 = ∂ nucli, lib, nbvicl (∂ nucli)	
	1	nbvicl (∂ nucli) = nbvd	
	3	lib : si nucalu (1) = 0 alors lib = 'PAS DE CADEAU DEMANDE' sinon si épuisé alors lib = 'DERNIER CADEAU DEMANDE EPUISE' sinon lib = 'PAS ASSEZ DE VIGNETTES POUR LE DERNIER CADEAU DEMANDE'	
	2	épuisé = (∂ nbcad (nuca) = 0)	
		RECH	
- k (ent) : indice du n° de cadeau dans le choix	2	nuca, cadeau : si nucalu (k) ≠ 0 alors CAD	- CAD : définit cadeau et nuca suivant qu'il y a ou non des cadeaux en réserve et que le client a assez de vignettes ou pas
	3	fin 2 = (cadeau ou (k = 10) ou (nucalu (k) = 0))	
	1	k = ∂k + 1	

		CAD	
	2	cadeau = (nbvd ≥ nbvi (nuca)) et (∂ nbcad (nuca) ≠ 0)	
	1	nuca = nucalu (k)	
		INI 2	
	1	k = 0	
	2	cadeau = faux	
	3	fin 2 = faux	
		AN	
- ligne 2 (résultat) : n° du client et nombre de cadeaux surprises si plus de 5 vignettes	1	ligne 2 : si nbvicl (i) ≥ 5 alors IMPRIMER	- IMPRIMER : imprime le n° du client et le nombre de cadeaux surprises auxquels il a droit
		IMPRIMER	
- nbcs (ent) : nombre de cadeaux surprises	2	ligne 2 = i, nbcs nbcs = (nbvicl(i))/5	
		INI 1	
	3	nucli, montant, nucalu = donnée	
	2	fin = faux	
	1	nbvicl, nucad, nbvi = donnée	

ANNEXE B

1. Procédure SNOOPY

```
!ASSIGN  X,MTN,FIL,(STS,ØLD),(NAM,n°cpte),(CTG)
!ASSIGN  ERRF,FIL,(STS,ØLD),(NAM,——),(UNT,ØP,X)
!ASSIGN  SØUR,MTN,FIL,(SIZ,4,1),DCB,(FRM,F),(REL,80)
!ASSIGN  DEFI,FRE,FIL,(SIZ,4,1),DCB,(FRM,V),(REL,132)
!ASSIGN  DØNN,MTN,FIL,DCB,(FRM,F),(REL,80)
!ASSIGN  MESS,FRE,FIL,(SIZ,4,1),DCB,(FRM,V),(REL,132)
!ASSIGN  LØ,DEV,ØUT,DCB,(LIN,255)
!ASSIGN  LM,FIL,(STS,ØLD),(NAM,——),(UNT,ØP,X)
!ASSIGN  PMDØ,FIL,(STS,ØLD),(NAM,BIDØN)
!RUN␣,  ØPTIØN,F=6
```

2. Procédure TABLEF

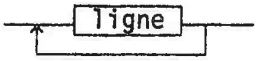
```
!ASSIGN  SI,ØPL,SØUR
!FØRTRAN SI,GØ
!ASSIGN  LIB,(STS,ØLD),(NAM,F4LIB),(UNT,AC,:SYS)
!LINK,ØPTIØN(UNSAT,LIB)
!ASSIGN  LØ,DEV,ØUT,DCB,(LIN,255)
!ASSIGN  10,ØPL,DØNN
!RUN
```

3. Procédure TABLEP

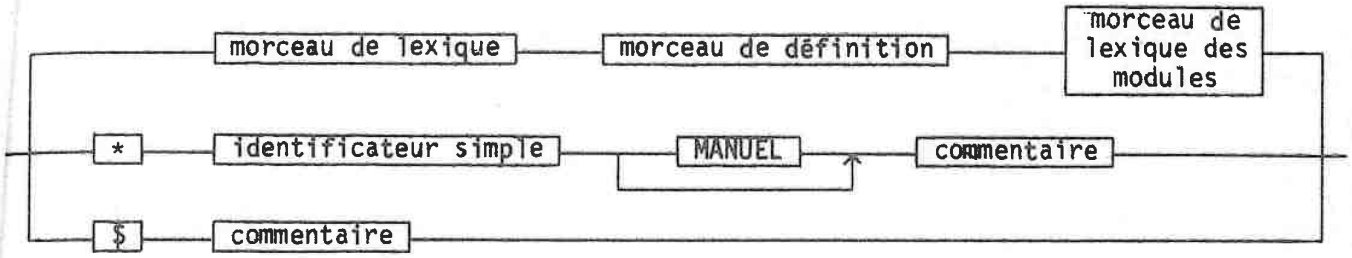
```
!ASSIGN  SI,ØPL,SØUR
!ASSIGN  PMDF,FIL,(STS,MØD),(NAM,BIDØN)
!PASCAL/T+,P+
!ASSIGN  LIB,FIL(STS,ØLD),(NAM,PASLIB),(UNT,AC,:SYS)
!LINK,ØPTIØN  (UNSAT,LIB)
!ASSIGN  PMDØ,FIL,(STS,ØLD),(NAM,BIDØN)
!RUN␣,ØPTIØN,F=4
```

ANNEXE C : GRAMMAIRE DE SNOOPY

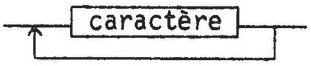
PROGRAMME



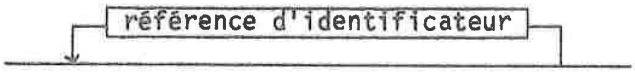
LIGNE



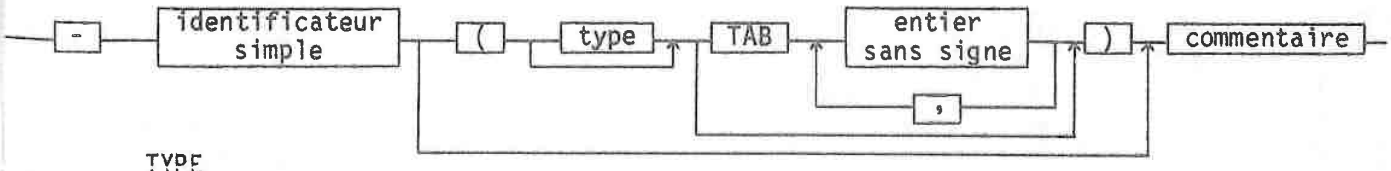
COMMENTAIRE



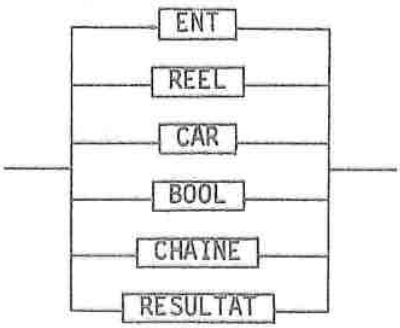
LEXIQUE



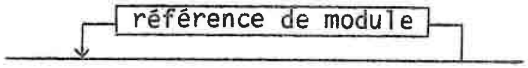
REFERENCE D'IDENTIFICATEUR



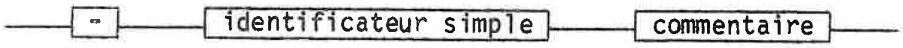
TYPE



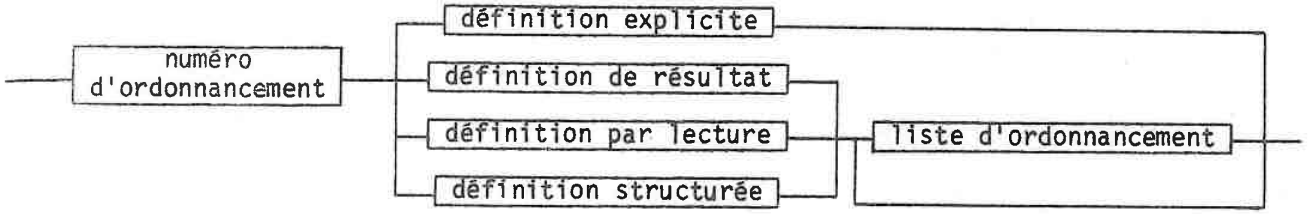
LEXIQUE DES MODULES



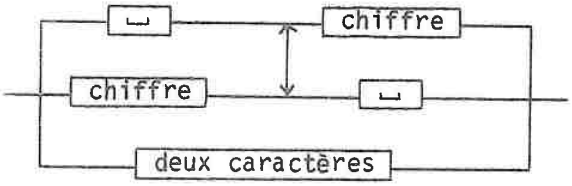
REFERENCE DE MODULE



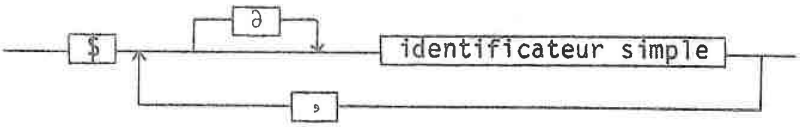
DEFINITION



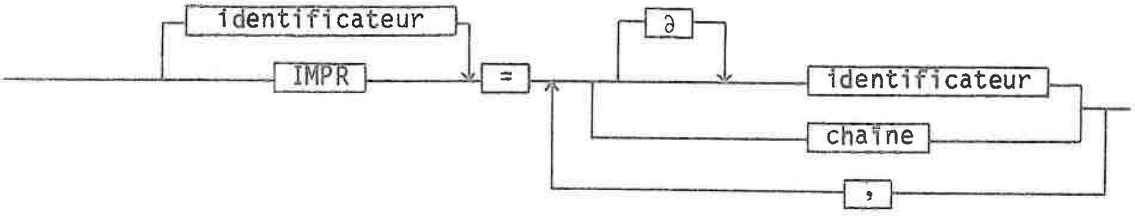
NUMERO D'ORDONNANCEMENT



LISTE D'ORDONNANCEMENT

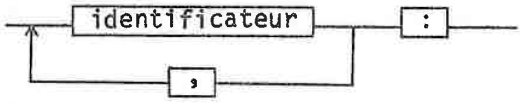


DEFINITION DE RESULTAT

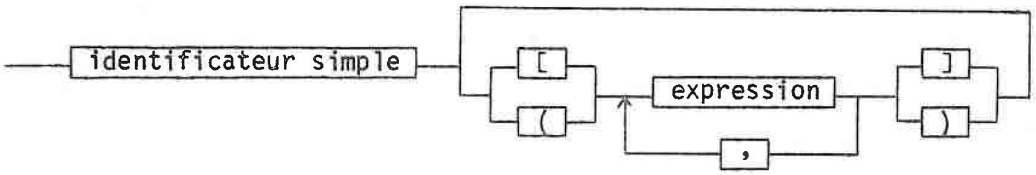


DEFINITION PAR LECTURE

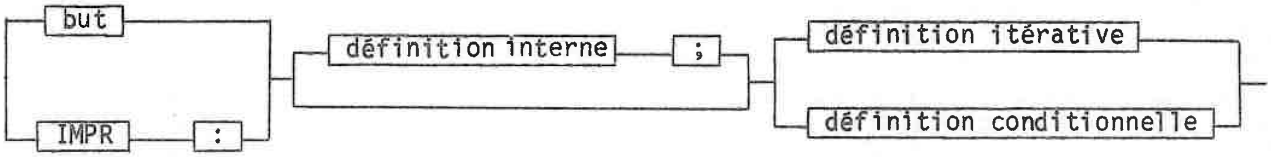
BUT



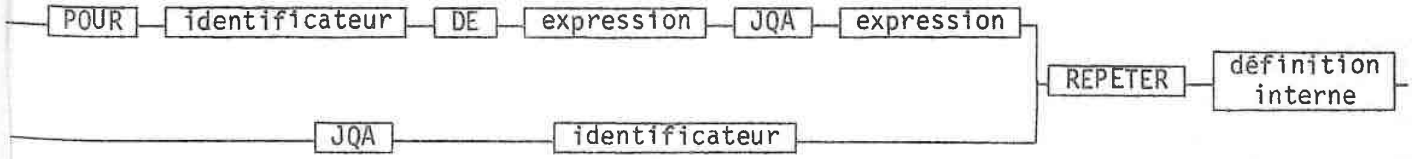
IDENTIFICATEUR



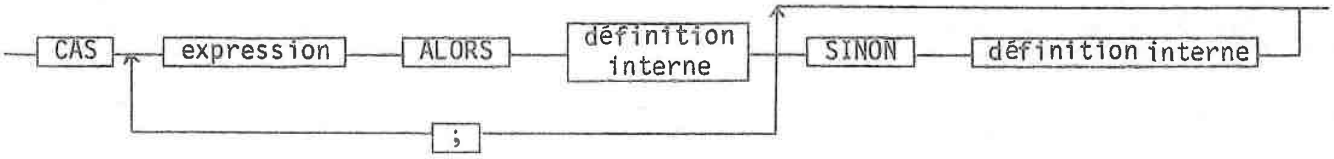
DEFINITION STRUCTUREE



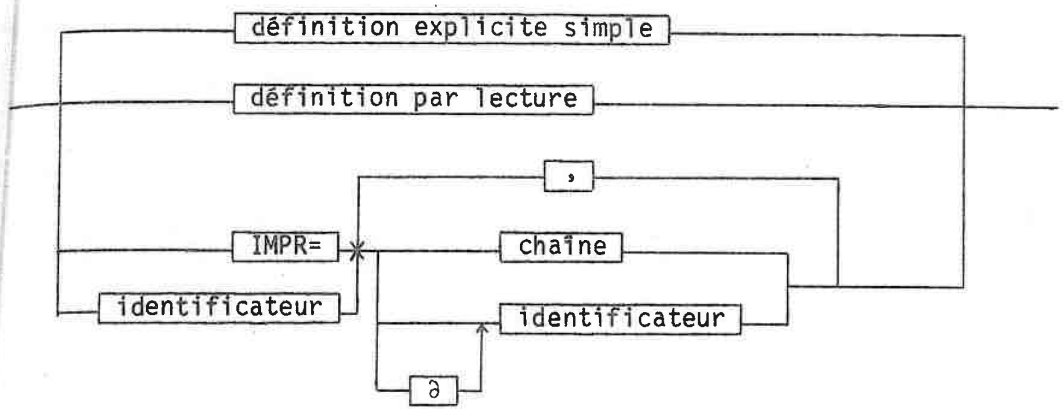
DEFINITION ITERATIVE



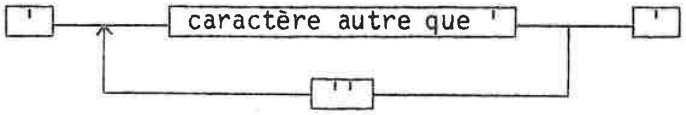
DEFINITION CONDITIONNELLE



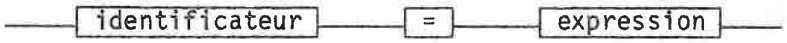
DEFINITION SIMPLE



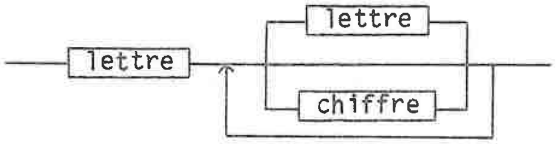
CHAINE



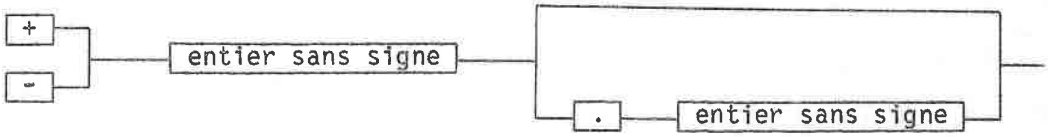
DEFINITION EXPLICITE SIMPLE



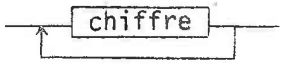
IDENTIFICATEUR SIMPLE



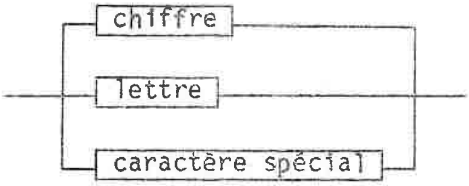
NOMBRE



ENTIER SANS SIGNE



CARACTERE

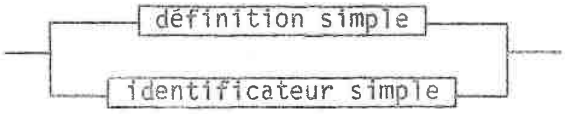


(voir 2.1. pour les caractères utilisables.)

EXPRESSION

(voir 5.4.)

DEFINITION INTERNE



annexe B

exemple de programme snoopy

```
*******  SNOOPY (VERS : 02/03/77)  *******


TYPE IMPLICITE : ENTIER
LANGAGE OBJET UTILISE : PASCAL
CODAGE DE L'ANALYSE SUR DEUX CARTES. PARAMETRES DE ZONES UTILISES :    39   80  119
ORDONNANCEMENT AUTOMATIQUE : OUI



*****************  04*05*77  ********************************


 1.   $    TRI PAR LA METHODE DE LA BULLE. ON IMPRIMERA :
 2.   $                - LES ELEMENTS TRIES
 3.   $                - LE NOMBRE DE PARCOURS DU TABLEAU
 4.   $                - LE NOMBRE DE TESTS
 5.   $                - LE NOMBRE DE PERMUTATIONS
 6.   $
 7.   -T(TAB 20) VECTEUR OU EST EFFECTUE LE  ! 1!IMPR:POUR I DE 1 JQA N REPETER          !-PARCOURS :REDEFENIT T EN LE PARCOURANT
 8.   TRI DES ELEMENTS.                      !  !       IMPR=T[I]                       !ET EN FAISANT PROGRESSER LES ELEMENTS
 9.   -NBP NOMBRE DE PARCOURS                ! 2!IMPR='NOMBRE DE PARCOURS :'.NBP       !LES PLUS GRANDS VERS LA DROITE
10.   -NBT NOMBRE DE TESTS                   ! 3!IMPR='NOMBRE DE TESTS :'.NBT          !-INI
11.   -N  NOMBRE D'ELEMENTS A TRIER          ! 4!IMPR='NOMBRE D''ECHANGES :'.NBE       !
12.                                          ! 5!T.NBE.NBT.NBP:INI!JQA FIN REPETER     !
13.   -NBE NOMBRE D'ECHANGES                 !  !          PARCOURS  < N                !
14.                                          ! 6! N = DONNEE                           !
15.   $                                       PARCOURS
16.   -Q RANG DE L'ELEMENT TRIE LE PLUS HAUT ! 1!Q.NBE.T:Q=1!POUR J DE 1 JQA AQ       !-ETAPE PERMUTE 2 ELEMENTS OU PROGRESSE
17.   -AQ ANCIENNE VALEUR DE Q               !  !       REPETER ETAPE                  !
18.                                          ! 2!NBT='NBT+'Q-1                         !
19.                                          ! 3!NBP='NBP+1                            !
20.                                          ! 4!FIN = Q=1                             !
21.                                          ! 5!AQ=Q-1                                !
22.   $                                        ETAPE
23.                                          ! 1!Q.NBE.T:SI 'T[J]>'T[J+1] ALORS CHANGER !-CHANGER PERMUTE DEUX ELEMENTS ET
24.                                          !  !                                      !MODIFIE Q
25.   $                              CHANGER     MANUEL
26.   -TRANS INTERMEDIAIRE POUR L'ECHANGE    !01!NBE='NBE+1                            !
27.   DES DEUX ELEMENTS A PERMUTER           !02!Q=J                                   !
28.                                          !05!T[J]=TRANS                            !
29.                                          !04!T[J+1]='T[J]                          !
30.                                          !03!TRANS='T[J+1]                         !
31.   $                                  INI
32.                                          ! 1! FIN = FAUX                           !
33.                                          ! 2!Q=N                                   !
34.                                          ! 3!NBT=0                                 !
35.                                          ! 4!NBP=0                                 !
36.                                          ! 5!NBE=0                                 !
37.                                          ! 6!T:POUR K DE 1 JQA N REPETER T[K]=DONNEE!
38.   $
```

```
     ORDRE DES DEFINITIONS (VERS : 14/09/76)

 MODULE  MODULE 1       6  5  1  2  3  4
 MODULE  PARCOURS       2  3  5  1  4
 MODULE  INI            1  2  3  4  5  6
 MODULE  ETAPE          1
 MODULE  CHANGER      ORDONNANCEMENT MANUEL



     SYMBOLES UTILISES DANS LE PROGRAMME

IDENTIFICATEUR(S) UTILISE(S)

        1   I           ENTIER      :    20 MOT(S)
        2   NUP         ENTIER      :     1 MOT(S)
        3   NET         ENTIER      :     1 MOT(S)
        4   N           ENTIER      :     1 MOT(S)
        5   NEF         ENTIER      :     1 MOT(S)
        6   J           ENTIER      :     1 MOT(S)
        7   FIN         BOOLEEN     :     1 MOT(S)
        8   D           ENTIER      :     1 MOT(S)
        9   M1          ENTIER      :     1 MOT(S)
       10   L           ENTIER      :     1 MOT(S)
       11   TRANS       ENTIER      :     1 MOT(S)
       12   K           ENTIER      :     1 MOT(S)



MODULE(S) UTILISE(S)

        1           MODULE 1    (NIVEAU  0)
        2           PARCOURS    (NIVEAU  1)   (INITIALISATION : INI      )   (MODULE ITERATIF)
        3           INI         (NIVEAU  1)
        4           $01 NEW     (NIVEAU  1)   (MODULE ITERATIF)
        5           ETAPE       (NIVEAU  2)   (INITIALISATION : $02 NEW )   (MODULE ITERATIF)
        6           $02 NEW     (NIVEAU  2)
        7           CHANGER     (NIVEAU  3)   (MODULE CONDITIONNEL)
        8           $03 NEW     (NIVEAU  2)   (MODULE ITERATIF)



JEU D'ESSAI

    10
     12    53    21     4    46    52    12    53    42    52
FIN DU JEU D'ESSAI


 SYNTAXE CORRECTE.
```

```
      TRADUCTION PASCAL (VERS : 26/01/77)


***    1- PROGRAM SNOOPY;
***    2- VAR
***    3-     DONNEE : TEXT;
***    4-     NBP      ,NBT      ,N        ,NBE      ,I        ,Q        ,AQ       ,
***    5-     J        ,TRANS    ,K          : INTEGER;
***    6-     FIN        : BOOLEAN;
***    7-     T          : ARRAY[1.. 20] OF INTEGER;
***    8-
***    9- PROCEDURE READALFA(VAR NOM:ALFA);
***   10-   VAR I:INTEGER;T:ARRAY[1..8] OF CHAR;
***   11-   BEGIN
***   12-   REPEAT
***   13-     READ(DONNEE,T[1])
***   14-   UNTIL (T[1]<>' ');
***   15-   FOR I:=2 TO 8 DO READ(DONNEE,T[I]);
***   16-   PACK(T,1,NOM)
***   17-   END;{READALFA};
***   18-
***   19- BEGIN
***   20-   RESET(DONNEE);
***   21-   READ(DONNEE,N );
***   22-   FIN := FALSE;
***   23-   Q:=N;
***   24-   NBT:=0;
***   25-   NBP:=0;
***   26-   NBE:=0;
***   27-   FOR K        :=1 TO N DO
***   28-     READ(DONNEE,T[K]);
***   29-   WHILE (NOT FIN      ) DO BEGIN
***   30-     NBT:=NBT+Q-1;
***   31-     NBP:=NBP+1;
***   32-     AQ:=Q-1;
***   33-     Q:=1;
***   34-     FOR J        :=1 TO AQ DO BEGIN
***   35-       IF T[J]>T[J+1]  THEN BEGIN
***   36-         NBE:=NBE+1;
***   37-         Q:=J;
***   38-         TRANS:=T[J+1];
***   39-         T[J+1]:=T[J];
***   40-         T[J]:=TRANS
***   41-                                 END {THEN};
***   42-                                  END {FOR};
***   43-     FIN := Q=1
***   44-                               END {WHILE};
***   45-   FOR I        :=1 TO N DO BEGIN
***   46-     WRITE(T[I]);
***   47-     WRITELN
***   48-                                 END {FOR};
***   49-   WRITE('NOMBRE DE PARCOURS :',NBP);
***   50-   WRITELN;
***   51-   WRITE('NOMBRE DE TESTS :',NBT);
***   52-   WRITELN;
***   53-   WRITE('NOMBRE D''ECHANGES :',NBE);
***   54-   WRITELN;
***   55- END.


******** TEMPS UTILISE :    4.254 S.


- END PASCAL
JOB STEP 01 TERMINATED AT  11*54*25* AFTER 0000.07 MIN
.       CORE USED  0070     DISC USED  0013   WAIT TIME 0000.28
*****************************************************************************
!EXEC TABLEP
- SEER/PASCAL-SYSTEM,VERS. 1/09/75
```

```
- SFER/PASCAL-COMPILER.VERS. 1/09/75

- END COMPILATION



- BINARY OBJECT CODE WRITTEN ON GO FILE
- END PASCAL
JOB STEP 02 TERMINATED AT  11.54.43. AFTER 0000.03 MIN
.       CORE USED  0072      DISC USED  0019    WAIT TIME 0000.08
*****************************************************************************
V1001 1140903 03/009/065



(UNSAT.LIB)



JOB STEP 03 TERMINATED AT  11.56.14. AFTER 0000.04 MIN
.       CORE USED  0049      DISC USED  0016    WAIT TIME 0000.00
*****************************************************************************
- SFER/PASCAL-SYSTEM.VERS. 1/09/75

           4
          12
          12
          21
          42
          46
          50
          52
          52
          53
NOMBRE DE PARCOURS :          5
NOMBRE DE TESTS :          32
NOMBRE D'ECHANGES :          14
- END PASCAL
JOB STEP 04 TERMINATED AT  11.56.34. AFTER 0000.00 MIN
.       CORE USED  0070      DISC USED  0014    WAIT TIME 0000.14
*****************************************************************************
     TIME        TIME*CORE      CORE-USE       TIME*DISC      DISC-USE
    00.16          11.25            93%          04.82           34%

      DS          DS              SHR          SHR           CR          CP          LP
  I/O-BYTES   I/O-CALLS      I/O-BYTES   I/O-CALLS     CARDS       CARDS       PAGES
     70435A         297             00          00        82          00          09
SELF JOB TERMINATED SNOOPY        IUCA  11.56.36.7
```

annexe C

exemple de

session s.a.c.a.d. commentée

```
! SACAD
OPTION ? ! L (CR)                                    (1)
MODULE COURANT : MODULE PRINCIPAL
DESCRIPTION DES RESULTATS EXTERNES ?
LISTE : LISTE DE COUPLES (NOM, SALAIRE) % (CR)
LIGNE : MASSE SALARIALE % (CR)
% (CR)
DEFINITION 1 (LISTE, LIGNE)
LISTE % (CR)
LISTE DE COUPLES (NOM, SALAIRE) (RESULTAT)
LISTE : INI ; POUR I DE 1 JQA N REPETER IMPRIME % (CR)
DECRIRE N
ENT, NOMBRE D'EMPLOYES % (CR)
DECRIRE LE MODULE IMPRIME
IMPRIME UN NOM ET UN SALAIRE % (CR)
DESCRIPTION DES RESULTATS EXTERNES DE IMPRIME        (2)
UNE LIGNE : UN NOM ET UN SALAIRE % (CR)
% (CR)
DECRIRE LE MODULE INI
% (CR)
DEFINITION 2 (N, LIGNE)
LIGNE % (CR)
MASSE SALARIALE (RESULTAT)
LIGNE = 'MASSE SALARIALE' , MS % (CR)
REEL, MASSE SALARIALE DE L'ENTREPRISE % (CR)
DEFINITION 3 (N, MS)
N % (CR)
NOMBRE D'EMPLOYES (ENT)
N = DONNEE % (CR)
DEFINITION 4 (MS)
MS % (CR)
MASSE SALARIALE DE L'ENTREPRISE (REEL)
MS : FUSION 1 % (CR)                                 (3)
1 LIGNE , MS : INI ; POUR I DE 1 JQA N REPETER IMPRIME
DESCRIPTION COMPLEMENTAIRE DE IMPRIME
```

```
ET CALCULE  MS  % (CR)
DESCRIPTION COMPLEMENTAIRE DE INI
% (CR)
! LIGNE, MS  :  INI ; POUR  I DE !  JQA  N  REPETER  IMPRIME
2 LIGNE  =  'MASSE SALARIALE' , MS
3 N  =  DONNEE
CHOISIR UN MODULE COURANT (IMPRIME, INI)
IMPRIME  % (CR)
IMPRIME UN NOM ET UN SALAIRE ET CALCULE  MS
DEFINITION  1  (UNE LIGNE, MS)
UNE LIGNE  % (CR)
UN NOM ET UN SALAIRE (RESULTAT)
UNE LIGNE = NOM, SALAIRE  % (CR)
DECRIRE  NOM
CHAINE, NOM DE L'EMPLOYE  % (CR)
DECRIRE, SALAIRE
REEL, SALAIRE DE L'EMPLOYE  % (CR)
DEFINITION  2  (MS, NOM, SALAIRE)
≠ MON  (CR)                                        (4)
TACHE ?
MODI  (CR)
S/IMPRIME/M/IMPRETCAL/  (CR)                       (5)
CONS  (CR)
MODULE  COURANT  IMPRECAL
IMPRIME UN NOM ET UN SALAIRE ET CALCULE MS
DEFINITION  2  (MS, NOM, SALAIRE)
NOM  % (CR)
NOM DE L'EMPLOYE  (CHAINE)
NOM  =  DONNEE % (CR)
DEFINITION  3  (MS, SALAIRE)
SALAIRE  % (CR)
SALAIRE DE L'EMPLOYE
SALAIRE  = SH  *  NH  % (CR)
DECRIRE  SH
REEL, SALAIRE  HORAIRE  (CR)
```

```
DECRIRE NH
ENT, NOMBRE D'HEURES % (CR)
DEFINITION 4 (MS, SH, NH)
MS % (CR)
MASSE SALARIALE (REEL)
MS = @ MS + SALAIRE % (CR)
ATTENTION MS DOIT ETRE INITIALISE DANS INI          (6)
≠ ADM (CR)                                          (7)
MODULE ?
INI % (CR)
DESCRIPTION COMPLEMENTAIRE DE INI
INITIALISATION DE MS % (CR)
DEFINITION 5 (SH, NH)
SH % (CR)
SALAIRE HORAIRE (REEL)
SH = DONNEE % (CR)
DEFINITION 6 (NH)
NH % (CR)
NOMBRE D'HEURES (ENT)
NH = DONNEE % (CR)
1 UNE LIGNE = NOM, SALAIRE
2 NOM = DONNEE
3 SALAIRE = SH * NH
4 MS = @ MS + SALAIRE
5 SH = DONNEE
6 NH = DONNEE
CHOISIR UN MODULE COURANT (INI)
≠ MON (CR)
TACHE ?
MODI (CR)
F/IMPRETCAL/2, 6 (CR)                               (8)
2 NOM, NH = DONNEE
R/IMPRETCAL/5/INI/ (CR)                             (9)
DEFINITION 5 DE IMPRECAL REMONTEE DANS INI
ORDO (CR)                                           (10)
```

```
/IMPRETCAL/ A  CR
ORDRE 2 3 1 4
A  CR
ORDRE 3 1 2
CONS  CR
CHOISIR UN MODULE COURANT  (INI)
INI  %  CR
INITIALISATION  DE  MS
DEFINITION 2 (MS)
MS  %  CR
MASSE SALARIALE  (REEL)
MS  =  0.0  %  CR
1  SH  =  DONNEE
2  MS  =  0.0
ANALYSE TERMINEE
MODULE  NON ORDONNES  :  INI
≠ MON  CR
ORDO  CR
/INI/M  CR
ORDRE  ?
1  2  CR
TRAD  CR
PASC  CR
TRADUCTION  PASCAL  RECOPIEE SUR OBJET
VISU  CR
D 0, 0  CR                                                    (11)
1  LISTE, MS : INI ; POUR I DE 1 JQA N REPETER IMPRETCAL
2  LIGNE  = 'MASSE SALARIALE' , MS
3  N =  DONNEE
Ø  0, 0  CR
1  N  =  DONNEE
2  LISTE, MS  : INI ; POUR I DE 1 JQA N REPETER IMPRECAL
3  LIGNE = 'MASSE SALARIALE' , MS
FIN  CR
```

ANALYSE RECOPIEE SUR SOURCE

NOTES EXPLICATIVES

(1) Choix de l'option LIBRE (cf. 5.1)

(2) Le système demande de décrire les résultats externes de IMPRIME car c'est un module itératif (cf. 5.2.2.2)

(3) MS est défini par la même définition que LISTE

(4) passage sous contrôle du moniteur (tâche de modification)

(5) remplacement de l'identificateur de module IMPRIME par IMPRETCAL

(6) MS est défini par récurrence

(7) modification de la description de INI

(8) fusion de deux définitions

(9) remontée d'une définition de IMPRETCAL dans INI

(10) tâche d'ordonnancement

(11) impression de toutes les définitions du module principal

(12) impression de toutes les définitions ordonnées du module principal

annexe D

algorithmes utilisés

par le compilateur snoopy

- ordonnée (bool) vrai si le module peut être ordonné. - ordre (ent tab n) numéro des définitions dans le bon ordre. - n (ent) nb de définitions	2 1	ordonnée, ordre : INI ; pour i de 1 jqa n répéter T n = donnée	- T : définit ordre [i] et vérifie si la ième définition est compatible avec les autres - INI
		T	
- droite (ensemble tab n) - gauche (ensemble tab n) - vdroite (ensemble tab n) } (cf. texte) bon (bool) : vrai si la définition j est la ième dans l'ordre. j (ent) rang de la ième définition	2 3 1	ordonnée = ∂ ordonnée et bon ordre, droite, vdroite : si bon alors MODIF$j bon, j : INIT ; jqa fin répéter TEST $∂droite, gauche, ∂vdroite	- TEST : définit bon suivant que la définition j peut être utilisée ou pas (fin qd bon ou j = n) - MODIF : définit ordre [i] et modifie droite et vdroite - INIT
		TEST	
- old (ensemble) de tous les ∂ (remonté dans INIT)	2 1 3	bon = (droite [j] = ∅) et (gauche [j] ∩ old = ∅) j = ∂ j + 1 fin = bon ou (j = k)	
		INIT	
	1 2 3 4	bon = faux fin = faux j = 0 old = $\bigcup_{i=1}^{n}$ vdroite [i]	

	MODIF	
1	ordre [i] = j	
2	droite : pour k de 1 jqa n répéter droite [k] = ∂droite [k] - gauche [j]	
3	vdroite [j] = ∅	
	INI	
1	ordonnée = vrai	
2	droite, gauche, vdroite = donnée	

- analyse (résultat)	1	analyse : INI ; jqa finanalyse répéter MOD	- MOD : analyse un module - INI : initialise le compilateur
		MOD	
- nd (ent) nombre de définitions du module - définitions (fichier) où les blancs sont compactés - analex (résultat) analyse des lexiques - anadef (résultat) analyse des définitions	1	nd, définitions, analex : IN1 ; jqa finmodule répéter LIGNE	- IN1 : initialise l'analyse des lexiques - LIGNE : analyse une ligne - IN2 : initialise l'analyse des définitions - DEF : analyse une définition
	2	anadef : IN2 ; pour i de 1 jqa nd répéter DEF $ analex, définitions, nd	
	3	finanalyse = {fin du paquet ou données}	
		LIGNE	
	1	finmodule, analex, nd, définitions : si {entête de module, données ou fin} alors finmodule = vrai ; {non commentaire} alors NORMAL	- NORMAL : analyse une ligne normale
		NORMAL	
	1	{analyser le lexique gauche}	
	2	si nod ≠ {blanc} alors {nouvelle définition} et nd = ∂nd + 1	
	3	{compacter les blancs et ranger un morceau de définition}	
	4	{analyser le lexique droit}	
		DEF-INI-IN1-IN2	
		(cf. lexique)	

bibliographie

[1] J.E. SAMMET — *Roster of Programming Languages for 1973, SIGPLAN Vol. 9 n° 11 (1974)*

[2] R.E. BROWN — *Toward a better language for Structured Programming, SIGPLAN Vol. 11 n° 7 (1976)*

[3] D. SALOMON — *A design for FORTRAN to facilitate Structured Programming, SIGPLAN Vol. 12 n° 1 (1977)*

[4] B.W. LAMPSON, JJ.HORRING, R.L. LONDON, J.G. MITCHELL, G.L. POPOK — *Report on the Programming Language EUCLID, SIGPLAN Vol. 12 n° 2 (1977)*

[5] A.I. FORSYTHE, T.A. KEENAN, E.I. ORGANICK, W. STENBERG — *Introduction à la technique de l'ordinateur Masson (1974)*

[6] J.D. WARNIER — *Les Procédures de traitement et leurs données, les éditions d'organisation (1973)*

[7] O. LECARME — *Structured Programming, Programming teaching and the language PASCAL, SIGPLAN Vol. 9 n° 7 (1974)*

[8] N. WIRTH — *Systematic Programming : an introduction, Prentice Hall (1973)*

[9] G.R. FOULK — *The Do-trace : a simple and efficient Method for debugging goto free programs, SIGPLAN Vol. 10 n° 9 (1975)*

[10] — *Procedings of the International Conference on Reliable Software, SIGPLAN Vol. 10 n°6 (1975)*

[11] M. FRADIN, C. LETANG — *Introduction à l'algorithmique, Technique et Vulgarisation (1973)*

[12] C. PAIR — *Du problème à sa solution, Exposé fait aux journées IRIA Logique et Programmation (19-20 Nov. 1975)*

[13] B. HUC, F. BELLEGARDE — *Une méthode de programmation déductive, son implémentation, ses extensions, Communication faite aux journées du Sous-groupe AFCET Langages et Programmation (10-11 déc. 1976)*

[14] C. PAIR, J. MAROLDT — *Introduction à une méthode de programmation déductive (INPL Sept. 1975)*

[15] B. HUC — *Enseignement de l'Algorithmique par une méthode déductive, AFCET Informatique et Enseignement Vol. 2 n° 2 (Juin 1976)*

[16] D.E. KNUTH — *Structured Programming with GOTO Statements, ACM Computing Survey Vol 6 n°4 (1974)*

[17] C. PAIR — *Les itérations avec condition d'arrêt en programmation déductive, A tout CRIN n°1 (Sept 1976) (publication locale à Nancy)*

[18] F.R.A. HOPGOOD — *Techniques de compilation, Dunod (1970)*

[19] C. PAIR — *Cours de compilation (Université de Nancy I)*

[20] J. ARSAC — *Nouvelle leçon de Programmation, (à paraître), Dunod (1977)*

[21] E.W. DIJKSTRA — *A discipline of programming, Prentice-Hall (1976)*

[22] M. GRIFFITHS — *Program production by successive transformation, lecture notes in computer science, Vol 46, Springer Verlag (1976)*

[23] D.E. KNUTH *Fundamental Algorithms, The Art of Computer programming Vol 1, Addison Wesley (1969)*

[24] J.C. DERNIAME, C. PAIR *Problèmes de cheminement dans les graphes, Dunod (1971)*

NOM DE L'ETUDIANT : HUC Bernard

NATURE DE LA THESE : DOCTEUR INGENIEUR

VU, APPROUVE

& PERMIS D'IMPRIMER

NANCY, le 13 mai 1977

LE PRESIDENT DE L'UNIVERSITE DE NANCY I

M. BOULANGE

ERRATA

page I-4 **lire** Algol 60

page I-25 figure I2 **lire** terme précedent U_{i-1} | U ou @ U

page I-I3 figure 4f **lire** 70 Moy ← S + I

page I-26 figure I3 **lire** Somme = $\overline{\text{Somme}}$ + N

page I-28 figure I4 **lire** S= $\overline{S}$ + note **et** N = $\overline{N}$ + 1

page III-3 §322I **lire** En plus du nom et du type de chaque identificateur de résultat (intermédiaire ou final), le lexique doit contenir un certain nombre de renseignements...

page IV-5 figure 3 **lire** **sinon** **impr** = ' q nul ' $ p

page IV-I5 §444 ligne 3 **lire** (cf 336)

page IV-I5 §445 ligne 2 **lire** (cf 337)

page V-5 figure 3 ligne 3 **lire** 2 : X = DONNEE % (CR)

page VI-4 figure 1b **lire** | POUR | 3 |