5 33

Dactyl
Sc. N. 74/4 A (1)

Université de NANCY I
U.E.R. Mathématiques

Sc
N
1

LE PROJET CIVA

UN SYSTEME DE PROGRAMMATION MODULAIRE

TOME 1

UNIVERSITE NANCY BIBLIOTHEQUE

Thèse soutenue le 25 Janvier 1974

par

JEAN CLAUDE DERNIAME

pour obtenir le grade de Docteur d'Etat en Sciences Mathématiques
devant la commission d'examen :

M. J. LEGRAS — Président

M. J. ARSAC
M. J. CEA
M. M. DEPAIX
M. C. PAIR
— Examinateurs

74
4 A1

Université de NANCY I
U.E.R. Mathématiques

LE PROJET CIVA

UN SYSTEME DE PROGRAMMATION MODULAIRE

TOME 1

Thèse soutenue le 25 Janvier 1974

par

JEAN CLAUDE DERNIAME

pour obtenir le grade de Docteur d'Etat en Sciences Mathématiques
devant la commission d'examen :

M. J. LEGRAS	Président
M. J. ARSAC M. J. CEA M. M. DEPAIX M. C. PAIR	Examinateurs

Je remercie Monsieur le Professeur LEGRAS qui m'a prodigué de nombreux enseignements, s'est intéressé dès le début à mes travaux et a accepté de présider le Jury.

Monsieur le Professeur C. PAIR, Directeur de l'Institut Universitaire de Calcul Automatique de NANCY, est à l'origine du projet CIVA. Je lui exprime ma profonde reconnaissance pour la bienveillance et l'intérêt avec lesquels il a guidé mes recherches depuis mon D. E. A. jusqu'à cette thèse, pour l'aide qu'il n'a cessé de m'apporter et l'attention avec laquelle il a suivi le travail de notre groupe, assurant ainsi notre formation de chercheurs.

Je remercie Messieurs les Professeurs J. ARSAC, Directeur de l'Institut de Programmation de l'Université PARIS 6, J. CEA, Professeur à l'Université de NICE et M. DEPAIX, Directeur de l'U. E. R. de Mathématiques de l'Université NANCY 1, pour l'intérêt qu'ils ont porté à ce projet et pour l'honneur qu'ils me font à participer au Jury.

La définition et la réalisation du projet CIVA sont le fruit du travail d'une équipe que j'ai animée depuis octobre 1970. Mon rôle y a été essentiellement de proposer des travaux, de guider des recherches, d'aider à des rédactions et d'assurer la cohérence des idées proposées. Mais ce travail n'aurait pas pu exister sans la participation active de tous les membres du groupe que je tiens à remercier vivement pour leur appui. On trouvera, au long des chapitres qui suivent, de nombreuses références aux divers travaux des membres d'un groupe dont je ne suis ici qu'un porte-parole.

Une partie de ce travail a été réalisée dans le cadre d'un contrat du Comité de Recherches en Informatique du 1er octobre 1972 au 31 mars 1974 (contrat C. R. I. n° 73 004).

SOMMAIRE

PREMIERE PARTIE : PRESENTATION DU PROJET.

INTRODUCTION.

1. MODULARITE.

2. LE MODE DECLARATIF.

3. CONCEPTION DE LA REALISATION.

DEUXIEME PARTIE : DESCRIPTION DES CONSTITUANTS.

4. DESCRIPTION DES INFORMATIONS.

5. TABLES DE DECISION.

6. INSTRUCTIONS SIMPLES.

7. TRAITEMENT GLOBAL DES FILES ET DES ENSEMBLES.

8. INSTRUCTIONS ITERATIVES ET CINEMATIQUE DES FICHIERS.

9. LE METALANGAGE CIVA.

10. LES MODULES DE COMMANDE.

TROISIEME PARTIE : UTILISATION.

11. ACQUISITION DES DONNEES ET TRANSFERT DES RESULTATS.

12. ACQUISITION.

13. EDITION.

14. AUTRES OPERATIONS DE SERVICES : UNION, EXTRACTION, INTERROGATION.

15. EXEMPLES D'APPLICATION.

QUATRIEME PARTIE : REALISATION.

16. ORGANISATION GENERALE DE L'APPLICATION "REALISATION".

17. LA CODIFICATION.

18. LA TRADUCTION DES TABLES DE DECISION.

19. TRADUCTION DES EXPRESSIONS ARITHMETIQUES ET DES CONTROLES.

20. INTERPRETATION DES MODULES DE COMMANDE.

21. IMPLANTATION DES FILES ET DES ENSEMBLES.

22. ENTREES SORTIES IMPLICITES ET EXPLICITES.

23. TRADUCTION DES INSTRUCTIONS D'ITERATION.

24. EVALUATION DES INSTRUCTIONS D'ITERATION LOGIQUE.

25. LA RELIURE.

CONCLUSION.

ERRATA

Paragraphe 1.6. page 25

a) avant dernière ligne

les objets détruits ⟶ les objets déclarés.

Paragraphe 1.8. page 26 ligne 11

pour chaque élément du module de commande, Mo...

Paragraphe 2.42. page 7 ligne 3

du module ASKFICH ;

SELECT 1 <u>comme</u> FICH ; ⟶ SELECT 1 <u>type</u> FICH ;

ligne 6

SELECT 3 <u>comme</u> FICH ; ⟶ SELECT 3 <u>type</u> FICH ;

Paragraphe 6.5. page 12 ligne 1

(m2 - m1) x m 3 > 0 ⟶ (m2 - I) x m 3 > 0

Paragraphe 15.13. page 5 ligne 1

<u>type</u> <u>struct</u>... ⟶ <u>type</u> assuré <u>struct</u>.

PREMIERE PARTIE :

PRESENTATION DU PROJET

- Introduction.
- Chapitre 1 - Modularité.
- Chapitre 2 - Le mode déclaratif.
- Chapitre 3 - Conception de la réalisation .

INTRODUCTION.

Les difficultés de conception et de réalisation d'un traitement automatique de l'information sont bien connues, et de nombreuses méthodes d'aide à l'analyse et d'aide à la programmation se proposent de contribuer à les surmonter. Or ces difficultés ne croissent pas proportionnellement à la taille des programmes ni à l'importance des structures d'information qu'on peut être amené à décrire, elles croissent beaucoup plus vite. D'où l'idée naturelle de décomposer la réalisation en petites unités. On est alors ramené à deux problèmes :

- décrire la réalisation de ces parties
- assembler les différents programmes ainsi obtenus.

Pour simplifier le problème de l'assemblage des constituants, il est préférable de penser à des unités indépendantes, nous dirons des modules. On a alors réduit la complexité globale de l'opération. La programmation modulaire offre d'autres avantages :

il est possible de réaliser un module en ne tenant qu'assez peu compte de la réalisation des autres et, en particulier, il est possible de limiter l'impact d'une modification dans un module à ce module lui-même.

Cette idée n'est pas nouvelle, l'efficacité de la décomposition d'un programme en modules est connue depuis les réalisations de l'O. S. 360. Ses avantages :

- simplification de la conception ;
- facilité de modification du programme par remplacement de modules ;
- accélération de l'implantation

ont déjà été montrés (DIJKSTRA, 72), (NATO, 68-69), (PARNAS, 72) et l'expérience les prouve. C'est d'ailleurs un principe tout à fait général, que DESCARTES énonçait dans le Discours de la Méthode (LEYDE 1637) "diviser chacune des difficultés que j'examinerais en autant de parcelles qu'il se pourrait et qu'il serait requis pour les mieux résoudre".

Pour assurer l'indépendance des modules, ceux-ci doivent vérifier deux règles :

- les interfaces entre modules doivent être rédigés en suivant des conventions établies selon un schéma unique ;
- les modules travaillent ensemble doivent utiliser une représentation cohérente pour toutes les informations qu'ils échangent.

Pour garantir l'unicité du schéma de convention entre les modules il est préférable de disposer d'un langage contenant cette notion de module et prévoyant leurs liaisons.

En l'absence d'outil adapté à une description modulaire des traitements, de nombreux auteurs se sont posés la question de déterminer la taille optimale des modules : suffisamment grands pour qu'ils ne posent pas trop de problèmes d'assemblage, suffisamment petits pour que leur réalisation puisse être assurée par une seule personne (CROCUS, 73).

Par contre, si nous disposons d'un tel langage cette question tombe et nous ferons des modules aussi petits que les nécessités d'analyse l'imposeront. Car les modules ne sont pas seulement un outil de description de la programmation facilitant la maintenance des programmes mais aussi un concept d'analyse. Nous entrevoyons donc, par l'intermédiaire de la modularité, la possibilité d'unifier les concepts utilisés pour décrire l'analyse, la programmation et la maintenance d'un travail. C'est ce que nous avons tenté de faire.

Notre projet vise à donner à l'utilisateur la possibilité d'employer une formulation unique, c'est-à-dire un vocabulaire commun, mais aussi une présentation commune, pour représenter les différentes phases de la mise en oeuvre d'un travail : la conception, le compte-rendu d'analyse par la description des actions à entreprendre, la programmation, l'exécution et enfin la maintenance.

Notre but est donc de fournir un ensemble d'outils permettant de passer, sans heurt, de l'analyse à l'exploitation. Il s'agit essentiellement de définir un langage unique permettant de décrire à la fois les résultats de l'analyse, les programmes et leur exploitation et de construire un système (traducteur et ensemble de services) acceptant ce langage.

Nous nous plaçons dans le cas d'applications importantes pour lesquelles la notion de programme autonome disparaît au profit de celle de chaîne d'exploitation, et dont la durée de vie n'est pas très courte. Ce qui signifie que le passage de la conception à l'exécution pourra être relativement long, mais que cette exécution, certainement répétitive, devra être rapide. Il ne s'agit donc pas de faciliter la mise en oeuvre d'un programme pris isolèment, auquel cas les outils présentés seraient beaucoup trop lourds.

Notons cependant que nous avons pu constater le grand intérêt, sur la plan pédagogique, d'une telle présentation pendant l'apprentissage de la programmation.

En effet, nos propositions tendent à simplifier la conception d'une application et deux cas peuvent relever de notre tentative :

- la réalisation de "gros" programmes, tels que ceux rencontrés dans la gestion d'une entreprise, ou lors de la conception d'un système d'exploitation ;

- la réalisation de "petits" programmes proposés en travaux pratiques d'analyse ou de programmation pour entraîner les étudiants à concevoir un programme ou une application. On pourra voir, en annexe, un tel exemple d'utilisation.

Actuellement, aucun langage véritablement non ambigü n'est utilisé pour décrire les résultats de l'analyse. Ce qui a un certain nombre de conséquences fâcheuses :

- la description est floue, difficile à lire, difficile à comprendre ;

- elle n'apporte aucune aide conceptuelle : il est bien connu qu'un langage oriente fortement la pensée, au point qu'il n'existerait vraisemblablement aucune pensée sans langage ;

- il est difficile d'assurer la maintenance des travaux ou l'emploi de l'existant.

Lorsqu'un langage est utilisé pour cette description des résultats de l'analyse, il est différent de ceux utilisés dans les phases antérieures (de la conception à l'analyse détaillée), et de ceux utilisés pour la programmation et pour l'exploitation (langage de commande). Inversement, les langages de programmation ne possèdent guère d'outils adaptés à la conception et à l'analyse. Cette hétérogénéité des formulations crée des discontinuités et constitue un obstacle majeur à l'efficacité des communications entre les personnes intervenant à chaque étape. De plus, il est nécessaire de reprendre plusieurs fois les mêmes problèmes : ce qui a été fait à une étape devra être transformé à la phase suivante (par ex. : réorganisation presque toujours nécessaire avant la programmation).

Il nous semble donc important de développer, dès l'analyse, l'état d'esprit informatique, qui consiste à tout préciser de manière formelle, donc dans un langage, de manière à ne rien laisser à l'interprétation sémantique. Important pour les rapports entre les hommes, mais aussi parce que cela permettra de faire intervenir la machine plus tôt :

- en lui confiant directement les résultats de l'analyse ;

- en automatisant certaines tâches (regroupement, lancement, préparation des textes, etc....).

Pour permettre à l'analyste de décrire son dossier en précisant toutes les opérations à effectuer ou toutes les informations à traiter, il est souhaitable que le langage qu'il utilisera puisse refléter l'organisation de ces opérations et de ces informations. Or celle-ci ne s'ébauche que progressivement au cours de l'analyse. C'est donc au fur et à mesure de l'analyse qu'il faut pouvoir donner des précisions, en plusieurs couches successives. D'où l'idée naturelle de décomposer les opérations et les informations en autant d'éléments que nécessaire et de les nommer pour pouvoir les préciser ultérieurement, c'est-à-dire l'idée de modularité. S'appuyant sur cette idée de modularité, on peut utiliser deux méthodes de conception de la solution, deux méthodes d'analyse :

- une méthode de conception descendante (Wirth N. 71) qui est celle que nous venons de décrire : "on part du résultat que l'on souhaite obtenir et on définit par étapes une solution conduisant à ce résultat ; à chaque étape, certaines fonctions sont complètement définies alors que l'implantation des autres reste floue et sera précisée dans une étape ultérieure" (CROCUS , 73, chap. 7.31)

- une méthode de conception ascendante , dans laquelle on s'appuie sur l'ensemble des modules existants, qu'on étend pour obtenir un ensemble de plus en plus proche de la solution visée.

Dans la pratique, on utilise souvent alternativement l'une et l'autre méthode.

Modularité et méthode de conception descendante ont aussi pour avantage de permettre de préciser les actions ou les informations le plus tard possible. Ce sera là une autre idée directrice essentielle de notre travail : "éviter le plus longtemps possible à l'utilisateur d'avoir à contrôler le détail des opérations à entreprendre ou des informations à utiliser" et elle nous conduit à développer notre projet dans deux directions :

- permettre une description modulaire des actions et des informations ;

- permettre l'utilisation intensive du "mode déclaratif" dans lequel l'utilisateur n'a pas à connaître le détail de la programmation (des services par exemple) et permettre surtout la rédaction, par l'utilisateur lui-même, d'objets en mode déclaratif.

La modularité de la description des actions utilisée comme outil de conception, les instructions synthétiques, que ce soient les instructions d'itération permettant une manipulation globale des files d'objets ou les tables de décisions permettant une description globale des décisions à

prendre, nous amènent à éliminer complètement les branchements du langage. Les mérites de la programmation sans branchement ont déjà été démontrés (Arsac J. 71), (Arsac J. 72), (Dijkstra E. W. 72), (Knuth D. E. 71). Nous retiendrons surtout le fait que la pensée est un phénomène essentiellement linéaire et que, devant cette constatation, l'existence d'organigrammes complexes est une aberration.

La méthode de conception descendante est elle-même liée à cette nature linéaire de la pensée : c'est pour ne pas détourner notre attention du cours qu'elle suit dans la description d'une action que, dès qu'on peut caractériser une action plus simple par ses relations avec l'environnement, c'est-à-dire par ses spécifications, on ordonne l'exécution de cette action par son nom uniquement (appel de module), quitte à préciser plus tard la description exacte de cette action (écriture du module) (Parnas D. L. 72).

Une troisième idée, peut-être moins fondamentale, sous-tend également ce projet : elle consiste à dire que toutes les valeurs traitées dans une chaîne de traitement sont présentées en "format interne", c'est-à-dire sous la forme propice aux traitements, et non en "format externe", c'est-à-dire sous la forme propice à leur lecture par l'utilisateur. Ce qui implique que tout fichier doit subir, avant de pouvoir être traité, une opération de conversion appelée "acquisition", et après un traitement, avant de pouvoir être lu, une opération de conversion appelée "édition". Ceci présente de gros avantages tant du point de vue de l'analyse que de l'exécution d'une chaîne de traitement.

Pour l'analyse, toutes les opérations de contrôle de validité des données peuvent être ainsi groupées en une seule étape : le contrôle à l'entrée en ordinateur. Les types utilisés pour décrire les informations à traiter ne contiennent aucune des caractéristiques du format externe.

Quant à l'exécution, elle est considérablement accélérée du fait qu'il n'y a plus que très peu de conversions à faire au cours d'un traitement, alors qu'un programme Cobol, par exemple, perd le plus clair de son temps à convertir et reconvertir des valeurs.

Une présentation complète du projet doit donc commencer par présenter les idées générales qui ont présidé à sa conception. Nous avons, pour cela, essayé de minimiser le nombre de notions indépendantes, ce qui permet une description formelle où on ramène toutes les notions à un petit nombre (cf. chap. 1). Notre travail est construit autour de deux idées essentielles :

- modularité, ce qui nous amène à introduire la notion de "module" - unité de description modulaire d'actions, la notion de "classe" - unité de description modulaire d'informations - et les relations pouvant exister entre elles (ch. 1)

- mode déclaratif, qui permet à l'utilisateur d'éviter le plus longtemps possible d'avoir à contrôler le détail des opérations à entreprendre ou des informations à traiter. Il se traduit, dans le choix des instructions, par le choix d'instructions globales et, dans l'ensemble du projet, par l'introduction de la notion de métamodule et de métalangage pour décrire les métamodules.

La seconde partie du travail reprend la présentation générale de la première partie et présente une étude détaillée des outils proposés pour la description des actions et pour celle des informations. Nous y abordons successivement chaque type d'objet et chaque instruction du langage puis du métalangage.

La troisième partie est consacrée aux problèmes de mise en oeuvre des outils précédemment rencontrés : problèmes d'utilisation, rédaction de certains métamodules d'emploi courant, liens avec une méthode d'analyse et enfin exemples d'application.

Les mêmes deux idées essentielles nous ont guidé également pour mettre sur pied une réalisation du projet qui est décrite dans la quatrième partie. La réalisation, plus encore que la conception du projet, est le fruit d'un travail d'équipe qui a déjà été décrit par ailleurs (Aubry B. 73), (Benamghar L. 73), (Chabrier J. J. 73), (Dendien J. 73), (Ducloy J. 73), (Payafar M. 71), (Perrot D. 73). Nous nous contenterons donc d'indiquer ici les problèmes particuliers qui ont pu être rencontrés et les solutions qui leur ont été apportées, en insistant sur celles dont le champ d'application dépasse le cadre de ce projet.

CHAPITRE 1

MODULARITE

1.1 Modularité de la description des actions : modules.

1.2 Modularité de la description des informations : classes.

1.3 Notion d'application. Classe d'application et classe système.

1.4 Conditions de validité des identificateurs.

1.5 Durée de vie des objets et modification de la fonction "désigne".

1.6 Exécution d'un module m appelé par m'.

1.7 Modification de l'ensemble des classes et des modules intérieurs au système.

1.8 Module de commande.

1.9 Interprétation du module de commande.

1.10 Schéma général.

1.11 Emploi, dans une application, d'objets extérieurs à l'application.

CHAPITRE I

MODULARITE

Notre projet propose donc une description modulaire des actions (par des modules), mais aussi des informations (par des classes), ce qui est au moins aussi important (1). En effet, si l'analyste peut suivre le fil de sa pensée, qui est linéaire, pour exprimer les actions à entreprendre, dans l'ordre où les besoins s'en font sentir, il doit pouvoir le faire également pour décrire les informations qu'il utilise. Ainsi il constate, par exemple, qu'il a besoin, dans un dossier de personnel, d'une rubrique "état-civil", mais il ne pensera à détailler cette rubrique que beaucoup plus tard lors d'une étude plus approfondie (méthode descendante), ou bien il constatera qu'elle a déjà été décrite ailleurs (lien avec la méthode ascendante). S'il a pu faire ses descriptions dans un langage strictement défini, il n'aura plus à faire la synthèse de son étude pour passer à la programmation : celle-ci pourra être faite automatiquement à partir des différents modules, organisation qui est reflétée par le langage.

Dans ce chapitre, nous nous attachons à préciser les notions de module et de classe et les relations pouvant exister entre elles. Ceci nous amènera à décrire le système Civa dans son ensemble. Mais une description analogue pourrait être employée pour tout autre système de programmation modulaire.

Le système Civa comportant plusieurs utilisateurs, chacun d'eux ayant des applications différentes à traiter, les modules, les classes et les objets qu'ils créent doivent pouvoir être protégés contre l'emploi d'utilisateurs autres que le propriétaire ou au contraire doivent pouvoir être partagés. De même les identificateurs employés par un utilisateur

(1) Il est curieux de constater que, dans les essais de programmation modulaire, on se soit attaché presque exclusivement à la description modulaire des actions et non pas des informations.

doivent pouvoir être employés par d'autres, ou par lui-même, en désignant d'autres objets (problèmes de portée) ou encore les objets crées doivent pouvoir avoir des durées de vie différentes. Le système proposé doit pouvoir résoudre tous ces problèmes. Nous décrirons celui-ci en définissant d'abord les notions de bases que sont les modules et les classes et les relations qui peuvent les lier (cf. 1.1 et 1.2), la suite de la description n'utilisant que ces notions de base en définissant des classes ou des modules particuliers (cf. 1.3 à 1.10).

1.1 MODULARITE DE LA DESCRIPTION DES ACTIONS : MODULES.

Les actions à entreprendre sont décrites par des instructions. Celles-ci sont groupées en unités que l'utilisateur désigne par un identificateur. Ces unités sont des modules. Ils correspondent le plus souvent aux tâches à accomplir qui ont pu être mises en évidence au cours de l'analyse fonctionnelle, puis de l'analyse détaillée. Un module peut aussi être une unité plus petite : il correspond alors à une séquence d'actions individualisée et qui peut être entreprise en différents points d'une chaîne de traitement.

Un module est une suite de déclarations et d'instructions

Un module X peut appeler un autre module Y, ce que nous noterons X␣A␣Y : pendant l'exécution du module X, la rencontre d'un appel de Y, représenté dans le texte source par une occurrence de l'identificateur Y, demande l'exécution du module Y. Un module définit une suite de règles de calcul à appliquer à chaque appel (instructions). Ces règles peuvent dépendre d'un certain nombre de paramètres. Dans ce cas, chaque appel du module doit comporter le nom des objets "paramètres effectifs" sur lesquels les règles de calcul du module appelé doivent être appliquées.

Le texte d'un module pouvant s'appliquer à des paramètres doit comporter une liste d'identificateurs (paramètres formels) qui permettront de désigner les paramètres effectifs dans le texte du module.

Un module X peut utiliser une classe Z, ce que nous noterons X␣U␣Z et qui s'écrit, dans le texte du module X : utilise Z ; tous les identificateurs décrits dans Z sont alors utilisables dans le module X, ce que nous préciserons en étudiant les conditions de validité des identificateurs.

Un module peut contenir des déclarations d'objets internes. Un objet interne "contient" une valeur ; cette valeur est fixe ou, au contraire, elle peut être changée par une opération d'affectation : nous dirons que l'objet est une constante si la valeur qu'il contient n'est pas modifiable, et qu'il est une variable si elle peut être changée ; un objet est associé lors de sa déclaration à un identificateur (nom externe) qui "désigne" cet objet ; évaluer une déclaration est donc créer un objet possédant les propriétés indiquées par la déclaration et définir la fonction "désigne" pour l'identificateur qu'elle comporte.

Une module peut en particulier contenir des déclarations de modules. Les modules ainsi déclarés s'appellent <u>procédures</u>. Une procédure ne peut pas contenir de déclaration de procédure. Elle ne peut pas utiliser de classe.

Les déclarations écrites dans un module sont locales à ce module : l'objet, créé lors d'une déclaration locale et associé à un identificateur I, n'est accessible par cet identificateur que depuis le texte de ce module, c'est-à-dire que la fonction "désigne" diffère pour chaque module.

Une déclaration d'un identificateur doit toujours précéder son utilisation (au sens statique de l'ordre d'écriture dans le module).

Les déclarations d'un module sont évaluées dans un ordre arbitraire.

Nous noterons D_I la relation entre un module et un identificateur qu'il déclare : $M \sqcup D_I \sqcup J$. De même, nous noterons D la relation "déclare un module", c'est-à-dire $M \sqcup D \sqcup P$ pour le module M "déclare le module" P (P est donc une procédure).

Un module doit toujours être déclaré : dans un module, c'est alors une procédure, ou dans une classe d'application, comme nous le verrons en 1.3.

Un module déclaré est dit "intérieur" au système.

La déclaration d'un module consiste en la suite :

<u>module</u> < identificateur > [(liste de paramètres formels)] (!) ;
< texte du module > <u>fin mod</u>.

La distinction entre module et procédure est donc assez minime et elle ne porte que sur leur domaine de validité. D'un point de vue fonctionnel, ce sont deux notions différentes, même si les critères de choix entre l'une ou l'autre solution sont assez difficiles à établir et procèdent un peu de l'intuition de l'utilisateur. Toute unité de description d'action mise en évidence pendant l'analyse fonctionnelle doit donner naissance à un module. Celui-ci, appelons-le A, pourra être décomposé à nouveau en unités plus élémentaires qui seront des unités nommées, si on pense qu'elles pourront être utilisées plusieurs fois ou simplement par commodité de description. Ces unités

(1) Nous utiliserons dans ce travail, la notation de Backus généralisée pour décrire la syntaxe. (Cf. Genuys - 60).

seront des procédures locales à A, si l'on est sûr qu'elles ne seront utiles que dans A, sinon elles seront des modules. On peut donc dire que, pendant les différentes phases de l'analyse d'un projet, les unités de description d'action rencontrées seront des modules. Pendant la programmation de ces modules, on pourra être amené à créer de nouvelles unités qui seront des procédures pour la plupart, mais certaines unités seront d'utilisation plus courante dans les divers modules, elles seront alors des modules.

De même, si une action est toujours attachée à une information, on pourra avoir intérêt à faire de l'unité la décrivant une procédure attachée à cette information et donc déclarée dans la même classe.

1.2 MODULARITE DE LA DESCRIPTION DES INFORMATIONS : CLASSES.

Nous avons vu que les informations utilisées peuvent être décrites dans des modules, mais les identificateurs ainsi déclarés sont locaux à ces modules. Cette possibilité est particulièrement destinée aux variables de travail. Elle ne suffit donc pas.

Dans une application, les structures logiques peuvent être répétitives (elles le sont certainement lorsqu'on crée un fichier de travail depuis un fichier d'entrée par exemple). Elles ne doivent donc pas être liées à un traitement.

Cependant, à la compilation d'un module on a besoin de connaître les structures qu'il utilise : elles ne peuvent donc pas être liées uniquement aux informations elles-mêmes (comme dans un fichier auto-descriptif). Les structures logiques seront donc décrites dans des unités indépendantes appelées <u>classes</u> (il s'agit de classes formées d'informations qui "vont ensemble").

Une classe permet donc de rendre communes à plusieurs modules, ceux qui l'utilisent, des déclarations d'identificateurs et donc des descriptions de propriétés d'objets créés. Mais l'indépendance et la modularité de la description des objets n'est pas seulement nécessaire pour la description elle-même : elle l'est également pour le partage des objets. La déclaration d'un objet écrite dans une classe n'est évaluée qu'une fois, indépendamment du nombre de modules qui "utilisent" la classe. L'objet unique ainsi créé permet donc aux modules de communiquer des valeurs pendant leur exécution. Sa durée de vie dépend de la durée d'exécution de l'ensemble des modules qui "utilisent" la

classe qui le déclare, comme nous le verrons en 1.4.

Une classe est une suite de déclarations. Cette suite a pour signification un ensemble de définitions : les déclarations d'une classe sont évaluées collatéralement, c'est-à-dire dans un ordre arbitraire (cf. Van Winjgaarden A. 69). Une classe peut être "utilisée" par un module ou par une autre classe, la relation "utilise" notée $\cup$ est donc définie de $M \cup C$ dans C.

Si un module ou une classe T utilise C, ce qui s'écrit dans T : utilise C ; tous les identificateurs de C peuvent être employés dans T, et ils désignent les mêmes objets.

Exemple :

Soit la classe C définie par :

Classe C ; I entier < 5 000 ; T type entier < 5 000 fin classe

et deux modules M1 et M2 "utilisant" C.

C permet de mettre en commun à M1 et M2 une description d'information T.

M1 ou M2 peuvent contenir une déclaration telle que :

M T ;

qui serait équivalente à M entier < 5 000 ;

Pour la déclaration de I, la classe C permet en plus aux modules M1 et M2 de se communiquer les valeurs de I, cet identificateur désignant le même objet, qu'il se trouve dans M1 ou dans M2.

Tout comme la relation "appelle" permet de décomposer une description d'actions en modules, la relation "utilise" permet de décomposer une description d'informations en petites unités.

Comme pour les modules, nous noterons $c \cup D_I \cup J$ la relation "c contient une déclaration de l'identificateur J" et $c \cup D \cup m$ la relation "c contient une déclaration du module ou de la classe m". Une classe doit toujours être déclarée avant de pouvoir être utilisée : elle est déclarée dans une classe d'application, ce qui la rend intérieure au système.

La déclaration d'une classe s'écrit :

classe < identificateur > ; < suite de déclarations > fin classe.

Exemple :

Dans le module EXEMP, on déclare un fichier FCØM, dont on sait simplement qu'il s'agit d'un fichier de commandes. Une commande sera précisée ultérieurement dans la classe A. Dans cette classe, on précise un peu le détail d'une commande : elle est constituée de quatre parties auxquelles on donne un nom ; DESIGNATION sera décrite dans la classe B tout comme NØ de CØMMANDE : ce sont des types déclarés (cf. 4.31). CLIENT est un identificateur du même type que l'objet PERSONNE que nous supposerons déclaré dans la classe d'application, ce qui le rend utilisable dans toutes les unités de l'application (cf. 4.32. Déclaration par analogie).

```
Module Exemp ;

    utilise A ;
    FCOM file COM ; ..... texte du module EXEMP        fin module ;

Classe A ;

    utilise B ;
    type COM struct (CLIENT type PERSONNE, NØ DE COMMANDE, QUANTITE
                     entier, DESIGNATION)

    fin classe ;

Classe B ; CØ elle est certainement définie à un autre moment ØC ;
           type NØ DE COMMANDE entier < 5 000 ;
           type DESIGNATION struct (REF entier < 1 000, IDENT
                file (max 30) car)

Fin classe ;
```

Dans l'exemple ci-dessus, la modularité de la description des informations est exploitée avec une méthode de conception "descendante" de la structure des informations. Ce n'est pas la méthode la plus fréquente, qui consiste au contraire en une conception "ascendante" :

- au début de l'analyse on ne peut souvent préciser que la nature des objets devant intervenir dans une structure (dans un enregistrement d'un fichier par exemple) et peut-être certains regroupements élémentaires sous forme de segments ;
- ce n'est que lorsque l'on connaît l'ensemble des traitements de l'application et, en particulier, lorsque l'on connaît tous les endroits où il est fait référence à ces segments et l'ordre de ces références, que l'on peut en déduire la structure proprement dite ; ce n'est donc qu'à ce moment qu'on peut la décrire.

Les segments sont décrits dans des classes, utilisées par la classe construite à la fin de l'analyse pour décrire la structure. Cette construction pourrait d'ailleurs être automatique. La notion de classe et la relation "utilise" sont donc des outils élémentaires permettant de décrire modulairement des informations et donc progressivement, mais elles ne présument en rien de l'ordre dans lequel les descriptions sont conçues.

La liaison entre une description d'informations et les modules l'utilisant se fera à la compilation de ces modules (étude de la validité des identificateurs et leur résolution).

L'utilisation d'un fichier consiste à associer à un nom de fichier (collection de valeurs, localisé en bibliothèque) une structure logique (décrite dans une ou plusieurs classes reliées par "utilise"). La liaison entre une description et un ensemble de valeurs se fera à l'exécution des modules.

1.3 NOTION D'APPLICATION. CLASSE D'APPLICATION ET CLASSE DU SYSTEME.

Dans une entreprise, les problèmes de protection d'informations concernent plus des services que des individus et affectent en général l'ensemble des informations d'un service. Il est donc intéressant que les déclarations de classes et de modules ne soient pas accessibles à tous les utilisateurs et que le pouvoir (1) d'utilisation d'une classe ou le pouvoir d'appel d'un module ne soient pas attribués à tous les modules, mais à certains d'entre eux regroupés par exemple au niveau d'un service de l'entreprise.

De plus, la notion de "programme" n'apparaît plus explicitement. Un programme sera construit par le système à partir des classes ou modules situés en bibliothèque ou nouvellement décrits par l'utilisateur. Il est donc préférable d'instaurer un niveau intermédiaire entre ces objets et le système d'exploitation.

Ce niveau est lié au domaine d'application des différents traitements. Dans ce domaine, on pourra prendre un certain nombre d'options implicites : constantes, types, valeurs des variables d'exploitation, description des fichiers, protections, etc... On appellera un tel niveau de regroupement une application.

C'est au niveau de l'application que l'on définira les bibliothèques de classes et de modules. L'ensemble des noms de modules et de classes pourra donc être différent d'une application à une autre. Ce découpage en applications peut reflèter le découpage en services d'une entreprise.

Enfin, dans un tel service, les habitudes de programmation, les rapports entre les différentes chaînes de traitement, etc..., font que l'ensemble des fichiers de ce service ont des paramètres assez semblables, disposition, protection, validité, délai de rétention, etc... C'est donc à une application que sont rattachés les fichiers (et les classes les décrivant) et c'est au niveau de l'application que l'on pourra définir les paramètres implicites pour tous les fichiers de l'application (options par défaut).

(1) Cette notion de pouvoir, souvent définie comme la possibilité de passer outre à une protection, (CROCUS , 73) deviendra en fait inutile dès que nous aurons précisé les conditions de validité d'un identificateur.

Le nom d'une application est donc à rapprocher de la notion classique de numéro de compte d'un utilisateur dans un système d'exploitation.

Notons l'existence d'une application particulière, celle du système lui-même.

On peut associer à chaque application une classe d'application qui définit les paramètres implicites évoqués ci-dessus. Une classe d'application sera implicitement utilisée par tous les modules de l'application (1). Il existe une seule classe d'application par application. Cependant, celle-ci peut "utiliser" d'autres classes (elle peut être décrite modulairement). Les identificateurs de ces classes, ont les mêmes propriétés que ceux de la classe d'application.

Cette notion de classe d'application "utilisée" implicitement par tous les modules de l'application correspond au fait que, dans une équipe de travail, il n'apparaît pas utile de rappeler, à chaque petite réalisation, le sens des mots employés ou la valeur des constantes utilisées : dans une application "comptabilité", il n'apparaît pas utile de rappeler, à chaque utilisation, les différents taux possibles de la T. V. A. par exemple, ce sont des informations implicites de l'application. Une classe d'application constitue donc, avec les bibliothèques de l'application, la "nomenclature" des informations manipulées dans cette application, donc, par exemple, dans un service de l'entreprise.

Il existe également des informations communes à tous les services

(1) La classe d'application est donc à rapprocher du prologue de programmathèque d'un programme Algol 68 (Van Winjgaarden A. 69), qui constitue un ensemble de déclarations évaluées implicitement au début de tout programme.

dont la durée de vie doit donc être plus grande que la durée d'une application particulière et qui doivent être "utilisables" dans toutes les applications.

Nous introduirons donc également une classe d'application pour l'application "système". Elle est utilisée par toutes les autres classes d'application (2). C'est dans cette classe que sont définis les identificateurs réservés tels que les indicateurs de débordement, les noms des paramètres d'exploitation tels que les protections, les délais de rétention, la disposition des fichiers, etc...

La classe du système contient également des déclarations de modules particuliers communs à toutes les applications : compilateur, éditeur, programmes de service, etc... Elle contient aussi des déclarations d'objets utilisables uniquement en lecture (ils sont définis comme des constantes), ne pouvant être modifiés que par l'intermédiaire de certaines procédures. Ces procédures sont déclarées dans une classe particulière appelée classe de modification (cf. 1.7).

On peut définir un nombre quelconque de modules ou de classes. Seuls, ceux qui sont déclarés dans une classe d'application sont dit <u>intérieurs au système</u>, les autres lui étant extérieurs. Les modules et classes intérieurs au système sont en nombre fini.

Donnons une définition plus précise de ces notions.

Il existe une classe particulière notée C_s et appelée <u>classe du système</u>, qui n'en utilise aucune autre $\cup (C_s) = \emptyset$.

Il existe une classe notée C_m et appelée classe de modification, telle que :

$C_m \cup C_s$ et $\nexists c \in C$ telle que $c \cup C_m$.

Nou verrons en 1.8 que seul un module de commande peut utiliser la classe de modification.

(2) La classe du système est donc à rapprocher du prologue standard d'Algol 68.

Une classe C_A est une <u>classe d'application</u>, si $C_A \neq C_M$ et $C_A \cup C_S$. La classe de modification est donc une classe d'application particulière. Une classe d'application peut déclarer des classes ou des modules.

Une classe ou un module T est dit intérieur au système si :

$\exists\ C_A$, classe d'application telle que $C_A \smile D \smile T$.

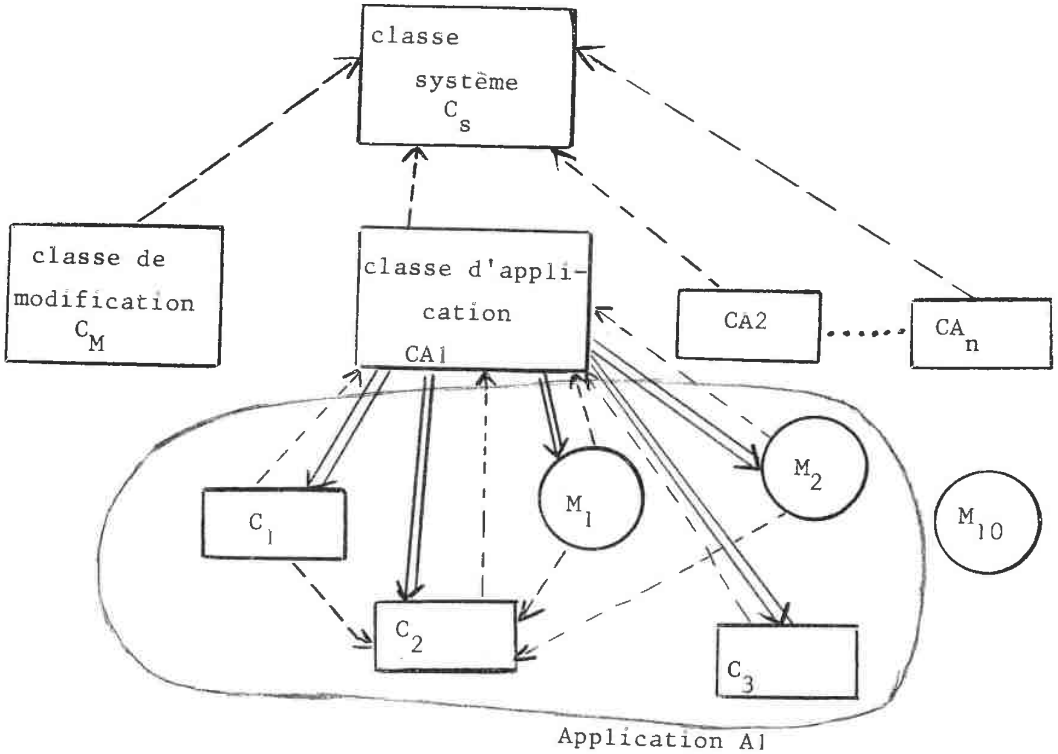
Une procédure P est intérieure au système si la classe ou le module T telle que $T \smile D \smile P$ est intérieur au système.

Une classe d'application C_A définit une <u>application</u> formée des classes et des modules de D (C_A).

Toute classe et tout module d'une application utilise implicitement la classe de l'application, et ce sont les seuls :

$\forall\ X \in M \cup C\ \ (\ C_A \smile D \smile X \Longleftrightarrow X \cup C_A)$.

Ce qui amène : $T \in C \cup M$ intérieur au système $T \smile \cup^2 \smile C_s$.(1)



(1) $\cup^2$ représente la relation $\cup \times \cup$. Dans la suite, nous utiliserons la notation $\hat{\Gamma}$ pour désigner la fermeture transitive, au sens large de la relation Γ : $\hat{\Gamma}(x) = \bigcup_{n=0}^{\infty} \Gamma^n(x)$.

Le schéma ci-dessus résume ce qui vient d'être dit. Un rectangle représente une classe, un cercle représente un module. $\dashrightarrow$ désigne la relation $\cup$ et $\Longrightarrow$ la relation D. Le module M_{10}, n'étant déclaré dans aucune classe d'application, est extérieur au système. C_1, C_2, C_3, M_1 et M_2 lui sont intérieurs.

1.4 CONDITIONS DE VALIDITE DES IDENTIFICATEURS.

Classiquement, une occurrence d'un identificateur est valide si on sait lui associer une signification, un objet existant qui a été déclaré, c'est-à-dire si l'on a défini, à un moment antérieur à son utilisation, quel sens et quel objet lui associer lors d'une utilisation. Dans un texte de programmation, une occurrence d'un identificateur sera donc valide si dans son contexte on trouve une déclaration le concernant. On appelle <u>domaine de validité</u> d'un identificateur l'ensemble des textes dans lesquels une occurrence de cet identificateur est valide.

Dans CIVA, une occurrence d'un identificateur "id" dans le texte d'un module est <u>valide</u> si "id" est déclaré dans ce module, ou dans une classe liée à ce module par une chaîne de "utilise" ou encore si ce module est déclaré dans un autre (cas d'une procédure) dans lequel "id" est valide.

Ce qu'on peut exprimer comme suit :

Soit M l'ensemble des modules, C celui des classes, I celui des identificateurs (il contient les identificateurs de modules et de classes).

Une occurrence de $i \in I$ est valide dans $m \in M \Longleftrightarrow m\ D_I\ i$ (1)

ou $\exists c \in C\ (c\ D_I i$ et $m \hat{\cup} c)$ (2)

ou $\exists m' \in M \cup C\ (m' D_m$ et $m' D_I i)$ (3)

ou $\exists m' \in M \cup C$ et $\exists c \in C (m' D_m$ et $m' \hat{\cup} c$ et $c\ D_I i)$ (4)

$$\Longleftrightarrow m\left[D_I \vee \hat{\cup} D_I \vee D^{-1} D_I \vee D^{-1} \hat{\cup} D_I\right] i$$

$$\Longleftrightarrow m\left[\hat{\cup} D_I \vee D^{-1} \hat{\cup} D_I\right] i$$

Dans le (1) il s'agit d'une déclaration de i locale au module m.

Dans le cas (2), le module m peut contenir une occurrence de i, car m est lié à une classe déclarant i par une chaîne d'"utilise". Dans le cas (3), m est une procédure. Elle est déclarée dans une classe

ou un module m', si i est déclaré dans m', une occurrence de i est valide dans m. (4) recouvre le cas où m est une procédure déclarée dans m' et i est déclaré dans une classe à laquelle m' est liée par une chaîne "d'utilise".

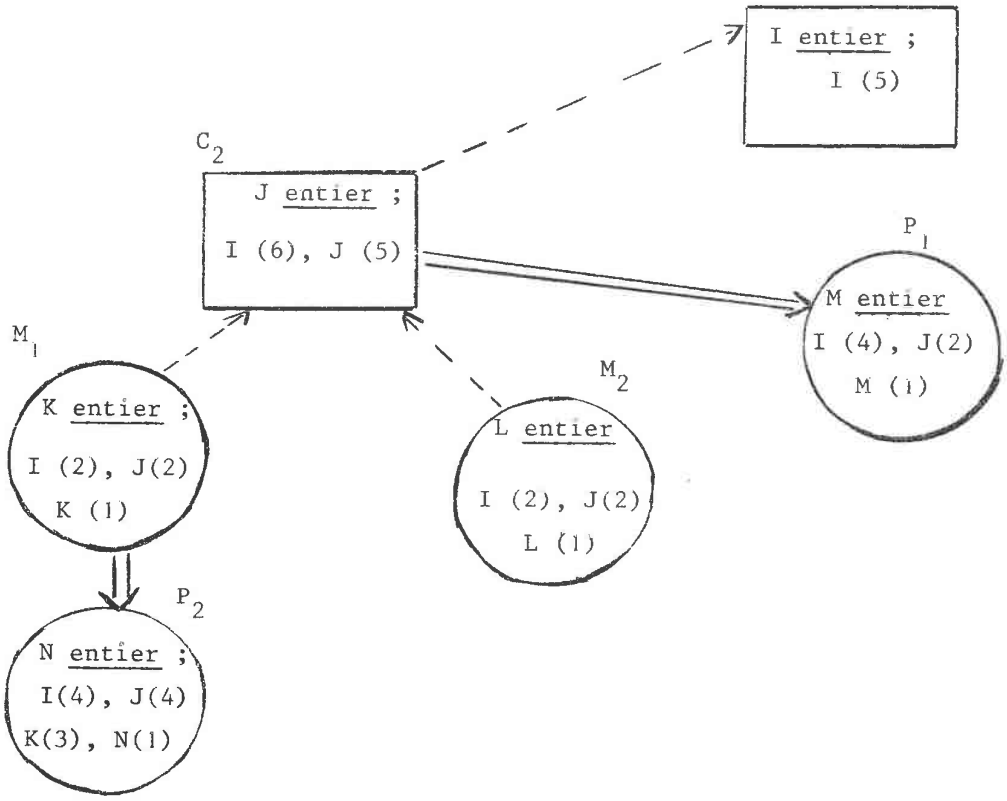
Une occurrence de $i \in I$ est valide dans $c \in C \Longleftrightarrow c\ D_I\ i$ (5)

ou $\exists c'\ (c\ \hat{U}\ c'$ et $c'\ D_I\ i)$ (6)

$\Longleftrightarrow c\left[\hat{U} D_I\right] i$

Exemple :

Dans le schéma ci-dessous, nous avons représenté un ensemble de classes et de modules avec les mêmes conventions qu'en 1.3. P_1 et P_2 sont des procédures. Chaque élément contient des déclarations. Nous avons fait figurer la liste des identificateurs valides dans chacun d'eux. Le chiffre entre parenthèse qui suit chaque identificateur indique selon quelle règle cet identificateur est valide.



Un appel de procédure doit correspondre à une occurrence valide de l'identificateur de la procédure appelée, d'où en notant P l'ensemble des procédures ($P_c M$).

$$\forall\, m \in M, \forall m' \in P\ (m\ A\ m' \Longrightarrow m\ \hat{\cup} D_I\ m').$$

De même pour un appel de module quelconque :

$$\forall\, m, m' \in M\ (m\ A\ m' \Longrightarrow m \left[\hat{\cup} D_I \vee D^{-1} \hat{\cup} D_I\right] m'.$$

La propriété vue en 1.3 disant :

$$\forall X \in M \cup C\ (C_A\ D\ X \Longleftrightarrow X \cup C_A)$$

permet de conclure que toute occurrence de l'identificateur d'un module (appel) déclaré dans une classe d'application est valide dans toute l'application et que toute occurrence de l'identificateur d'une classe (après <u>utilisé</u>) déclarée dans une classe d'application est valide dans toute l'application.

Il en est de même de tous les identificateurs déclarés dans une classe d'application.

Toutes les relations précédemment définies peuvent être considérées comme générales. En Civa, nous imposerons de plus une condition de non récursivité sur les modules et sur les classes :

- le graphe de la relation $\cup$ est sans circuit ;
- le graphe de la relation A est sans circuit ;
- le graphe de la relation D n'est pas nécessairement sans circuit.

Un module de nom m peut déclarer une procédure de nom m (déclaration locale.

1.5 DUREE DE VIE DES OBJETS ET MODIFICATIONS DE LA FONCTION "DESIGNE".

Exécuter la déclaration d'un identificateur, c'est créer un objet caractérisé par cette déclaration et l'associer à l'identificateur déclaré, on définit ainsi la fonction "désigne" pour cet identificateur. Cette association peut cesser à un instant ultérieur, l'objet correspondant est alors considéré comme détruit, ou elle peut être suspendue momentanément.

Nous appelons :

- <u>durée d'activation</u> d'un module, celle qui sépare le début de l'exécution de ce module, de sa fin ;

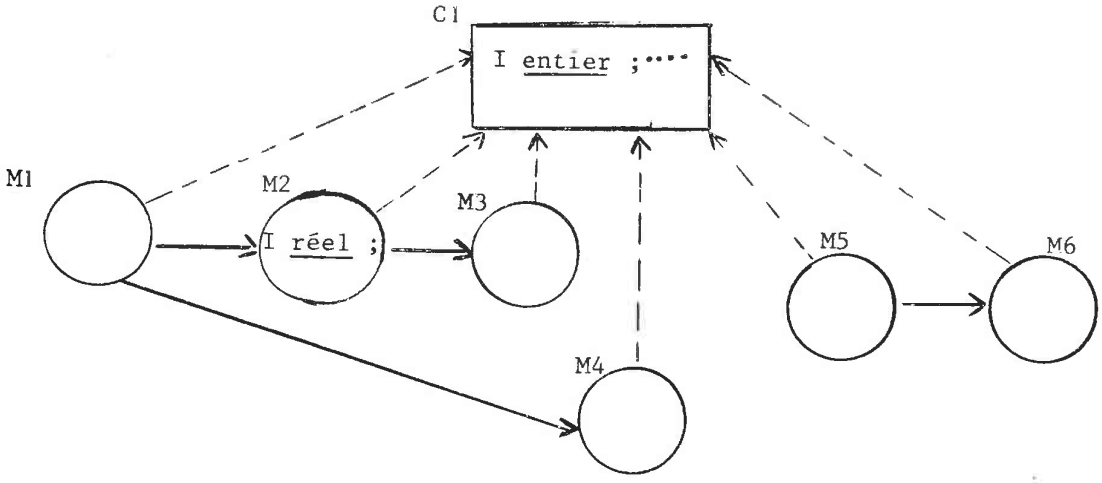
- durée de validité d'un identificateur i, la réunion des durées d'activation des modules appartenant au domaine de validité de i ;

- durée de vie d'un objet σ, celle qui sépare sa création de sa disparition : c'est-à-dire lorsqu'il n'existera plus d'identificateur pouvant "désigner" σ.

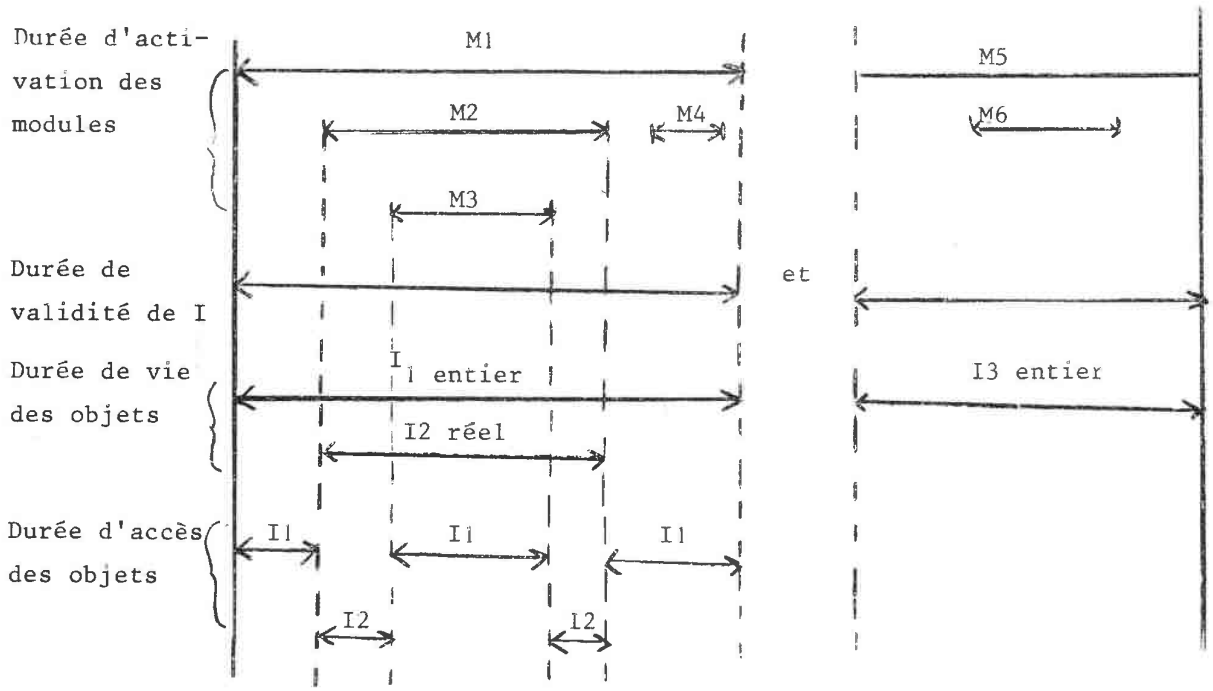
- durée d'accès à un objet σ, la réunion des temps pendant lesquels, il existe un identificateur (i) tel que désigne (i) = σ.

Nous n'avons pas défini dans le langage de directive d'équivalence entre identificateurs, et cet identificateur i est le même pendant toute la durée d'accès à l'objet σ.

Avant d'énoncer les règles qui définissent précisément ces durées, nous pouvons constater sur l'exemple ci-dessous qu'elles peuvent être différentes.



Aux conventions de 1.3 nous ajoutons : ⟶ représente la relation "appelle".



I1 et I3 sont deux objets différents de type entier. I3 est encore un autre objet de type réel. Tous trois sont désignés par I à des instants différents.

Soit i un identificateur et E l'ensemble des classes et modules contenant une déclaration de i. Nous notons d (m) la durée d'activation de m.

Le domaine de validité D de i est donc $\hat{U}^{-1}$ (E) et sa durée de validité est définie par $\bigcup_{m \in D} d(m)$.

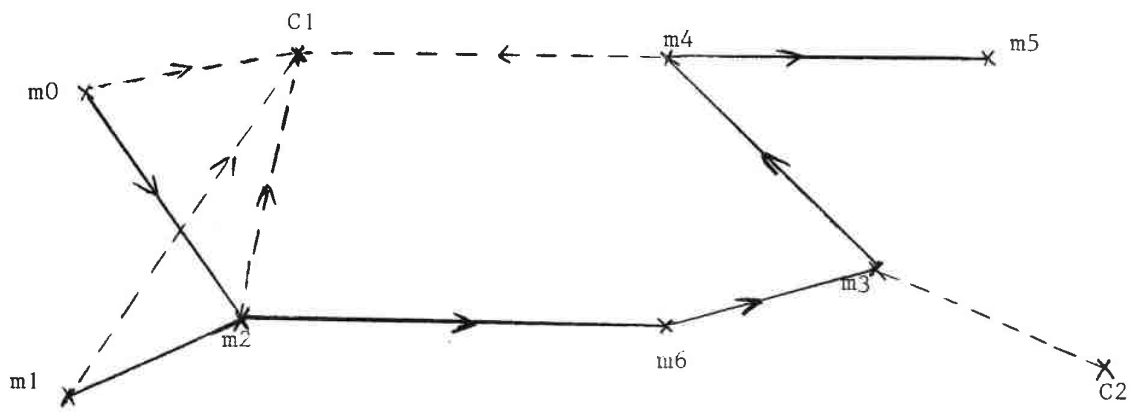
A chaque élément y de E, associons Y (y) $= \hat{U}^{-1}$ (y) $\cap$ M et considérons le sous-graphe (Y (y), A). Chaque composante connexe de ce sous-graphe détermine un objet associé à i.

Lors de l'exécution d'une chaîne d'appels de modules, chaque fois qu'on entre dans une nouvelle composante connexe, un objet est crée, dont la durée de vie est la durée d'activation du module constituant ce point d'entrée.

La durée d'accès de cet objet est égale à sa durée d'activation amputée des durées de vie d'objets $\mathcal{O}_k$ correspondant à des composantes connexes d'autres sous-graphes (Y ($\mathcal{R}$), A) rencontrées pendant cette activation.

Exemple :

Soit le digraphe $G = (M \cup C, A, \Upsilon)$ suivant. Supposons que $E = \{C1, C2, m5\}$ et que m ne soit pas appelé effectivement.



$\Upsilon^{1}(E) = \{c1, m_0, m_1, m_2, m_3, m_4, m_5, C_2\}$.

$Y(C1) = \{m_0, m_1, m_2, m_4\}$

$Y(C2) = \{m_3\}$

$Y(m_5) = \{m_5\}$

Y (C1) possède 2 composantes connexes et détermine deux objets O1 et O2, Y (C2) correspond à O3 et Y (m_5) à O4.

La durée de vie et la durée d'accès sont égales à la durée d'activation de m_3 pour O3, à celle de m_5 pour O4, à celle de m_1 pour O1.

O2 a pour durée de vie la durée d'activation de m_4. Sa durée d'accès est la même amputée de la durée d'activation de m_5 (durée de vie de O4).

Ces notions nécessaires pour la définition de Civa seront particulièrement utiles au chapitre 25, lorsqu'on voudra récupérer la place des objets détruits et définir des recouvrements.

1.6 EXECUTION D'UN MODULE m APPELE PAR m'

L'exécution d'un appel du module m dans un module m' consiste en trois opérations :

a) exécution des déclarations :
il s'agit des déclarations locales au module m et des déclarations contenues dans $\hat{U}(m)$.
L'exécution des déclarations locales consiste à créer les objets qu'elles décrivent et à définir la fonction "désigne" entre chaque identificateur et l'objet créé. Pour les déclarations de $\hat{U}(m)$, seules sont prises en compte les déclarations des classes C, $C \in \hat{U}(m)$ et $C \notin \hat{U}(m')$. En effet, pour toute classe ω de $\hat{U}(m) \cap \hat{U}(m')$, un identificateur i tel que $\omega D_I i$ désigne le même objet dans m et dans m' (m et m' appartiennent au même sous-graphe et, comme m' A m, ils appartiennent à la même composante connexe). L'exécution des déclarations d'une telle classe ω ne crée aucun objet et ne modifie rien.
Pour une classe C, $C \in \hat{U}(m)$ et $C \notin \hat{U}(m')$, il convient de créer les objets détruits et de mettre à jour la fonction "désigne" pour les identificateurs correspondants.

b) exécution successive des instructions de m, en particulier des appels de modules. Le rôle et la signification des instructions Civa seront précisés dans les chapitres 5 à 8.

c) mise à jour de la fonction "désigne" en supprimant les liaisons créées en a) pour les identificateurs déclarés dans m (on retrouve alors la définition antérieure de "désigne" si cet identificateur était valide dans m'), et pour les identificateurs des classes de $\hat{U}(m)$ ne figurant pas dans $\hat{U}(m')$.

1.7 MODIFICATION DE L'ENSEMBLE DES CLASSES ET DES MODULES INTERIEURS AU SYSTEME.

Il existe une classe d'application particulière, appelée classe de modification C_M, qui contient les déclarations des modules de modification. Ceux-ci permettent d'ajouter de nouvelles classes ou de nouveaux modules, dans une application d'en supprimer, ou de remplacer

le module ou la classe associée à un identificateur par un autre objet, donc de changer le texte des modules. Enfin, ces modules effectuent la mise à jour du graphe des appels de modules et celui de la relation "utilise".
La classe C_M contient également les procédures de modification des "paramètres d'exploitation", c'est-à-dire des variables réservées de C_s.

1.8 MODULE DE COMMANDE.

Il existe un module initial unique M_0 (c'est le "système Civa") qui utilise toutes les classes d'application et qui est constamment en cours d'exécution. Les objets déclarés dans les classes d'application ont donc une durée de vie égale à la durée d'activation de ce module : ce sont donc des objets permanents. C'est à ce module M_0 que sont soumis les travaux à exécuter.

Cependant l'exécution de travaux n'est pas permanente et l'exécution de M_0 peut être momentanément suspendue.

Nous appellerons module de commande un module différent de M_0 et qui utilise C_M. C'est une suite de déclarations et d'instructions soumise à M_0 : pour chaque élément du module de commande M_0 créé un module qu'il rend intérieur au système, exécute et détruit aussitôt. Un module de commande ne peut donc être appelé par aucun autre module que par M_0. Il n'a d'ailleurs pas d'identificateur externe. Le module M_0 contrôle les conditions que doit vérifier un module de commande.

Le texte du module de commande comporte l'indication de l'application dans la classe de laquelle M_0 le module de commande. Il "utilise" donc cette classe d'application.

Il s'agira toujours de modification à apporter à la classe système ou à la classe de l'application du module de commande, ce qui assure les protections souhaitables. Ces modifications consistent essentiellement en déclarations à ajouter dans la classe d'application (définition de nouvelles classes, de nouveaux modules ou de nouveaux objets) ou à en supprimer.

Un module de commande utilise la classe du système C_s. Il a donc

la possibilité, utilisant C_S et C_M, de modifier les variables réservées de C_S que nous avons appelées "paramètres d'exploitation", par l'intermédiaire des procédures déclarées dans C_M. C'est le seul type de module à pouvoir le faire.

Un module de commande peut appeler des autres modules. Une exécution Civa correspond à l'exécution d'un module de commande. Un module M appelé par un module de commande correspond à ce qu'on appelle habituellement une unité de traitement, les différents appels de modules contenus dans un module de commande constituant une chaîne de traitement. Une unité de traitement de premier module M, nous dirons aussi de module directeur M, est constituée des ensembles $\hat{A}$ (M) et $\widehat{U}$(M).

Un module de commande permet donc de demander l'exécution d'unités de traitement ; comme il permet également de fixer les valeurs des "paramètres d'exploitation" et de gérer les "bibliothèques" des classes et des modules d'une application, il correspond donc aux paquets de cartes de commandes auxquels nous sommes habitués pour décrire la phase d'exploitation des travaux.

Habituellement les cartes de commande représentent les instructions (statiques le plus souvent) d'un langage de commande qui pourrait contenir, comme un langage de programmation, (il les contient quelquefois), des instructions d'affectation à des variables de commande, des instructions conditionnelles, des instructions de saut et des sous-programmes de commande (procédure cataloguée). Nous devons donc pouvoir retrouver dans un module de commande les mêmes outils que ceux utilisés pour l'écriture des modules. Il est extrêmement important de les retrouver, et sous la même forme, pour atteindre l'un de nos objectifs qui est de faciliter le passage de la programmation à l'exploitation et leur description commune dès la phase de l'analyse.

Un module de commande contient donc des instructions Civa, avec quelques restrictions (pas d'instruction itérative par exemple). Il utilise des objets locaux au module de commande ou communs à celui-ci et aux modules de l'application : ils sont alors décrits dans une classe que nous appellerons classe de commande. C'est une classe sans propriété particulière.

Un module de commande peut donc contenir :

- en tête le nom de l'application à laquelle il appartient ;
- la création éventuelle d'une classe de commande. Celle-ci peut déjà exister dans l'application, on rencontrera alors l'indication de la classe de commande utilisée ;

Enfin, dans un ordre quelconque :

- des déclarations : il s'agit d'identificateurs locaux à ce module de commande, leur durée de vie sera égale à la sienne ;
- une demande éventuelle de modifications de la classe de l'application ;
- des créations de nouvelles classes et de nouveaux modules ; ils sont rangés dans une bibliothèque de l'application ;
- des suppressions de classe ou de module ;
- des demandes de modifications de classes ou de modules existants ;
- des appels de modules (demande d'exécution d'unités de traitement) ;
- d'autres instructions.

Il s'agit là essentiellement d'instructions d'affectation et d'appels des procédures de modification des "paramètres d'exploitation" ; elles permettent donc de fixer les paramètres d'exploitation pour la durée d'activation de ce module de commande. On peut y rencontrer également des tables de décision.

1.9 INTERPRETATION D'UN MODULE DE COMMANDE.

Dans la pratique, un module de commande est interprété, c'est-à-dire que chaque instruction du module de commande est exécutée dès sa rencontre et sa traduction. Le mode interprétatif pourrait être utilisé tout au long de l'exécution d'une chaîne de traitement (cas des systèmes de programmation modulaire en temps réel par exemple). En Civa, les modules appelés par un module de commande (unité de traitement) sont traduits et édités avant de pouvoir être exécutés.

La création d'un module ou d'une classe entraîne la mise en bibliothèque de cette unité sous son format externe (texte source). Un appel de module déclenchera toutes les opérations nécessaires avant son exécution : si le module demandé a déjà été édité et si aucune modification du texte source de l'un des composants qu'il "appelle" ou

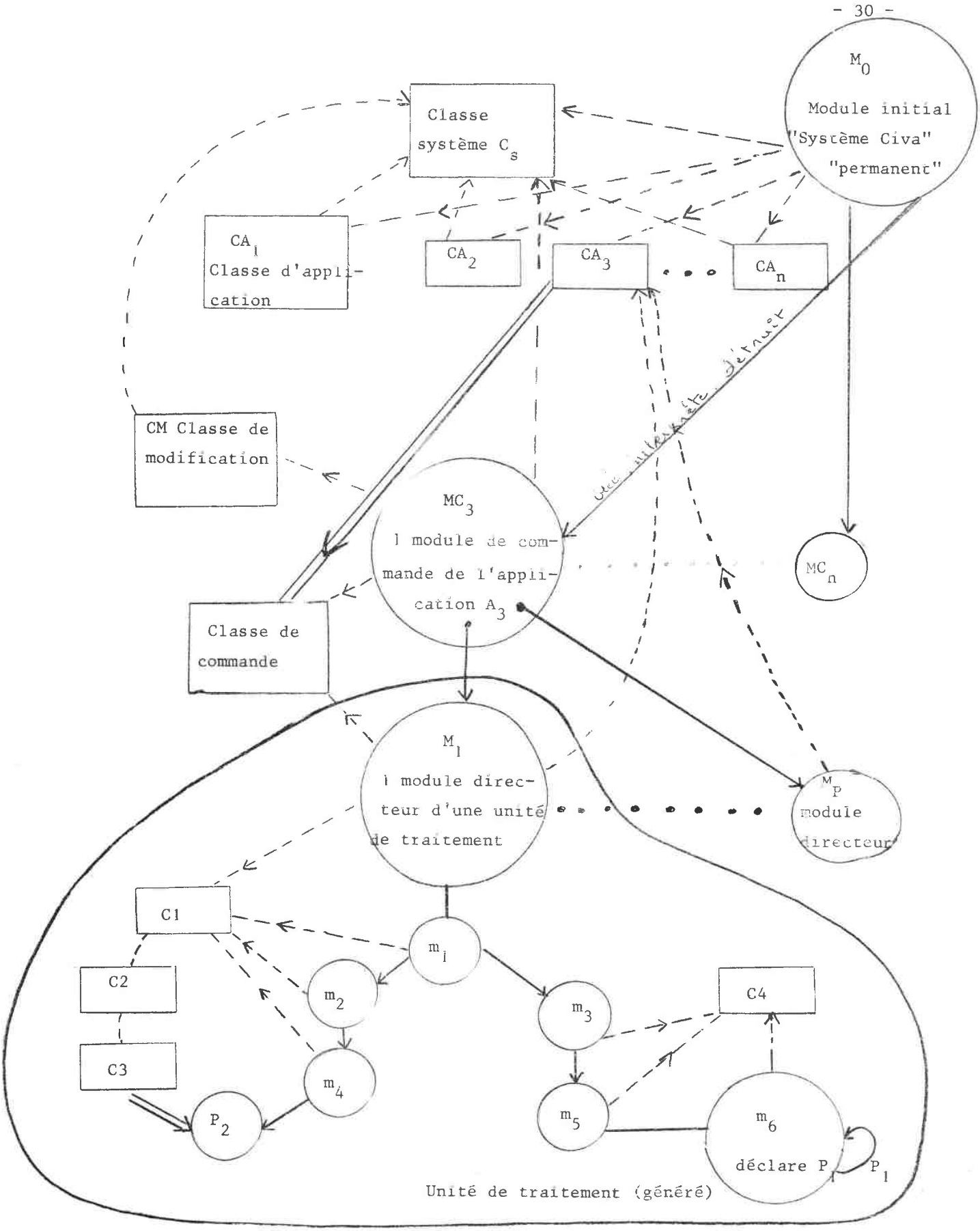
"utilisé" n'est intervenue depuis son édition, il sera exécuté immédiatement ; s'il n'est pas édité, tous les modules et toutes les classes pouvant être utilisés par lui seront examinés et il sera procédé éventuellement à leur compilation. L'édition de liens sera alors demandée automatiquement (après appel du pré-éditeur, cf. "réalisation") : l'unité de traitement pourra être exécutée.

1.10 SCHEMA GENERAL.

---------- "utilise" U

——————→ "appelle" A

═══════> "déclare" D, en général cette relation n'a pas été représentée : les constituants de l'unité de traitement sont déclarés dans A_3, les classes d'application dans la classe système.


M_0
Module initial
"Système Civa"
"permanent"
Classe
système C_s
CA_1
Classe d'appli-
cation
CA_2
CA_3
CA_n
CM Classe de
modification
MC_3
1 module de com-
mande de l'appli-
cation A_3
MC_n
Classe de
commande
M_1
1 module direc-
teur d'une unité
de traitement
M_P
module
directeur
C1
C2
C3
C4
m_1
m_2
m_3
m_4
m_5
m_6
déclare P_1
P_1
P_2
Unité de traitement (généré)

1.11 EMPLOI, DANS UNE APPLICATION, D'OBJETS EXTERIEURS A L'APPLICATION.

Les relations "appelle"et "utilise" ne sont définies qu'entre objets de la même application. Cette définition, restrictive, assure une protection efficace des informations de chaque application : la notion d'application est alors à rapprocher de celle de numéro de compte. Elle ne permet pas un partage d'information ou une mise en commun de modules entre deux applications. Ces partages seront possibles à condition que l'autorisation en ait été donnée explicitement par le propriétaire d'une application (voir le chapitre 21 - Interprétation du module de commande). Ainsi, à l'adjonction dans l'application A d'un module ou d'une classe, il pourra être précisé, que son emploi ("appelle" ou "utilise") est autorisé par les objets d'une autre application B. Dans un texte source de B, on rencontrerait

utilise C de A si C est une classe ; si l'objet ainsi partagé est un module, il conviendra d'indiquer dans un texte de l'application B qu'on appelle un module de A : ce texte, module ou classe devra comporter en son début une directive "appelle" qui s'écrit :

appelle M de A ; ou M est un module.

Dans la suite du texte, M pourra être désigné par l'identificateur M, si dans l'application B cet identificateur n'est pas déjà utilisé pour désigner un module et par M de A s'il y a un risque de confusion.

On pourra considérer que la rencontre de utilise C de A ou de appelle M de A est interprétée comme une demande d'adjonction dans la classe d'application A de l'unité M de A ou C de A. Cette adjonction est "temporaire" en ce sens qu'elle est limitée à l'exécution du module de commande. De plus, cette adjonction n'est possible que si l'unité M ou A est déjà compilée dans A.

CHAPITRE 2

LE MODE DECLARATIF

2.1 INTRODUCTION.

2.2 MISE EN OEUVRE DES "ENONCES".

2.3 LES INSTRUCTIONS DESORDONNEES.

2.4 LES METAMODULES.

2.41 La notion de métamodule.

2.42 Ecriture des métamodules. Métalangage.

2.43 Exemples de métamodules.

2.5 MODE DECLARATIF ET CHOIX DES INSTRUCTIONS.

2.6 MODE DECLARATIF ET DESCRIPTION DES INFORMATIONS.

2.61 Déclarations de type.

2.62 Métamodules et déclarations.

2.7 PREMIERE CONCLUSION.

CHAPITRE 2

LE MODE DECLARATIF

2.1 INTRODUCTION.

Le mode déclaratif permet de réaliser l'un de nos buts, qui est "d'éviter le plus longtemps possible" à l'utilisateur d'avoir à contrôler le détail des opérations à entreprendre ou des informations à traiter. Il existe en effet deux grandes façons de demander un travail à un ordinateur (Codasyl - 71 a. et b.).

La première, accessible uniquement au programmeur, consiste à donner des séquences d'ordres à exécuter, en utilisant un langage support. Ces ordres peuvent être les instructions élémentaires du langage ou des regroupements de ces instructions (procédures et modules). C'est la possibilité offerte par les assembleurs et les langages "orientés-procédure". Nous retrouvons là l'aspect classique de la programmation : le mode impératif, dans lequel tous les détails du traitement sont imposés par le texte source.

La seconde consiste à décrire le traitement sans indiquer comment il doit être effectué. Il faut pour cela disposer d'outils plus élaborés et d'utilisation facile, même pour un non programmeur : l'utilisateur n'a pas à contrôler, ni même à connaître, la séquence d'instructions à exécuter. Il doit simplement préciser les informations nécessaires à leur utilisation : c'est le mode déclaratif. Nous qualifions "d'énoncé" un objet en mode déclaratif, pour lequel l'utilisateur se contente d'énoncer ce qu'il y a à faire sans préciser (à cet endroit) comment.

Des générateurs de tri, les services, les "packages" en général sont des "énoncés".

Il n'y a pas de frontière entre les deux modes. En effet, mettre une procédure à la disposition des autres utilisateurs par l'intermédiaire d'une bibliothèque en fait déjà un "énoncé" et, dans les applications scientifiques, la disposition d'une bonne bibliothèque

de procédures est plus importante que la connaissance approfondie d'un langage de programmation. De même on peut prendre les "énoncés" comme base d'un langage impératif de plus haut niveau, comme par exemple un langage d'utilisation d'une bibliothèque.

Il est clair que dans un langage de commande, pour l'exploitation , on fait appel uniquement à des "énoncés" : traducteurs, éditeurs, sous-programmes de gestion des fichiers, etc...

En programmation, on utilise essentiellement le mode impératif, alors qu'au cours de l'analyse, au moins pendant les premières phases, la description des actions et des informations est souvent encore très imprécise et ne nécessite que des "énoncés". D'ailleurs, même quand il passe à l'analyse détaillée, l'analyste désire bien rester le plus longtemps possible dans le mode déclaratif. Ce désir impose une analyse modulaire et il ne sera complètement satisfait que s'il dispose d'une importante bibliothèque de modules décrivant des actions (programmes ou services), mais aussi d'informations qu'il suffira d'utiliser par leur nom sans avoir à les décrire à nouveau.

2.2 MISE EN OEUVRE DES ENONCES.

Une première solution consisterait à utiliser un langage support uniquement "orienté-procédure", pour la rédaction des modules, un système de gestion de bibliothèque et un ensemble important d'énoncés (services) rédigés par le fournisseur du software. Cet ensemble serait nécessairement limité et figé pour l'utilisateur.

Il est donc préférable que ce langage soit étendu pour qu'il puisse également être utilisé pour permettre la création d'énoncés par certains utilisateurs : on pourra par exemple rédiger des générateurs d'interrogation, de mise à jour, de création, de restructuration, d'établissement de bilans, etc... sur ses fichiers.

Les points que l'on peut vouloir laisser dans l'ombre sont essentiellement l'ordre d'exécution des instructions et l'adaptation d'une action à des paramètres. D'où la proposition de deux types d'objet autonomes pouvant être rédigés par l'utilisateur : les instructions désordon-

nées et les métamodules (1). De là aussi provient la proposition d'instructions dont la forme particulièrement souple en fait des "énoncés" : tables de décision ou instructions de traitement global de files.

2.3 LES INSTRUCTIONS DESORDONNEES.

Une instruction désordonnée est une unité de description d'action (procédure ou module) pour laquelle l'ordre d'exécution des instructions n'est pas explicitement imposé : l'utilisateur se contente de "déclarer" (c'est le mode déclaratif de 2.1) l'ensemble des actions qu'il veut voir exécuter sans se préoccuper de l'ordre d'exécution de ces actions. Ainsi, l'énoncé suivant, qui est d'une formulation courante, peut être une formulation de programmation :

Calculer B = A x 5 + solde client + taux * somme
sachant que A = D + 52, taux = 0.75, somme = 1 200,
D = 7, solde client = 2 345.

Plus génralement, l'écriture mathématique d'un système d'équations ne fait pas intervenir l'ordre d'utilisation de ces équations alors que, pour le résoudre, un programme doit le faire (Gallaire - 72), les langages de programmation étant habituellement des langages impératifs. Des études ont déjà été faites pour définir des langages acceptant une formulation "déclarative" des calculs à entreprendre : programmation dirigée par les données ou langages à assignation unique (Chamberlin, Gallaire 72, Tesler L.).

Une instruction désordonnée est composée essentiellement d'une suite d'instructions itératives et d'instructions d'affectation vérifiant une contrainte : un identificateur ne figure qu'une seule fois à gauche d'une instruction d'affectation. L'ordre d'évaluation de ces

(1) Cette distinction est faite pour faciliter l'exposé, mais un métamodule pourra contenir une instruction désordonnée, tout comme un métamodule n'est qu'un module particulier. Il peut avoir simplement une présentation différente.

instructions sera déterminé par le compilateur comme nous le verrons au chapitre 17, alors que dans les articles cités l'ordonnancement est fait à l'exécution, le texte source étant interprété. Il utilise pour cela le graphe (sans circuit) dont les sommets sont les identificateurs utilisés dans les instructions d'affectation et dont la relation Γ est :

$x \Gamma y \Longleftrightarrow$ y figure à droite d'une affectation de premier membre x.

$\hat{\Gamma}$ ne définit qu'un préordre sur l'ensemble des sommets du graphe. Les points équivalents dans ce préordre correspondent à des instructions qui seront exécutées dans un ordre arbitraire.

Les instructions autres que les affectations et les itérations seront exécutées dans un ordre arbitraire.

2.4 LES METAMODULES.

2.41 La notion de métamodule.

Les métamodules sont des "énoncés" qui dépendent de paramètres et en particulier qui peuvent dépendre de fichiers. En effet, il est important de pouvoir décrire une action sur un enregistrement de fichier sans connaître précisément ce fichier. Mais, pour des raisons d'efficacité, la liaison entre le fichier virtuel (celui connu des modules) et le fichier réel (il a une existence physique : un support, des valeurs,.....) ne doit pas se faire, pendant l'exécution, pour chaque enregistrement du fichier. Il est préférable de faire cette liaison à la compilation d'un appel de module utilisant un fichier. Les actions ainsi décrites le sont donc par un métamodule.

C'est également pour pouvoir effectuer le plus complètement possible la liaison entre un fichier virtuel et un fichier réel à la compilation d'un appel de module que tous les fichiers utilisés dans une application sont en format interne, c'est-à-dire que chacune des valeurs qu'ils contiennent sont présentées sous leur format de traitement (converties) et non sous leur format de présentation externe. Des métamodules d'acquisition et d'édition permettront de décrire la correspondance entre un fichier externe et un fichier interne.

Un métamodule est un module particulier formé, comme un module, d'une suite de déclarations, d'instructions, de méta-instructions

destiné à construire un texte du langage source. Les déclarations et instructions d'un métamodule sont générées pour chaque référence à ce métamodule rencontrée dans un module ou une classe, en fonction des conditions présentes pour cette référence (description des paramètres effectifs par exemple).

2.42 Ecriture des métamodules - Métalangage.

Pour réaliser l'adaptation d'un métamodule aux conditions présentes lors d'une référence, il faut entreprendre des calculs (essentiellement des tests) lors de cette phase de production de texte, antérieure à compilation. Ces calculs sont décrits dans le texte source à l'aide d'un métalangage.

Cette notion n'est pas nouvelle (cf. chap. 9). La nature d'un métalangage apparaît clairement à propos d'un méta-assembleur (cf. 9.1). Il permet de décrire des calculs effectués avant ou pendant la traduction d'un texte source et non après (exécution proprement dite). C'est un langage de construction de texte.

Comme un langage de programmation, il comprend des constantes, des variables , des instructions d'affectation, des instructions conditionnelles ou itératives portant sur des métavariables et des "sous-programmes" : ce sont les métamodules.

Un métamodule peut être utilisé pour représenter un texte quelconque : tout ou partie d'un module ou d'une classe.

2.43 Exemples de métamodules.

a) Ainsi, pour décrire un service (1) d'acquisition de fichiers,

(1) Nous utiliserons quelquefois ce terme, car les métamodules sont bien adaptés à l'écriture des programmes souvent appelés "programmes de service".

on pourrait écrire un métamodule d'acquisition (2) simple, (pas de création de fichier de rebut) ; il permet de transformer un fichier en format externe, représenté par une description que nous appellerons format, en un fichier "interne", appelé INFO dans cet exemple.

```
MODULE    ACQUISITION : INFØ fichier, MODELE Format ;
          Pour chaque X de INFØ faire
          entrer (X, MODELE) finpour finmod ;
```

L'instruction entrer, qui sera étudiée au chapitre 8, permet de faire la conversion d'un enregistrement en format externe en un élément en format interne (il est entré dans le système).

Ce métamodule pourrait être utilisé dans un module pour demander l'acquisition du fichier A selon le format G (décrit dans la classe E) ; le fichier A est supposé décrit dans la classe D :

```
MODULE    ACQ ; utilise D, E ;
          ACQUISITION : FICHIER : A, FORMAT : G   fin module
```

A la compilation de ce module, il sera généré un texte du service, adapté aux descriptions (interne et externe) du fichier A.

b) De même pour rédiger un service d'interrogation indirecte ("off-line") d'un fichier : interrogation est un service permettant d'extraire du fichier "source" deux fichiers "produit 1" et "produit 2" et un fichier dit "rebut" selon des conditions vérifiées par les éléments du fichier source. Interrogation compte également le nombre d'éléments de chaque fichier construit.

```
MODULE interrogation : source, produit 1, produit 2, rebut fichier,
          C1, C2, C3 condition, E1, E2, E3 entier ;
CO fin des paramètres CO ;
```

(2) Les problèmes rencontrés à l'acquisition : reconnaissance, conversions, contrôles de validité des informations, constitution des fichiers acquis ou des fichiers de rebut sont étudiés au chapitre 11 et dans (Chabrier J. J. 73).

$E_1 = E_2 = E_3 = 0$; premier Y de produit 1 ; premier Z de produit 2 ;
premier T de rebut ;

Pour chaque X de SOURCE faire

Conditions	C_1	V	F	F ;
	C_2	F	V	V ;
	C_3	V	V	F ;
Actions	A_1	1 ;		
	A_2	-	1 ;	
	A_3	-	-	1 fin table fp ;

procédure A_1 ; $E_1 = E_1 + 1$; nouveau Y = X fin ;

procédure A_2 ; $E_2 = E_2 + 1$; avancer Z ; Z = X fin ;

procédure A_3 ; $E_3 = E_3 + 1$; nouveau T = X fin fin mod ;

Ce métamodule pourrait être utilisé pour créer un module d'interrogation d'un fichier FICH déclaré dans la classe A.

module ASKFICH ;
utilise A ;
SELECT 1 comme FICH ; co déclaration d'un nouveau fichier SELECT 1 ayant la même structure que le fichier FICH co ;
SELECT 2, SELECT 3 comme FICH ;
INTERROGATION : SOURCE : FICH, PRODUIT 1 : SELECT 1, PRODUIT 2 : SELECT 2, REBUT : SELECT 3,

C_1 : DATE de NAISSANCE $<$ 15.03.32 et DATE de ENTREE $>$1950,

C_2 : QUALIFICATION = 3,

C_3 : ENFANT $\geqslant$2 ; co E_1, E_2, E_3 ne sont pas précisés ce qui signifie que les paramètres effectifs correspondants s'appellent E_1, E_2, E_3 également co.

c) Les exemples a, b, n'utilisent pas le métalangage, la seule adaptation à faire lors d'une référence à ces métamodules consiste en une substitution des paramètres effectifs aux paramètres formels. L'exemple suivant décrit des opérations plus complexes et utilise le métalangage défini au chapitre 9. Tout identificateur et tout symbole constitué d'une suite de caractères du métalangage commence par le caractère $.

Extraction de la structure B dans la structure A.

On veut décrire le transfert des valeursde certains champs d'une structure A vers une structure B telle que tout nom de feuille dans B est un nom de feuille dans A ; le transfert n'est effectué que pour les champs qui sont nommés dans les deux structures ; un tel transfert est qualifié d'extraction de la structure B dans la structure A.

```
$ mod  $ extraction ( $ A, $B) ;
  $ Pour chaque  $ X  $ de  $ mot feuille ($ B)  $ faire
     $ Pour chaque   $ Y  $ de  $ mot feuille ($ A)  $ faire
        $ si  $ X = = $ Y  $ alors  $X = $ Y  $ fsi
       $ fpc
      $ fpc
     $ fin mod ;
```

$ mot feuille ($ B) est une métafonction prédéfinie qui a pour valeur la liste de valeurs constituée des noms des feuilles de $ B (chaînes de caractères).

On pourrait rédiger de nombreux services de cette manière, tels que des services d'acquisition de fichier assurant, grâce aux conditions précisées dans une description externe, un contrôle de la validité des données à leur entrée en ordinateur et créant les fichiers de rebut correspondant. De même, c'est par des métamodules que seront écrits les services d'édition.

Les métamodules peuvent être écrits par l'utilisateur.

Il convient néanmoins que les auteurs du projet suggèrent un certain nombre de services. On trouvera donc dans la troisième partie de ce travail quelques exemples de services proposés et leur utilisation possible. En particulier, nous nous attacherons à la description de services permettant d'appliquer commodément des opérateurs élémentaires de gestion.

Prenons par exemple les opérateurs définis dans (Reix R. 71) ; ils sont répartis en quatre classes selon le type des modifications qu'ils introduisent.

Les opérateurs de classe 0, ce sont la duplication et la conversion, sont réalisés à partir des instructions d'affectation et d'entrée. Les opérateurs de classe 1, ceux "qui modifient l'ordre des rubriques dans la file sans modifier la structure de l'article" sont rédigés essentiellement à l'aide des instructions "pour chaque" :"pour chaque" et option d'ordre pour le tri, "pour chaque" et table de décision pour l'interclassement, et pour la partition. Les opérateurs de classe 2, ce sont ceux qui modifient la structure des articles, sont réalisés à l'aide des instructions "vers" ou d'affectation. (Union, éclatement). Les opérateurs de classe 3, entraînant une modification de la file des valeurs ou la création de nouvelles valeurs (mise à jour, actualisation), sont écrits à l'aide des instructions de traitement des files et des ensembles et des instructions d'affectation.

La forme parfois complexe de ces exemples nous amène à remarquer que si, dans la description du langage que nous donnons par la suite, le nombre de détails syntaxiques nous semblent être en désaccord avec l'objectif visant à décrire également les résultats de l'analyse, ce désaccord n'est qu'apparent : le détail des instructions n'intervient "pratiquement" que pour la rédaction des "énoncés" et de quelques modules qui seront utilisés comme des "énoncés" : ils sont situés aux points de sortie du graphe des appels de module, et dans toutes les autres phases de la description, on utilise ces objets par leur nom : on atteint bien ainsi le but poursuivi en introduisant le mode déclaratif.

Mais la frontière entre l'analyse et la programmation n'est pas si nette qu'il faille employer un langage pour l'une et un langage plus simple pour l'autre ; nous préférons proposer un langage unique, simple pour l'analyse mais pouvant permettre des formes plus complexes lors de la programmation.

2.5 MODE DECLARATIF ET CHOIX DES INSTRUCTIONS DU LANGAGE.

Le choix des instructions proposées dans le langage doit répondre au même souci : "éviter le plus longtemps possible à l'utilisateur d'avoir à contrôler le détail des opérations à entreprendre ou des informations

à traiter".

En plus des instructions classiques, affectation ou instruction conditionnelle (il n'y a pasd'instruction de branchement), il convient donc d'introduire des instructions "synthétiques" analogues à des énoncés, en ce sens que l'utilisateur n'aura pas à connaître le détail de leur exécution. Il sera cependant indispensable d'introduire également les instructions élémentaires suffisantes pour lui permettre de décrire l'équivalent de ces énoncés, s'il désire rester maître dans le détail de l'ordre d'exécution.

D'autre part, la forme même de ces instructions dépend essentiellement de la façon dont les concepts correspondants sont exprimés habituellement pendant l'analyse. Nous ferons un effort pour qu'elle soit essentiellement naturelle, car ce n'est qu'à cette condition qu'on peut en espérer une utilisation commode.

Ainsi, nous utiliserons des tables de décision pour décrire un ensemble de choix (cf. chap. 6), (Aubry B. 73). Ce sont des instructions de mode déclaratif , en ce sens que l'ordre exact d'évaluation des conditions sera inconnu de l'utilisateur : il sera déterminé par le traducteur. Une table de décision pouvant conduire à plusieurs ordres d'évaluation possibles, il ne s'agit donc pas d'un objet décrit en mode impératif, mais bien d'une déclaration des relations existant entre des conditions et des actions à entreprendre ; le traducteur pourra essayer de déterminer un ordre optimal, au point de vue de la place occupée par le programme objet par exemple, ou encore du temps moyen d'évaluation de la table. Mais pour permettre à l'utilisateur d'imposer un ordre d'évaluation s'il le désire, nous disposerons également d'instructions conditionnelles simples.

Pour demander un traitement répétitif, les langages de programmation nous ont habitués à mettre en évidence le nombre de fois qu'il faudra exécuter ce traitement et utiliser un compteur, ce qui est fortement lié au mode d'accès aux éléments d'un ensemble, puisque ce compteur sera souvent utilisé comme fonction d'accès par une variable indicée. Ainsi, pour calculer la somme des éléments d'un ensemble F , en Algol 60, nous ferons de F une tableau et nous écrirons :

```
..... ;  S : = 0 ;
pour I : = 1 pas jusque N faire S : = S + F (I) ; .........
```

I, N et l'écriture F (I) sont en fait des détails technologiques commodes, mais qui n'ont rien à voir avec l'énoncé du problème, ni la description de sa solution : partir de 0 et ajouter successivement à S chacun des éléments de F ; c'est-à-dire :

..... ; S : = 0

pour chaque élément de F faire S : = S + élément ;

Remarquons que cette forme d'écriture est beaucoup moins impérative que la précédente dans laquelle on impose l'utilisation d'un compteur et la façon de calculer l'adresse d'un élément du tableau depuis son origine.

Or il est très fréquent qu'un traitement itératif soit ainsi lié à un ensemble d'objets à traiter dans un certain ordre, — nous parlerons alors de file —, ou même dans un ordre arbitraire et pouvant varier, — nous parlerons alors d'un ensemble de valeurs. Nous introduirons les instructions "pour chaque" permettant de décrire le traitement de tous les éléments d'une file ou d'un ensemble, sans que l'on ait à contrôler le détail du bouclage. Là encore, l'utilisation du mode déclaratif dans cette instruction permettra au traducteur d'effectuer une optimisation du temps d'exécution. Cependant, des opérations élémentaires associées au traitement d'une file, "nouveau", "premier", "rang", "suivant", etc.... (cf. chap. 5), ou des boucles classiques avec une variable contrôlée, nous permettront d'imposer une façon d'exécuter ce traitement.

Il était tentant d'essayer de définir une instruction pour décrire les opérations de transferts entre objets structurés (Move et Move corresponding de Cobol par exemple) qui aurait été aussi une instruction de mode déclaratif. Mais cette fois, il existe un nombre beaucoup trop important de conventions possibles pour définir une telle instruction, ce que nous illustrerons aux chapitres 6 et 14. Nous avons donc préféré n'introduire qu'une instruction d'affectation simple entre structures, laissant le soin à l'utilisateur (à nous-mêmes également bien sûr) de définir d'autres types de transferts par des méta-modules.

Cette idée simple, a, en fait, une grande importance dans la conduite du projet : chaque fois que nous nous sommes trouvés devant un choix entre diverses formes d'instructions possibles, nous avons essayé d'éliminer des solutions restrictives qui imposeraient notre point de vue à l'utilisateur, quitte à proposer des solutions plus générales

mais qui risquent de demander un plus grand effort d'écriture pour leur utilisation. Il en est ainsi, par exemple, en ce qui concerne le contrôle des erreurs à l'acquisition d'une fichier, où nous avons éliminé la solution, simple à mettre en oeuvre, mais contraignante pour l'utilisateur, qui consisterait à n'entreprendre qu'une action unique à la rencontre d'une erreur quelconque dans un enregistrement : refuser l'enregistrement ; mais pour pouvoir user des facilités ainsi laissées à l'utilisateur celui-ci doit préciser quels contrôles il veut exercer et quelles actions il veut entreprendre en cas d'erreur, lorsqu'il fait appel à un métamodule d'acquisition, ou même, il peut être amené à écrire lui-même un nouveau métamodule d'acquisition.

2.6 MODE DECLARATIF ET DESCRIPTION DES INFORMATIONS.

Paradoxalement, on pourrait dire que les déclarations habituelles d'identificateurs ne sont pas vraiment faites en mode déclaratif : on doit préciser la nature exacte de l'ensemble des valeurs que pourra prendre une variable, c'est-à-dire son type, ou sa place précise par rapport à d'autres identificateurs dans une structure. Il serait souhaitable de pouvoir disposer de moyens de déclarations qui nous évitent d'avoir à entrer dans des détails de ce genre.

Une première façon de faire pourrait être de supprimer les déclarations : on peut remarquer qu'on n'a pas besoin de déclarer les tableaux en APL ; ceci augmente certainement la concision du langage P, mais certainement pas la facilité de communication des programmes. que serait-ce alors dans le cas de structures plus complexes ? D'ailleurs l'absence de déclaration conduit à ce que toutes les valeurs soient du même type ou alors qu'un objet puisse contenir des valeurs de type quelconque.

2.61 Déclarations de type.

Il nous semble important, au contraire, que tous les identificateurs ne soient pas de même type. Nous entendons par type d'un objet l'ensemble des valeurs que peut contenir cet objet. Nous utiliserons également les termes type d'un identificateur pour nommer le type de l'objet désigné par l'identificateur.

Ces types doivent même être nombreux et pouvoir être définis par l'utilisateur. Nous introduisons des déclarations de types par analogie à celui d'un identificateur déjà déclarés (comme , cf. chap. 4) ou par des déclarations explicites de type, en donnant la possibilité de construire de nouveaux types à partir des types existants : construction par restriction (définition d'intervalles, par exemple) ou par composition (produit cartésien d'ensembles pour définir les structures).

Les déclarations de type permmettent habituellement de ne pas avoir à répéter de nombreuses fois les mêmes renseignements, mais, également, lorsqu'on rencontre pour la première fois une notion, de lui donner un nom ainsi qu'à l'ensemble des valeurs qu'elle peut prendre, quitte à préciser ce type plus tard.

2.62 Métamodules et déclarations.

Les constituants du métalangage sont essentiellement des instructions de manipulations de textes, c'est-à-dire qu'ils permettent, à partir des conditions présentes au moment de la traduction, les paramètres effectifs des métamodules par exemple, de produire un texte plutôt qu'un autre, en ne se préoccupant que de la forme du texte généré et non de son contenu. En particulier, le texte Civa, décrit par un métamodule, peut très bien contenir des déclarations, ce qui permet aussi de décrire des structures paramètrées, c'est-à-dire des types dont la forme ne sera complètement précisée que lors d'une référence à ce métamodule . Ceci, ajouté à la présence de types conditionnels, devrait permettre une grande souplesse dans la présentation des déclarations.

2.7 PREMIERE CONCLUSION.

Notre projet consiste essentiellement en la mise au point d'un langage qui s'applique aussi bien à l'analyse qu'à la programmation, l'exploitation ou la maintenance des unités de traitement importantes.

Il s'appuie sur une technique de description modulaire, qui vise à décrire une tâche de façon arborescente : plus on s'approche de ses extrémités et plus on entre dans les détails du traitement ou des informations. Cette description nécessite un langage d'analyse approprié. L'article (Rolland C. 71) présente une réalisation qui met en évidence cet aspect du projet. De ce langage naturel, découle un langage de description des résultats de l'analyse (1), tels que les textes sources se déduisent facilement des descriptions du dossier d'analyse. Ce langage constitue le langage de programmation Civa.

.../...

(1) Il n'y a pas là contradiction avec le but indiqué dès le début de cette introduction : "définir un langage unique....". La description de l'analyse doit être faite sans contrainte excessive, elle doit être lisible par des non-informaticiens, elle doit donc être rédigée le plus naturellement possible. Le langage Civa ne contient que des éléments dont la nécessité semble naturelle et ne diffère du langage utilisé dans le dossier que par une codification. (Sans aucune réorganisation). C'est le texte même de cette description qui devient un module de programmation. Et si, pour un utilisateur donné, le langage utilisé pour les dossiers diffère de celui utilisé pour la "programmation", ce n'est qu'une question d'habitude. Il ne devrait rapidement plus en être ainsi dès que les mauvaises habitudes seront perdues.

Il s'appuie également sur un important service de gestion de bibliothèque. En effet, on doit trouver dans ces bibliothèques toutes les unités de nomenclature des applications (classes, modules, méta-modules).

Enfin, le système d'exploitation utilisé doit être adapté pour assurer l'exploitation des chaînes de traitement obtenues et fournir des mesures d'efficacité.

Notre projet devrait donc faciliter la mise en oeuvre et l'exploitation des travaux "d'application", c'est-à-dire des travaux importants, demandant une analyse poussée, aboutissant à une série d'unités de traitement, qui seront exploitées pendant un temps long, au cours duquel le problème pourra évoluer.

Nous avons vu les avantages de la modularité pour la programmation. Notre projet devrait également faciliter le passage de l'analyse à la programmation, de la programmation à l'exploitation, l'utilisation de l'existant et la "maintenance". Le passage de l'analyse à la programmation est effectivement facilité : il nécessite habituellement une restriction de la description (si on utilise cobol en particulier) pour obtenir un texte absolument linéaire (programme) d'une description qui ne l'est pas toujours ; ici c'est la <u>description exacte</u> du résultat de l'analyse qui formera les constituants d'une unité de traitement, sans nécessiter de restructuration. Le passage de la programmation à l'exploitation est facilité par les modules de commande (cf. 1.8) qui suppriment cette frontière.

Pour l'utilisation de l'existant : si cet existant est rédigé sous forme de modules et de classes, il n'y a aucun problème, si ce n'est qu'il est indispensable de disposer d'une documentation précise de la bibliothèque d'une application (des commentaires attachés aux déclarations d'identificateur permettent d'établir cette documentation) (cf. Chap. 4); si cet existant est déjà rédigé dans un autre langage, il doit pouvoir être utilisé : les sous-programmes Fortran, Cobol, Symbol, pourront être appelés comme des modules. La génération automatique d'une classe à partir d'une "division donnée" de cobol est à l'étude.

Quant à la maintenance, le découpage même d'une application en modules et classes est un outil de maintenance. Un mode de compilation en "mise au point" permettra une mise en oeuvre facile des différents concepts de mise au point (extractions d'informations, instructions "quand", etc...) (cf. chap. 10). De plus les instructions de modifications de textes sources utilisables dans les modules de commande (cf. chap. 10) permettront des mises à jour faciles. Enfin, le système de traitement du module de commande (cf. 21), évite à l'utilisateur d'avoir à se préoccuper de l'état dans lequel se trouvent les composants de son application (source, compilé, compilé en "mise au point", édité), les transitions étant gérées automatiquement.

CHAPITRE 3

CONCEPTION DE LA REALISATION

3.1 REMARQUES GENERALES.

3.2 COMPILATION.

3.21 Organisation.

3.22 Problèmes particuliers.

3.3 EDITION ET PREEDITION DE LIENS.

3.4 INTERPRETATION DU MODULE DE COMMANDE.

CHAPITRE 3

CONCEPTION DE LA REALISATION

3.1 REMARQUES GENERALES.

La conception de la réalisation se devait d'être une mise en pratique de l'esprit dans lequel nous proposons de concevoir une application. Nous avons d'abord utilisé une méthode de conception ascendante (cf. introduction), commençant par examiner séparément comment résoudre certains problèmes, tels que la traduction des expressions arithmétiques ou le traitement des débuts et fins de module etc... Nous avons utilisé ensuite une méthode de conception descendante pour construire la réalisation. Mais rapidement, nous avons buté sur l'absence de cadre dans lequel inscrire nos travaux et d'outils pour les décrire. Pour le faire, nous avons choisi d'utiliser Métasymbol, langage de méta-assemblage sur CII 10070, sous le système d'exploitation SIRIS 7 (CII), essentiellement parce qu'il contient, sous des formes plus ou moins cachées, de grandes possibilités de description modulaire des actions et des informations. Nous avons choisi également de produire des "modules objets", c'est-à-dire un texte qui sert d'entrée à l'éditeur de liens de SIRIS 7, constitué essentiellement de directives de chargement (avec ou sans translation par exemple) destinées à être interprétées par l'éditeur. Pour dégager les programmeurs des contraintes dues à la présentation de ces modules-objets, un jeu important de procédures élémentaires en Métasymbol a été mis au point (Ducloy J. 73) ; utilisant ces procédures, d'autres procédures permettant de générer des instructions machines, des séquences, des réservations, etc... ; utilisant celles-ci, des procédures pour la traduction d'instruction Civa. Seulement après, nous avons organisé la structure du compilateur. Il s'agit donc bien d'une méthode de conception ascendante, du moins jusqu'au début de l'écriture proprement dite du compilateur.

Il convient de bien préciser que c'est le compilateur qui est écrit en Métasymbol, il comprend des appels des procédures citées précédemment. Lors de la traduction du compilateur, une séquence d'instructions est générée à chacun de ces appels. Cette séquence permettra, lors d'une

compilation Civa, de générer un texte à mettre dans un module objet.

3.2 COMPILATION.

3.21 Organisation.

La compilation d'un module, (cf. chap. 16) et (Ducloy J. 73), se déroule en deux phases :

- codification, chargée de résoudre tous les problèmes de transformation des noms utilisés dans le programme source en adresses relatives à l'origine d'un module objet, et d'identifier chacune des instructions ;
- génération du module objet proprement dit.

Pour des raisons exposées au chapitre 9, le traitement des éléments du métalangage est mêlé à la phase de codification.

Un module conduit à la production d'une zone d'instructions et de constantes, appelée module-objet, repérée par une définition externe, ainsi qu'à une référence à une zone contenant les objets de taille connue à la compilation (et à la taille de cette zone). Pour les objets dont la taille est inconnue à la compilation, un pointeur situé dans la zone précédente permet d'y accéder. La zone des objets de taille variable est gérée dans une mémoire chaînée attribuée par pages (Bazerque G. 69), (Lion F. 73), (Perrot D. 73).

La compilation d'une classe ne donne lieu qu'à une codification ; elle consiste à placer chacun des objets déclarés dans la classe par rapport à l'origine de chacune des zones associées à la classe. La codification traite également les relations utilise. Le texte obtenu est conservé.

La compilation d'une classe ou d'un module conduit également à la construction d'un enregistrement du fichier descriptif des classes et des modules d'une application. Ce fichier contient, pour chaque unité de nomenclature de l'application, des renseignements sur les noms et tailles de ses différentes zones et la liste des modules qu'elle appelle et des classes qu'elle utilise.

3.22 Problèmes particuliers.

Le traitement des instructions de transfert a particulièrement retenu notre attention (Benamghar L. 73) ainsi que la traduction des expressions arithmétiques et l'optimisation possible du code objet (Ducloy J. 73). De même, la traduction des instructions "pour chaque" devrait tenir compte du fait qu'il s'agit d'une instruction d'itération permettant un traitement global des files, pour mettre en oeuvre les possibilités d'optimisation que cela comporte.

La traduction des tables de décisions (cf. chap. 18) et (Aubry B. 73) est un point particulièrement important. Le traducteur actuel utilise une méthode de séparation et d'évaluation progressive (Branch and Bound) ; elle permet de minimiser le nombre de tests à décrire pour pouvoir évaluer complètement la table. Ce n'est pas suffisant, car cette méthode, la plus courante, ne tient compte ni des fréquences différentes de réalisation des réponses à ces tests, ni de leur coût d'évaluation respectif. Pour les tables d'utilisation très fréquente, une première traduction (effectuée en mode "mise au point") permettra d'ajoindre à la description proprement dite de la table en langage objet, des ordres de mise à jour de mesures statistiques (fréquences de réalisation par exemple). Ces mesures pourront alors être utilisées lors d'une traduction ultérieure de la même table, pour déterminer un ordre d'évaluation des conditions, donc une traduction de la table, minimisant le temps moyen d'évaluation complète de la table.

La solution proposée à ce sujet dépasse largement le cadre du projet Civa et peut être appliquée à toutes les tables de décision, indépendamment du projet lui-même.

3.3 EDITION ET PREEDITION DE LIENS.

Les différents modules objets issus des classes et des modules intervenant dans une unité de traitement sont regroupés en un segment unique, appelé module de chargement, par l'intermédiaire de l'éditeur de liens de Siris 7. Dans ce module de chargement, tous les noms ont été transformés en des adresses par rapport à son origine, c'est-à-dire qu'il n'y a plus de références externes insatisfaites.

Mais, à l'édition de liens, les références externes correspondant aux zones de variables de taille fixe ne peuvent être satisfaites puisqu'il n'y a encore eu aucune réservation pour ces zones. De plus l'éditeur de liens ne tient pas compte du graphe des appels des modules d'une unité de traitement pour diminuer la place qu'elle occupe en mémoire centrale.

Il nous a donc semblé opportun d'introduire une étape supplémentaire entre la compilation et l'édition de liens que nous avons appelée pré-édition de liens. Elle pourrait être insérée à l'éditeur lui-même ; ceci nous aurait amené à modifier considérablement, si ce n'est recommencer, l'éditeur de liens de SIRIS 7, solution qui nous a fait reculer.

A partir du fichier descriptif des classes et des modules, le prééditeur construit le graphe des appels de modules et le graphe de coexistence des zones variables de taille fixe. Il en déduit un arbre de recouvrement pour les modules objets, et fixe une adresse pour chacune des zones de variables de taille fixe, en cherchant à minimiser l'occupation totale de la mémoire. Il construit un module objet représentant les implantations relatives de ces zones et il appelle l'éditeur de liens, en lui fournissant l'arbre de recouvrement obtenu et le nom de tous les modules objets nécessaires.(cf. Chap. 25) et (Dendien J. 73 a et 73 b).

l'intérêt essentiel de cette façon de procéder est que le travail de "réalisation des noms", c'est-à-dire la suite de transformations à faire subir à un identificateur du texte source pour lui faire correspondre une adresse réelle en mémoire, est effectué avant l'exécution, de manière statique, ce qui fait perdre un temps non négligeable à la préparation des unités de traitement mais accélère beaucoup l'exécution. Quant au problème de la construction automatique d'un arbre de recouvrement optimal, il dépasse largement le cadre de cette réalisation et les solutions proposées pourraient être appliquées ailleurs.

3.4 INTERPRETATION D'UN MODULE DE COMMANDE.

Le travail à exécuter est soumis au système par l'intermédiaire d'un module de commande (cf. 1.8). Celui-ci a essentiellement pour rôle d'assurer l'enchaînement de l'exécution des unités de traitement et de

demander au système d'exploitation les ressources nécessaires. Mais il doit également satisfaire toutes les demandes des utilisateurs :

- adjonction, suppression, remplacement ou modification des modules et des classes de son application (cf. 1.7) et (Payafar M. 71) ;
- évaluation de déclarations ou d'instructions Civa autres que les appels de module ;
- compilation, en mode "mise au point" ou "exploitation" et éditions de liens demandées explicitement, ou, le plus souvent, implicitement à la rencontre d'un appel de module qui est alors le premier module d'une chaîne de traitement (module directeur).

Il doit enfin assurer l'accès aux identificateurs réservés de la classe d'application du système (C_S) et leur modification (C_M).

Il entretient, dans le fichier descriptif des classes et des modules, les indicateurs d'état de ces unités. Un module ou une classe peut être à l'état "source", "compilé en mode mise au point" ou "compilé en mode normal" ou encore "édité" (cf. chap. 21). Le module de commande effectue les différentes transitions.

A la rencontre d'un appel de module, celui-ci doit être dans l'état "édité", sinon il faut l'y mettre. Un module X, ne peut passer à l'état "édité" que si tous les modules appartenant à $\widehat{A}$ (X) et toutes les classes de $\widehat{U}$(X) sont à l'état "compilé" : si ce n'est pas le cas, le module de commande doit d'abord lancer les compilations nécessaires. Lors d'une modification du texte source d'un module ou d'une classe Y, cette unité retombe à l'état "source" quel que soit son état antérieur (il s'agit en fait d'une nouvelle unité, même si elle a le même identificateur que la précédente, qui est détruite). Un module "édité" touché par cette modification revient à l'état "compilé" - c'est-à-dire un module "édité" X, tel que X $\widehat{A}$ Y ou X $\widehat{U}$ Y selon le cas.

C'est également au niveau du module de commande qu'on assure la gestion de la bibliothèque de l'application. Celle-ci contient :

- un fichier des textes sources des modules, classes, métamodules et classes de commande ;
- la traduction de la classe de l'application et éventuellement les objets qui y sont déclarés ;
- un fichier des classes "compilées" ;
- un fichier des modules-objets ;

- un fichier des modules de chargement ;
- un fichier utilisé par les sauvegardes (cf. mise au point) ;
- le fichier descriptif des classes et des modules ;
- les fichiers de l'application, en format interne : ils sont crées par les chaînes de traitement en utilisant un métamodule d'acquisition ; ils sont édités en utilisant un métamodule d'édition.

REFERENCES DE LA PREMIERE PARTIE

ARSAC J. 72 Un langage de programmation sans branchements. RIRO B / 2 Juin 72 page 3 - 34

71 Quelques remarques et suggestions sur la justification des algorithmes. RIRO B/3 1971.

AUBRY B. 73 Traduction et optimisation des tables de décision dans le projet CIVA. Thèse de 3ème Cycle NANCY Février 1973.

BAZERQUE G. 69 Implantation des objets du langage PL1 sur une calculatrice RIRO B 1 Mai 69.

BENAMGHAR L. 73 Mouvements d'informations dans le projet CIVA. Thèse de 3ème Cycle NANCY Juin 1973.

CHABRIER J. J. 73 Acquisition et édition des fichiers, entrées-sorties et analyse des données dans le projet CIVA. Thèse de 3ème Cycle NANCY Mai 1973

CHAMBERLIN

C. I. I. Symbol Métasymbol Manuel d'utilisation CII C 900 952.

CODASYL a) Systems Comittee "Feature Analysis of Generalized Data Base Management systems". IFIP Administrative Data Processing Group - Amsterdam.

71 b) Systems comittee. "Introduction to" feature Analysis of Generalized Data Base Management systems". C. A. C. M. Vol. 14 n° 5 p. 308 - 318 May 1971

DENDIEN J. 73 a) Pré-édition de liens dans le projet CIVA
Doctorat d'Ingénieur NANCY 1 Mars 1973

b) Gestion de mémoire dans un programme modulaire
Congrès AFCET Grenoble nov. 72.

DIJKSTRA E. W 72 Notes on structured programming in structured programming
A. P. I. C. Studies in Data Processing, Academic Press (1972).

DUCLOY J. 73 Compilation dans le projet CIVA.
Doctorat d'Ingénieur NANCY 1 Mars 1973

GALLAIRE 72 Projet d'une étude sur les systèmes Software - Hardware de mise en oeuvre des langages à assignation unique.
Journées d'études sur les recherches en structure de machines et architecture de systèmes. RENNES Nov. 72 (IRIA ed.).

GENUYS, POYEN, VAUQUOIS 60 Traduction du rapport Algol 60. Chiffres 3.1 Mars 60.

HERTSCHUH N. 73 Méthode d'analyse et projet CIVA
Thèse de 3ème Cycle NANCY 1973.

KNUTH D. E. 71 and FLOYD R. W.
Notes on avoiding GOTO statements. Information Processing letters, 1 (1971), 23 - 31 North Holland Publishing.

LION F. 73 Gestion des objets de taille variables dans le projet Civa. Thèse de 3ème Cycle - Université Nancy 1 (A paraître).

NATO Science Comittee 69
Software Engineering Techniques (Garmish 1968).
P. NAUR, B. RANDELL, Editors 1969.

NATO Science Comittee 70
Software Engineering Techniques (Rome 1969)
J. BUXTON, B. RANDELL, Editors 1970.

PARNAS D. L. 72 A technique for software module specification with examples. C. A. C. M. 15, 5 (May 1972)

PAYAFAR M. 73 La modification des textes dans le projet CIVA NANCY 1971.

REIX R. 71 L'analyse informatique de gestion Tome 1 DUNOD 1971.

PERROT D. 73 Codification et compilation dans le projet CIVA Thèse de 3ème Cycle NANCY 1973.

ROLLAND C. 71 Essai d'un langage d'analyse. Application à un traitement de gestion administrative. Bigen n° 14 1971.

VAN WINJGAARDEN A. 69 Report on the algorithmic language Algol 68 - Math. Centrum Amsterdam 1969.

Traduction dans "Définition du langage algorithmique Algol 68. BUFFET J., ARNAL P., QUERE A. 2diteurs. HERMANN Paris 1972

WEINBERG G. 71 The psychology of Computer Programming. Van Nostrand (1971).

WIRTH N. 71 Programm developpment by stepwise refinement C. A. C. M. 14, 4 (April 1971).

CROCUS 74 Principes de conception des systèmes d'exploitation A paraître Début 1974 - Paris.

CHAPITRE 4

DESCRIPTION DES INFORMATIONS

4.1. INTRODUCTION.

4.2. TYPE DES OBJETS.

4.21. Notion de type.

4.22. Déclarations.

4.23. Types prédéfinis.

4.24. Types définis par une puissance d'ensemble : files.

4.25. Types définis par un produit d'ensemble : structures.

4.26. Types définis par restriction.

4.27. Types définis en extension.

4.3. INDICATION DU TYPE D'UNE VARIABLE.

4.31. Déclaration de type.

4.32. Déclaration par analogie.

CHAPITRE 4

DESCRIPTION DES INFORMATIONS

4.1. INTRODUCTION.

Nous avons vu au chapitre 1 que les informations sont décrites modulairement par l'intermédiaire de classes et nous avons étudié les règles qui régissent la durée de vie des objets utilisés. Il nous reste donc essentiellement à préciser quels objets sont utilisables en CIVA et comment les déclarer. Ces objets sont classiques : constantes et variables entières, décimales, réelles, etc... ou structurées, d'autres le sont moins, telles que les files d'objets ou les ensembles. Les files et les ensembles seront étudiés au chapitre 7, dans lequel nous présentons ces objets et leur traitement. Nous précisons ici la notion de type et la façon de définir de nouveaux types.

Cela fait, tous les problèmes posés par la description des informations ne seront pas entièrement résolus pour autant.

Un identificateur et un type associés à une information, sont des renseignements bien insuffisants pour la décrire complètement. Au cours de l'analyse des données - et c'est dès ce moment, que, grâce à la modularité des classes, sa description peut apparaître sous une forme qui ne sera modifiée que par adjonction de renseignements nouveaux - une information, telle que le numéro matricule d'un employé, par exemple, peut être caractérisée par un cortège plus important de renseignements, consignés en général dans le dossier d'analyse, tels que :

a) relations que la grandeur correspondante doit vérifier lors de son acquisition ou lors de chaque attribution de valeur.

   Ces relations peuvent être exprimées par le type de la variable associée à cette grandeur : c'est le cas, comme nous le verrons, lorsqu'une valeur doit rester inférieure à un seuil fixé ; d'autres ne le peuvent pas comme, pour notre exemple, la correspondance biunivoque entre un nom et un numéro matricule.

b) quelle action entreprendre lorsque ces relations ne sont plus vérifiées ?

c) par qui cette information est-elle créée ?

d) par qui est-elle modifiée ?

e) à qui est-elle destinée ?

f) si elle est transmise à l'extérieur du calculateur, sous quel format ? etc...

Tous ces renseignements apparaissent également nécessaires à l'analyste, et il les consigne, en général dans l'ordre dans lequel leur nécessité lui apparaît, dans un dossier d'analyse sans se préoccuper de savoir s'ils doivent ou non figurer dans un "programme". Le texte source des modules et des classes doit donc pouvoir accepter ces renseignements. Ils pourront être décrits par l'intermédiaire de commentaires particuliers. Ces commentaires seront toujours attachés à un identificateur. Nous appellerons un tel commentaire la documentation de cet identificateur. Il s'agit bien d'un commentaire au sens classique dans la mesure où le traducteur des classes et des modules ne s'en préoccupe pas, mais en fait, une documentation pourra être structurée, essentiellement pour indiquer la nature des renseignements qu'elle contient.
Ces renseignements sont ceux cités dans le paragraphe précédent pour les identificateurs déclarés dans les classes et les modules. Ils peuvent être associés à un identificateur seul, ou à une classe, et, partant à tous les identificateurs de la classe.
Pour les noms de module, la documentation sert à préciser la fonction remplie par ce module, c'est-à-dire ses spécifications. Des normes de rédaction de la documentation des modules sont à l'étude pour permettre une utilisation plus commode d'une méthode de conception ascendante, s'appuyant sur un inventaire des modules existants. Des services particuliers permettent de déduire, des documentations ainsi définies, la documentation complète d'une chaîne de traitement qui est la réunion des documentations des identificateurs qu'elle met en jeu, éventuellement classée. Ils permettent également d'obtenir la nomenclature d'une application.

L'étude de ces problèmes de documentation et de la réalisation des services correspondants est actuellement en cours (BARTHELEMY C. et PHILIPPE).

Pour les points a) et b) de l'énumération précédente, dire qu'une grandeur doit vérifier des relations et qu'une action doit être entreprise dans le cas contraire, c'est demander au système d'exercer un certain nombre de contrôles, chaque fois qu'on modifie ou qu'on accède à cette information. Ces contrôles peuvent donc intervenir à 3 moments différents :

- à l'entrée de données en ordinateur ou "acquisition"
- pendant l'exécution d'une unité de traitement : contrôles à l'exécution
- à la sortie des résultats (contrôles à l'édition).

A l'acquisition d'un fichier, c'est-à-dire lors de l'entrée en ordinateur des informations correspondantes et du passage du format externe au format interne, un certain nombre de contrôles peuvent être demandés : contrôles de syntaxe, contrôles de séquence, contrôles de vraisemblance, etc... Les relations à vérifier seront décrites explicitement dans la description externe d'un fichier. Cette description, qui peut également être modulaire indique essentiellement le format externe de l'information, les contrôles à exercer et quels modules appeler lors de ces contrôles. Elle sert d'entrée à un métamodule d'acquisition (2.32., chapitre 9).

La même démarche peut être employée pour décrire les contrôles à effectuer à l'édition. L'étude systématique des contrôles possibles à l'acquisition ou à l'édition, de leur description et des métamodules d'acquisition et d'édition est faite dans (CHABRIER J.J., 73) et présentée aux chapitres 11 à 14.

Les contrôles à effectuer pendant l'exécution doivent être exercés à chaque modification de la grandeur à contrôler. Des instructions particulières ("quand"), permettront de déclarer et d'indiquer quelle action entreprendre lorsque cet événement survient. Nous reviendrons, au chapitre 6, sur cette notion en traitant de la mise au point des unités de traitement pour laquelle elle s'avère particulièrement utile (DUCLOY J., 73).

4.2. TYPE DES OBJETS.

4.21. Notion de type.

Nous distinguerons essentiellement deux sortes d'objets :

- des valeurs, ou constantes, susceptibles d'être "contenues" par des variables ou "désignées" par des identificateurs : ainsi, si l'on a A entier = 5, l'identificateur A "désigne" la constante 5 ;

- des variables ; une variable "contient" une valeur à un instant donné ; la valeur contenue par une variable peut changer (affectation). Une variable est "désignée" par un identificateur. On peut donc dire qu'une variable est un emplacement.

Nous appelons type d'une variable l'ensemble des valeurs que peut contenir cette variable. Dire qu'une variable est d'un certain type, c'est dire qu'elle peut contenir des valeurs appartenant à ce type. Nous le faisons par une déclaration d'un identificateur (Nous avons vu au chapitre 1 que la localisation de cette déclaration délimite sa portée et détermine la durée de vie de l'objet créé).

Définir un langage de programmation, c'est, entre autres choses, définir les types de variables qu'il peut décrire et l'ensemble des opérations que l'on peut effectuer sur elles. A la définition du langage, certains types peuvent être fixés définitivement : ainsi, l'écriture A entier ; indique que A "désigne" une variable qui peut contenir une valeur qui est un élément de l'ensemble des entiers. Certains langages n'acceptent que ces types prédéfinis. Il est cependant utile de pouvoir définir de nouveaux types.

4.22. Déclarations.

Une déclaration de constante s'écrit sous la forme :

< identificateur > < type prédéfini > = < notation de constantes >.

Les types prédéfinis sont décrits ci-dessous :

Exemple A entier = 15 ; B dec 9 (5) V 9 (2) = 45.37 ;
C réel = 3.1416 ;

sont des déclarations de constantes.

Une déclaration de variable s'écrira sous la forme :

<déclaration de variable>:: = <identificateur> <type>[<liste de valeurs>].

ou, plus généralement, pour les déclarations multiples :

<décl. multiple>:: = <identificateur> [, <identificateur>]* <type> [<liste de valeurs>].

<type> est une définition du type à associer à l'identificateur déclaré.

<type> :: = <type prédéfini> | <type file> | <structure> | <code> | <analogie> .

<type prédéfini> :: = entier | ent | réel | complexe | de précision | booleen | caractère | car | <decimal> .

<décimal> :: = décimal <modèle> dec <modèle> .

<modèle> :: = 9 (m) V (n).

< m > :: = <entier> (inférieur à 16).

< n > :: = <entier> (inférieur à 15).

<type file> voir chapitre 7.

<structure> :: = struct (<déclaration de variable>[,<déclaration de variable>]*).

<code> :: = code <liste de valeurs> .

<liste de valeurs> :: = (<notation de constante>[,<notation de constante>]*).

Remarque : les valeurs doivent appartenir au même type.

analogie :: = type identificateur . (cf. 4.32.).

4.23. Types prédéfinis.

Les types prédéfinis sont entier, réel, complexe, double précision, booleen, caractère.

L'ensemble des valeurs désigné par l'un de ces types est précisé selon le matériel utilisé :

entier : compris entre -2^{31} et $+2^{31}-1$
par exemple pour la réalisation actuelle ;

réel : se présente sous la forme $\pm m.n\ E \pm e$, avec
8 chiffres significatifs pour la mantisse,
2 chiffres pour la caractéristique ;

complexe : a + ib représenté par a, b, a et b étant réels ;

double précision : comme réel mais avec 16 chiffres significatifs ;

booleen : 2 valeurs "Vrai" et "Faux".

Les autres types prédéfinis concernent les nombres décimaux. Il y a autant de types décimaux que de présentations possibles (présentation interne). Nous utilisons la notation :

9 (n) V 9 (m)

dans laquelle le nombre de 9 écrit devant V ou désigné par n est le nombre de chiffres avant la virgule, le nombre de 9 écrit derrière V ou désigné par m est le nombre de chiffres après la virgule.

exemple : décimal 9 (5) V 9 (2)
définit un type dont la valeur 52327,50 est un élément.

4.24. Types définis par une puissance d'ensemble : files.

Les files seront étudiées plus précisèment au chapitre 7.
Une file d'éléments de type t peut prendre ses valeurs dans l'ensemble $\{t\}^n$ où n est la taille de la file.
La définition d'un type par une puissance sera demandée par le mot file, suivi, en général de la taille de la file et du type t.

file (25) entier et file (24) entier
sont deux types différents.

Nous définissons trois catégories de files :

- les files de taille fixe :
  les valeurs pouvant être contenues par une variable de type "file de taille fixe n d'éléments de type t" appartiennent toutes à $\{t\}^n$.
  Une file de taille fixe est déclarée sous la forme :

  <identification> file (<taille>) <type>.

- les files de taille bornée :
  les valeurs pouvant être contenues par une variable de type "file de taille bornée m d'éléments de type t" appartiennent à $\bigcup_{i=1}^{m} \{t\}^i$
  Une file de taille bornée est déclarée sous la forme :

  <identification> file (max <taille>) <type>, où <taille> fixe la valeur de m.

Nous définissons une fonction standard taille actuelle (f) qui a pour valeur la taille de la constante file convenue au moment de l'appel de cette fonction par la variable de type file f.

- les files de taille variable :

le nombre de leurs éléments est quelconque. Les valeurs pouvant être convenues par une variable de type "file de taille variable" appartiennent à $\bigcup_{i=1}^{p} \{t\}^i$ où p est une limite définie à l'implémentation, si cela est nécessaire.

Ces files sont déclarées sous la forme :

<identificateur> file < type>.

La fonction "taille actuelle (f)" est définie de la même manière pour les files de taille variable (1).

Dans les déclarations ci-dessus, une < taille > désigne un nombre entier ou une identification de constante. < Taille > peut également être une identification de variable ou une expression. Dans ce cas, cette variable doit contenir une valeur (ou cette expression doit être calculable) au moment de la déclaration. La taille ainsi déterminée ne sera plus modifiée dans la suite de l'exécution, il s'agit donc bien de files de taille fixe.

Les files seront étudiées plus précisèment au chapitre 7.

Exemples :

```
MEM   file (25) caract ;
ARCHI file (max 57) décimal  999 V 99 ;
TSP   file entier ;
VSO   file (N) réel ;
AN    file (max 20) entier ;
```

sont des déclarations de files.

(1) variable signifie variable pendant l'exécution et non pas variable d'une exécution à une autre (tableaux variables d'Algol), les tableaux d'Algol sont en fait des tableaux de taille fixe : la taille d'une variable de type tableau ne change jamais ; la notion de "tableau variable" d'Algol permet de déclarer des variables de type tableau de taille fixe, mais dont les tailles seront différentes d'une élaboration à l'autre de la même déclaration.

4.25. Types définis par un produit d'ensembles : structures.

Nous définissons une composition de types par un produit d'ensembles : une structure à deux champs de type t1 et t2 est une variable pouvant contenir des valeurs de $\{t1\} \times \{t2\}$.
Une variable de ce type est déclarée par :

<identification> struct([<identification>]<type>,
[<identification>]<type>).

Nous définissons de même une structure à n champs ($n \geqslant 1$).

Les identifications de champs de la structure permettent de définir des fonctions d'accès aux valeurs des champs. Les types de champs d'une structure sont quelconques, en particulier, ils peuvent être du type file, ou comme nous le verrons en 4.32., des types déclarés.

Exemples :

```
Personne struct (nom file (max 20 ) car,
                 prénom file (max 15) car,
                 adresse file (max 50) car) ;

individu struct (état civil struct (nom file (max 20) car,
                                    prénom file (max 20) car),
                 adresse file (max 50) car) ;
```

sont deux déclarations de variable structurée.

Dans le texte d'un module ou d'une classe, un champ d'une structure peut être désigné par l'identificateur qui lui est associé dans la déclaration de la structure, et en cas d'ambiguité par les identificateurs des champs qui le précèdent dans la structure.
Ainsi, avec les deux déclarations ci-dessous :

état civil, nom de état civil, nom de individu, nom de personne, adresse de individu, etc... permettant de désigner des champs sans ambiguïté ; prénom, nom, ou adresse seuls sont ambigus.
Les ambiguïtés peuvent exister du fait de la déclaration de deux champs de même identification dans deux structures différentes ou du fait de la déclaration d'un champ et d'une variable de même identification.

4.26. Types définis par restriction.

Si une variable ne peut prendre ses valeurs que dans un sous-ensemble de l'un des ensembles pouvant être défini par les déclarations précédentes, et si on peut définir ce sous ensemble par une propriété, nous dirons que l'on définit ainsi un type par restriction. Les propriétés utilisables seront représentées par des expressions booléennes.

Exemple : entier < 5 000, réel /= 0 ; décimal > 35,5

sont des définitions de type par restriction.

4.27. Types définis en extension.

Lorsqu'une variable ne peut prendre ses valeurs que dans un ensemble restreint de valeurs, on pourra définir son type par la liste de ces valeurs.

Exemple : file (max 11) car (marié, célibataire, veuf, divorcé) ;

La liste des notations de constantes définit les valeurs de l'identificateur déclaré. Ainsi, une variable du type décrit ci-dessus ne peut contenir que des files de 11 caractères dont les valeurs sont citées explicitement entre les deux parenthèses. Elles sont éventuellement complétées par des blancs.

La liste contient des notations de valeurs du type indiqué, dans leur forme habituelle.

Dans le cas particulier des entiers, nous considérons la déclaration

I code (marié, célibataire, veuf, divorcé) ;

comme une simplification d'écriture représentant la suite de déclarations

marié entier = 1 ; célibataire entier = 2 ; veuf entier = 3 ;
divorcé entier = 4 ;
I entier (1, 2, 3, 4).

Une déclaration du type entier code s'écrit :

< identificateur> code (< identificateur> [,< identificateur>] *).

L'instruction d'affectation de valeur à une variable d'un type défini par restriction ou en extension n'est définie que pour les valeurs ainsi caractérisées : toute tentative d'affectation d'une valeur n'appartenant pas au type considéré entraine une erreur (arrêt et message).

4.3. INDICATION DU TYPE D'UNE VARIABLE.

Dans ce paragraphe, nous ne définissons pas de nouveaux types, mais simplement des moyens de désigner les types.
Nous avons vu en 4.22., que la déclaration d'une variable s'écrit en général : <identificateur><type> [<liste des valeurs>].

Cela peut paraître fastidieux à l'utilisateur de préciser chaque fois complètement le type des informations qu'il décrit, lorsque de nombreux types sont identiques. De plus, cela se prête mal à la modularité de la description des informations dans laquelle on voudrait définir progressivement une structure par exemple. Deux autres façons d'indiquer le type dans une déclaration nous permettront de rendre plus souple la description des informations.

4.31. Déclaration de type.

Au cours de la conception descendante de l'organisation d'un ensemble d'informations, il semble important de pouvoir décrire progressivement les résultats obtenus. En particulier, il est indispensable de pouvoir dire qu'une variable structurée est composée d'un certain nombre de champs et de les nommer (leur nom est en général représentatif de leur fonction), quitte à ne préciser qu'ultérieurement, par exemple dans une autre classe, le type exact de ces champs (cf. l'exemple de 1.2.).
Pour préciser le type de ces champs, nous serons amenés à déclarer un nouveau type, sans pour autant déclarer un objet de ce type, c'est-à-dire sans pour autant faire de réservation.
Une déclaration de type s'écrit :
<déclaration de type> :: = type <identificateur><type>
Exemples : type A entier < 5 000 ;
type B struct (C entier, D file (15) dec 9 (4)
V 9 (2));
sont des déclarations de type.

4.32. Déclaration par analogie.

Il reste à pouvoir dire qu'un objet que l'on déclare est d'un type déclaré, c'est-à-dire qu'il est du même type que celui désigné par un identificateur.
Nous ferons à ce niveau l'abus de langage fréquemment utilisé, qui consiste à prendre pour identificateur du type celui d'un objet

déclaré de ce type ; nous voudrons donc pouvoir dire également qu'un objet est du même type qu'un autre déjà déclaré.
Cette seconde façon d'indiquer le type d'un objet est particulièrement utile dans les phases de conception ascendante au cours desquelles on utilise ce qui est déjà décrit. Lorsqu'un identificateur A est déjà déclaré (il est donc d'un certain type), une déclaration de B par analogie à A permettra de déclarer un nouvel objet du même type que A.
Une déclaration par analogie s'écrit :

<identificateur> type <identificateur> .

Elle signifie que l'on déclare un objet, "désigné" par l'identificateur de gauche, A par exemple, avec le type désigné par l'identificateur de droite, B par exemple.
B peut être l'identificateur d'un objet : A est alors du même type que B ;
B peut être l'identificateur d'un type : A est alors du type B.
Enfin, dans la déclaration d'une structure, la rencontre d'un identificateur de type T, comme déclaration de champ sera considérée comme une simplification d'écriture d'une déicaration par analogie qui s'écrirait T type T.
Exemples : (cf. 1.2.).

```
Module  EXEMP ;
     utilise A ;
     F COM file type COM ; .... texte du module EXEMP... fin module ;

Classe A ;
     utilise B ;
     type CØM struct (CLIENT type PERSONNE, NØ DE COMMANDE,
                      QUANTITE entier, DESIGNATION)

     fin classe ;
Classe B ;
     CO elle est certainement définie à un autre moment OC ;
     type NØ DE CØMMANDE entier < 5 000 ;
     type DESIGNATION struct (REF entier < 10 000, IDENT file
                              (max 30) car) ;
     PERSONNE struct (NOM file (max 20) car,
                      PRENOM file (max 15) car,
                      ADRESSE file (max 50) car)
     fin classe.
```

Dans l'exemple ci-dessus FCOM désigne un objet du type file de taille variable, composée d'objets du type déclaré par COM. Ce type est défini dans la classe A.
Dans la classe A, CØM est déclaré comme un type "structure" dont on se contente alors de préciser quels sont les champs.
CLIENT est un identificateur d'objet du même type que l'objet désigné par PERSONNE, tandis que NO DE COMMANDE désigne à la fois la deuxième partie de la structure COM et le type NO DE COMMANDE. De même pour DESIGNATION.
Dans la classe C, sont précisés les types utilisés : NO DE COMMANDE et DESIGNATION. Elle contient également la déclaration de l'objet PERSONNE (définition d'un type et déclaration d'un objet de ce type).
Dans le cas de la déclaration d'une file ou d'un ensemble d'objets d'un type déclaré, le mot type pourra être omis. Ainsi :

FCOM <u>file</u> COM ;

est équivalent à :

FCOM <u>file</u> <u>type</u> COM ;

REFERENCES DU CHAPITRE 4

BARTHELEMY C., PHILIPPE,
74 Problèmes d'analyse et de documentation dans le projet CIVA. A paraître. NANCY.

CHABRIER J. J. 73 Acquisition et édition des fichiers, analyse des données dans le projet CIVA. Doctorat 3ème cycle. Université NANCY I. Déc. 73

DUCLOY J. 73 Compilation dans le projet CIVA. Doctorat d'Ingénieur. Université NANCY I. Mars 73.

CHAPITRE 5

TABLES DE DECISION

5.1 INTRODUCTION.

5.2 TABLES DE DECISION CLASSIQUES.

5.21 Exemples.

5.22 Définitions.

1 Tables de décision.

2 Ambiguïté.

5.23 Valeur d'une table de décision.

1 Valeur d'une entrée de la table.

2 Valeur d'une règle.

3 Exécution d'une table.

5.24 Simplification courante. Règle E.

5.3 MODIFICATIONS POSSIBLES DES TABLES CLASSIQUES.

5.31 Changer l'ordre d'exécution des actions.

5.32 Eviter les ambiguïtés formelles.

5.33 Accepter les ambiguïtés réelles.

5.34 Intérêt des tables obtenues.

5.35 Introduire des tables de sélection.

5.36 Accepter des simplifications d'écriture.

5.37 Accepter des combinaisons des divers types de tables : tables généralisées.

5.4 TABLES DE DECISION DANS LE PROJET CIVA.

5.41 Contenu et exécution d'une table.

1 Notation.

2 Contenu et valeur d'une entrée de condition.

3 Contenu et valeur d'une entrée d'action.
4 Valeur et exécution d'une règle normale.
5 Valeur d'une table.

5.42 Exécution d'une table.
1 Table de sélection.
2 Table de décision.
3 Optimisation.

5.43 Présentation d'une table.
1 Dessin d'une table .
2 Simplification d'écriture.
3 Perforation.

5.44 Exemples de tables.

5.45 Tables de dimension 2 .
1 Exemples.
2 Présentation d'une table de dimension 2 .
3 Exécution d'une table de dimension 2.

5.46 Expressions en table de sélection.

CHAPITRE 5

TABLES DE DECISION

5.1 INTRODUCTION.

Les tables de décision sont un moyen d'expression clair et assez naturel des conditions régissant des actions à refaire. En informatique, elles constituaient à l'origine un moyen de communication commode entre les analystes et les programmeurs. Mais leur champ d'application est beaucoup plus vaste : c'est un outil de description des actions à entreprendre (automatisées ou non) et elles peuvent être également un outil de programmation (IBM, 65), (Picard, 69), (Kirk, 65). Dans CIVA, elles sont utilisées pour décrire l'organisation des chaînes de traitement en modules ; elles interviennent donc très tôt en analyse : elles expriment les conditions suffisantes pour que les modules doivent être appelés ; elles figurent également comme outil de programmation dans l'écriture détaillée des modules. La différence entre les deux types d'utilisation tient surtout au fait qu'en programmation on attache souvent beaucoup d'importance à l'ordre précis selon lequel les tests et les actions doivent être exécutés (les langages de programmation habituels imposent de prévoir cet ordre par la séquence d'écriture des instructions), alors qu'un analyste se contente d'indiquer ce qu'il veut voir réaliser (relations entre conditions et actions) sans se préoccuper des détails précis de réalisation : il travaille en "mode déclaratif" alors qu'un programmeur travaille en "mode impératif".

Les tables de décision conviennent bien à cette utilisation par un analyste puisque l'ordre d'évaluation des conditions n'apparaît pas.

Elles peuvent donc être considérées comme des "énoncés" (cf. Chap. 2) et plus particulièrement comme des "instructions désordonnées"

Pour qu'elles soient utilisables comme outil de programmation, il faut que l'on dispose d'un traducteur de tables de décision en un langage de programmation.

Des méthodes de traduction sont connues et des traducteurs existent (traducteur IBM vers Fortran (IBM, 68), DETAB 65 vers Cobol (ICL, 69) et (Callahan, 67).

IL faut aussi que ces traducteurs produisent un code objet efficace : nous utiliserons un traducteur qui minimise le nombre total de tests générés pour représenter une table et, pour le cas de chaînes de traitement répétitives, un traducteur minimisant le temps moyen d'évaluation d'une table.

Mais les tables de décision ne peuvent devenir un véritable outil de programmation que si elles sont incluses dans le langage de programmation lui-même. Elles constituent donc une instruction du langage.

Nous avons donc introduit dans le langage CIVA, une instruction particulière permettant de décrire les tables de décision "classiques". En fait, la forme de ces tables est le résultat d'un choix parmi les différentes formes qui auraient pu être retenues. Nous nous attacherons donc à examiner les points sur lesquels les choix auraient pu être différents. Après avoir étudié les tables de décision classiques (cf. 5.1), nous examinerons les diverses voies possibles pour étendre leur définition (cf. 5.2). Nous proposerons alors la forme des tables acceptées dans le langage (cf. 5.3). Les exemples rencontrés lors de l'étude des extensions possibles en seront des cas particuliers.

5.2 TABLES DE DECISION CLASSIQUES .

5.21 Exemple :

1) Un représentant de commerce doit visiter un client en ville et il se demande s'il doit utiliser sa voiture. Sa décision peut être fonction du temps qu'il fait, de la distance à parcourir et des difficultés de stationnement.
En notant V pour VRAI, F pour FAUX et X pour indiquer que l'action correspondante doit être entreprise, la table suivante peut représenter les conditions pour que les différentes décisions possibles soient prises.

TABLE I

Le stationnement est très difficile dans le quartier du client			V	F			
La distance est < à 500 m	V	V	F	F	F		
La distance est > à 1 000 m			F	F	F	V	V
Il pleut	V	F	F	F	V	F	V
Prendre un imperméable	X				X		X
Prendre la voiture				X	X	X	X
Aller à pied	X	X	X				

Elle exprime explicitement 7 règles correspondant aux colonnes de la table. La règle 1 par exemple signifie que si la distance est inférieure à 500 m et s'il pleut, il convient de prendre un imperméable et d'aller à pied. Remarquons que toutes les combinaisons possibles entre les valeurs des conditions ne sont pas représentées.

Si une case du quart supérieur droit ne contient ni V, ni F, cela signifie que la condition correspondante n'a pas d'importance pour cette règle.

2) Un problème d'assurance-vie

Devant un client susceptible de souscrire une assurance-vie, le démarcheur se propose de respecter les règles suivantes :

A - Si le client a moins de 30 ans

a) s'il est en excellente santé et n'a jamais été accidenté : proposer un contrat de type A

b) s'il est en mauvaise santé et a déjà été accidenté : refuser tout contrat.

c) Dans les autres cas : faire une enquête

B - Si le client a plus de 30 ans, se comporter de la même façon en proposant un contrat de type B au lieu du type A. On peut représenter ces règles de comportement par la table suivante :

TABLE 2

Age $\leqslant$ 30 ans	V	V	V	V	F	F	F	F
Bonne santé	V	V	F	F	V	V	F	F
Déjà accidenté	F	V	F	V	F	V	F	V
Contrat A	X							
Contrat B					X			
Enquête		X	X			X	X	
Refus				X				X

3) Reprenons l'exemple donné dans (IBM, 66) décrivant l'utilisation des tables de décision pour l'automatisation des études de fabrication, en particulier, dans une entreprise de production d'appareils de mesures de grandeurs électriques, pour représenter la détermination de l'enroulement de la bobine d'un galvanomètre. les études de fabrication sont représentées par un ensemble de tables du type de celle-ci :

TABLE 3

Dimension de l'échelle	4	4	4	8
Unité à mesurer	Milli-volt	Milli-volt	Milli-Ampère MA	Milli-ampère
Plage de mesure	10 - 75	76 - 200	10 - 200	75 - 250
Nombre de tours de l'enroulement	24	45	24	50
Nature du fil	Cu	Cu	Cu	Cu
Diamètre du fil	16	8	16	16
Numéro du schéma	A 1234	B 5678	F 9012	K 7821
Description par table	10	17	10	7

Cette table est un moyen suffisamment naturel de description des actions à entreprendre dans chacun des cas pour qu'elle se passe de commentaire.

5.22 Définitions .

5.221 Table de décision.

Une table de décision est une représentation synthétique d'un ensemble de règles, chacune d'elles exprimant les conditions suffisantes pour que des décisions soient prises.

Elle comprend quatre parties :

	SOUCHE	ENTREES
CONDITIONS	Souche de condition	Entrées de Condition
ACTIONS	Souche d'action	Entrées d'action

Une souche de conditions est en général une expression booléenne et une souche d'action une instruction. Une entrée de condition est en général une valeur booléenne, mais elle peut être également une valeur arithmétique (table 3 de 5.21) ou une expression booléenne ou arithmétique. Une entrée d'action consiste le plus souvent en un signe indiquant si l'action décrite par la souche doit être exécutée.

On dit qu'une table est à entrées limitées si les entrées de condition sont uniquement les valeurs booléenne "VRAI" et "FAUX", et qu'elle est à entrées étendues dans les autres cas. Le plus souvent, une table à entrées étendues est représentée en exprimant dans la souche une définition partielle des conditions ou des actions, la définition étant complétée par les entrées qui précisent les réalisations "utiles" parmi celles qui sont possibles. La table 3 de 5.21 est une table à entrées étendues de ce type, les tables 1 et 2 sont des tables à entrées limitées.

Une colonne de la partie des entrées exprime une règle. Lors d'une évaluation de la table, si les réalisations des conditions sont telles que les entrées de condition de la règle sont toutes vérifiées, alors il faut exécuter les actions repérées dans l'entrée d'action correspondante.

Une table est dite complète si toutes les réalisations possibles des conditions sont représentées : pour être complète, une table à entrées limitées comportant n conditions doit donc posséder 2^n règles ayant des ensembles d'entrées de conditions différents. La table 2 de 5.21 est une table complète.

5.222 Ambiguïté .

La table 1 de 5.21, ne semble pas complète bien qu'elle le soit. En effet, les règles 1, 2, 5, 6, 7 de cette table 1 comportent des conditions indifférentes que nous noterons maintenant dans la partie entrée des conditions. Chacune d'elles représente donc plusieurs règles : la règle 1 représente 4 règles, ayant toutes les mêmes entrées d'action et ayant respectivement pour entrées de condition : FVFV, VVFV, FVVV, VVVV. La table 1 représente donc une table qui a 20 règles, c'est-à-dire qu'elle contient des règles ayant les mêmes entrées de condition. On dit que ce sont des règles ambiguës.

L'ambiguïté peut n'être qu'apparente si les entrées d'action de deux règles ambiguës sont les mêmes : la table contient deux fois la même règle, c'est une redondance non gênante. Dans les autres cas, la table est dite ambiguë et, bien souvent, les traducteurs de tables de décisions n'acceptent pas les tables ambiguës.

5.23 Valeur d'une table de décision .

5.231 Valeur d'une entrée de la table.

a) Entrée d'action.

Elle n'a que deux valeurs possibles : un X dans la ligne i, signifiant qu'il faut exécuter la ième action, un "blanc" signifiant qu'il ne faut pas l'exécuter.

b) Entrée de condition.

Dans le cas d'une table à entrées étendues, la valeur d'une entrée de condition est celle d'une expression booléenne obtenue en complétant l'expression partielle de la souche correspondante avec l'expression partielle de l'entrée (il convient souvent d'insérer un signe =). Ainsi dans la table 3 de 5.21, la valeur de la première entrée de condition est :

dimension de l'échelle = 4

Pour une table à entrées limitées, la valeur d'une case est "VRAI" si l'entrée est indifférente, sinon c'est celle de l'expression booléenne obtenue en écrivant :

souche de la condition = entrée de la condition

5.232 Valeur et exécution d'une règle.

La valeur d'une règle est :

E_1 et E_2 et..... et E_n

où les E_i désignent les valeurs des n entrées des conditions de la règle.

Exécuter cette règle c'est :

"exécuter" A_1 et A_2 et A_3..... et A_m

où les A_j désignent les valeurs des m entrées d'action de la règle , la conjonction "et" impliquant souvent que les actions seront entreprises dans l'ordre où elles sont écrites dans la table.

5.233 Exécution d'une table .

Pour la plupart des traducteurs, l'exécution d'une table peut être définie ainsi :

- s'il existe une, et une seule, règle dont la valeur est "VRAI", l'exécution de la table consiste à exécuter cette règle ;

- s'il n'existe pas de telle règle, la table est inopérante (instruction vide) ;

- s'il n'existe plus d'une telle règle, l'exécution de la table n'est pas définie.

REMARQUE :

Ce dernier cas est détecté en présence des réalisations des conditions quand on interprète la table. Il peut être détecté dès la traduction de la table quand on génére une traduction.

5.24 Simplification courante. Règle E.

Une table incomplète peut être complétée implicitement par le traducteur de la table, qui prévoit d'entreprendre une action par défaut : signaler une erreur, si on n'accepte pas les tables incomplètes, ou passer en séquence par exemple.

Elle peut également être complétée explicitement par une règle représentant les autres cas, habituellement notée E (pour Else Rule), que l'on notera également sinon ou S.

Ainsi, la table 2 pourrait être décrite :

TABLE 4

					E
Age $\leqslant$ 30 ans	V	V	F	F	
Bonne santé	V	F	V	F	
Déjà accidenté	F	V	F	V	
Contrat A	X				
Contrat B			X		
Enquête					X
Refus		X		X	

TABLE 5

				E
Age ≤ 30 ans	V	F	--	
Bonne santé	V	V	F	
Déjà accidenté	F	F	V	
Contrat A	X			
Contrat B		X		
Enquête				X
Refus			X	

5. 3 MODIFICATIONS POSSIBLES DES TABLES CLASSIQUES.

5.31 Changer l'ordre d'exécution des actions.

On a vu (cf. 5.232) que l'exécution d'une table est couramment définie comme l'exécution d'une règle, c'est-à-dire :

"exécuter" A_1 et A_2 et A..... A_m dans l'ordre dans lequel les actions figurent dans la table.

On peut vouloir imposer pour chaque règle un ordre différent de l'ordre d'écriture des actions dans les souches ou au contraire ne pas imposer le choix au traducteur.

a) L'ordre dans lequel les actions sont décrites dans les souches reflète l'ordre dans lequel leur nécessité est apparue à l'analyste, et pas nécessairement un ordre d'exécution. Lorsque l'ordre d'exécution est significatif, cette définition nous oblige à modifier la table ou alors à répéter plusieurs fois la même action dans la souche. Il est donc préférable de permettre à

l'utilisateur d'imposer explicitement un ordre d'exécution des actions, par exemple en leur donnant un numéro d'ordre dans chaque règle : les actions seront exécutées dans l'ordre croissant des numéros.

b) Au contraire, il peut être intéressant de laisser au traducteur la liberté de déterminer dans quel ordre il fera exécuter les actions correspondantes, et même la latitude d'effectuer des actions simultanément. Dans ce cas, on dit que les actions sont exécutées collatéralement (Buffet , 72).

Deux actions pouvant être exécutées collatéralement seront repérées dans une règle par des numéros égaux.

5.32 Eviter les ambiguïtés formelles.

Reprenons l'exemple de la table 1 (cf. 5.21), dont nous avons affirmé (cf. 5.22) qu'elle est ambiguë en constant qu'elle contient plus de 2^4 règles.

Envisageons les règles 1 et 7 de cette table :

Règle 1 -, V, -, V

Règle 7 -, -, V, V

Elles représentent respectivement les entrées de condition des règles :

R. 11 : V, V, V, V, ; R. 12 : F, V, V, V, ; R. 13 : V, V, F, V, ;
R. 14 : F, V, F, V, ;
R. 71 : V, V, V, V, ; R. 72 : F, V, V, V, ; R. 73 : V, F, V, V, ;
R. 74 : F, F, V, V,.

Les entrées d'action de R 1 et celle de R 7 étant différentes, les règles R 11 et R 71 sont réellement ambiguës - on dira aussi contradictoires - ainsi que les règles R 12 et R 72.

Or cette contradiction est uniquement formelle, car lorsqu'on se réfère à la sémantique des conditions 1 et 2 de la souche :

Distance $<$ 500 m et Distance $>$ 1 000m,

on constate que les règles R 11, R 71, R 12 ne sont jamais réalisables. Cette ambiguïté ne pose pas de problème si on interprète la table, elle risque d'en poser si on génère une traduction.

Il semble alors intéressant de signaler au traducteur que certaines entrées de condition ne sont pas réalisables, ce qui

évitera de faire des tests inutiles et éventuellement, comme c'est le cas ici, permettra d'éviter des contradictions.

On peut donc proposer la modification suivante :

Si dans une règle, on sait par les relations sémantiques entre les conditions, que la valeur d'une entrée résulte de celles des autres entrées de la même règle, on pourra le signaler, et on aura intérêt à le faire pour obtenir une meilleure traduction.

Ainsi dans la règle 1, on aurait intérêt à signaler que la condition 3 (distance $>$1 000 m) est nécessairement fausse. Nous le ferons en notant un signe + dans l'entrée correspondante. Bien que cela soit inutile pour l'évaluation de la règle, le signe + pourra être suivi du caractère V ou F selon la valeur de la condition imposée, ce qui facilitera la lecture de la table et pourra permettre des vérifications.

5.33 Accepter les ambiguïtés réelles .

Dans le cas où deux règles possèdent le même ensemble d'entrées de condition mais diffèrent par leurs entrées d'action, il y a ambiguïté réelle. Lorsque cette ambiguïté n'est pas seulement formelle (cf. 5.32), deux positions peuvent être adoptées pour définir la valeur de la table.

- considérer que, les actions étant différentes, il risque d'y avoir une contradiction entre la sémantique de chacune d'elles, et, partant, déclarer la table incorrecte ;

- considérer que les actions doivent toutes être entreprises et que l'utilisateur doit prendre soin d'éviter de commander ainsi des actions sémantiquement contradictoires.

La position habituelle est la première. Nous proposerons d'accepter également la seconde en prenant la précaution , à la traduction d'une table, de signaler à l'utilisateur la présence d'une ambiguïté réelle et en indiquant les règles qui en sont l'origine.

Nous constaterons, en 5.34, l'intérêt d'une telle position. L'exécution d'une table ambiguë peut être définie comme suit :

Exécuter collatéralement (cf. 5.31) l'ensemble des règles dont la valeur est "VRAI" ;

il est en effet difficile de déterminer un ordre d'exécution par

la table seule ; on pourrait prendre l'ordre des règles, mais cela est difficilement compatible avec l'efficacité du traducteur.

5.34 Intérêt des tables obtenues.

Dans l'article (Markia A, 71) , il est proposé une table de décision pour décrire les règles d'attribution de certaines primes, règles qui sont définies ci-dessous. L'importance de la table obtenue permet à l'auteur de mettre en évidence la lourdeur des tables de décision. L'utilisation des entrées indifférentes et des entrées imposées permet de simplifier considérablement la table proposée.

Règles d'attribution des primes :

- primes d'ancienneté (PA) : si (Age $<$ 30 et Ancienneté $\geqslant$ 5)
  ou (Age $\geqslant$ 30 et Ancienneté $\geqslant$ 3
- primes spéciales (PS) : si non cadre et Anc $\geqslant$ 3,
- intéressement des employés (IE) : si non Cadre et Anc $\geqslant$ 5,
- intéressement des cadres (IC) : si Cadre et Anc $\geqslant$ 5,
- intéressement spécial (IS) : si Anc $\geqslant$ 5,
- prime jeune ménage (PJM) : si marié et AGE $<$ 30

L'utilisation des tables de décision sous leur forme classique conduit à la table 6, forme qui évidemment est lourde et difficilement utilisable.

TABLE 6

AGE < 30	F	F	F	F	F	F	F	F	V	V	V	V	V	V	V	V	V
ANCienneté ≥ 3	V	V	V	V	V	V	V	V	F	F	V	V	V	V	V	V	V
ANCienneté ≥ 5	F	F	F	F	V	V	V	V	F	F	F	F	F	V	V	V	V
Cadre	F	F	V	V	F	F	V	V	F	V	F	F	V	F	F	V	V
Marié	F	V	F	V	F	V	F	V	F	F	F	V	V	F	V	F	V
PA (Prime d'ancienneté)	X	X	X	X	X	X	X	X	-	-	-	-	-	X	X	X	X
PS (Prime Spéciale)	X	X	-	-	X	X	-	-	-	-	-	-	-	X	X	-	-
IE (Intéressement employés)	-	-	-	-	X	X	-	-	-	-	-	-	-	X	X	-	-
IC (Intéressement cadres)	-	-	-	-	-	-	X	X	-	-	-	-	-	-	-	X	X
IS (Intéressement spécial)	-	-	-	-	X	X	X	X	-	-	X	X	-	X	X	X	X
P. J. M. (Jeune Ménage)	-	-	-	-	-	-	-	-	X	X	-	X	X	-	X	-	X

Si on admet les entrées imposées de 5.32 et les entrées indifférentes sans admettre de table ambigüe, on obtient une table légèrement plus simple. (Table 7')

TABLE 7

	AGE < 30	ANC ≥ 3	ANC ≥ 5	Cadre	Marié	P. A.	P. S.	I. E.	I. S.	I. C.	P. J. M.
	F	V	F	F	-	×	×	-	-	-	-
	F	V	F	V	-	×	-	-	-	-	-
	F	+ V	V	F	-	×	×	×	×	-	-
	F	+ V	V	V	-	×	-	-	×	×	-
	V	F	F	-	F	-	-	-	-	-	×
	V	V	F	F	F	-	-	-	×	-	-
	V	V	F	F	V	-	-	-	×	-	×
	V	V	F	V	V	-	-	-	-	-	×
	V	+ V	V	F	F	×	×	×	×	-	-
	V	+ V	V	F	V	×	×	×	×	-	×
	V	+ V	V	V	F	×	×	-	×	×	-
	V	+ V	V	V	V	×	×	-	×	×	×

Si nous admettons de plus une table ambiguë telle que définie en 5.33, nous obtenons, cette fois, une table assez simple pour penser pouvoir reviser le jugement proposé en (Markia A., 71).

TABLE 8

AGE < 30	-	F	-	-	-	V
ANC ⩾ 3	+ V	V	V	+ V	+ V	-
ANC ⩾ 5	V	-	-	V	V	-
Cadre	-	-	F	F	V	-
Marié	-	-	-	-	-	V
P. A.	X	X	-	-	-	-
P. S.	-	-	X	-	-	-
I. E.	-	-	-	X	-	-
I. S.	X	-	-	-	-	-
I. C.	-	-	-	-	X	-
P. J. M.	-	-	-	-	-	X

Remarquons de plus que cette table s'obtient beaucoup plus directement à partir de l'énoncé que la table 6.

5.35 Introduire des tables de sélection .

Nous avons vu, (cf. 5.33) que devant deux règles ambiguës, deux positions peuvent être adoptées : signaler une erreur ou exécuter collatéralement les deux règles. En fait, une autre position est également possible : n'exécuter qu'une seule règle. Il convient alors de préciser laquelle : nous dirons que la règle choisie sera la première rencontrée dans l'ordre d'écriture de la table.
Une table définie ainsi sera appelée table de sélection.

Tables de sélection et tables de décision sont des objets identiques lorsqu'elles ne contiennent pas de règles ambiguës : dans ce cas, il n'y a jamais qu'une seule règle dont la valeur est "VRAI" pour une réalisation des conditions, et l'ordre d'exécution des règles n'a pas d'importance.

Nous proposons donc d'accepter deux types de tables : tables de décision, acceptant les ambiguïtés et considérant que deux règles ambiguës doivent être exécutées collatéralement, et tables de sélection, acceptant les ambiguïtés, mais n'exécutant que l'ensemble d'actions de la première règle rencontrée.

5.36 Accepter des simplifications d'écriture .

a) On pourrait utiliser une représentation plus naturelle et plus commode dans le cas particulier où chaque règle a une et une seule entrée de condition valant "VRAI", les autres entrées étant indifférentes, sauf peut être pour une seule règle ayant toutes ses entrées de condition à "FAUX". Ainsi la table 9 pourrait s'écrire plus facilement par la table 10.

TABLE 9

A + B >0 et C >0	V	-	-	-	F
A + B >0 et D = 0	-	V	-	-	F
A + B >0 et A-B = 2	-	-	V	-	F
A + B >0 et A-B >1	-	-	-	V	F
A 1	1	-	-	-	-
A 2	-	1	-	-	-
A 3	-	-	1	-	-
A 4	-	-	-	1	-
A 5	-	-	-	-	1

TABLE 10

	A + B > 0 et C > 0	A + B > 0 et D = 0	A + B > 0 et A-B=2	A + B > 0 et A-B > 1	Sinon
A1	1	—	—	—	—
A2	—	1	—	—	—
A3	—	—	1	—	—
A4	—	—	—	1	—
A5	—	—	—	—	1

b) Si les expressions booléennes figurant alors dans les entrées des conditions ont une partie commune, on peut accepter de ne l'écrire qu'une seule fois.

Aussi, la partie "conditions" de la table 10 pourrait s'écrire :

A + B > 0	C > 0	D = 0	A-B = 2	A-B > 1	SINON

c) Si les entrées sont toutes à "blanc" sauf une entrée par action, alors la partie "actions" de la table peut subir une simplification analogue. La table 10 s'écrirait ainsi : Table 11

A + B > 0	C > 0	D = 0	A-B = 2	A-B > 1	SINON
	A1	A2	A3	A4	A5

On retrouve ainsi une forme des tables de décision à entrées étendues, acceptant les ambiguïtés, et que nous considérons comme une écriture simplifiée d'une table à entrées limitées.

Cette forme d'écriture pourra être exploitée pour la traduction d'une table de sélection simplifiée : si la table 11 est une table de sélection, elle peut se traduire en évaluant successivement chacune des conditions et en s'arrêtant dès qu'une condition est vraie.

d) Tables réduites à une seule condition

Une table réduite à une seule condition s'écrira de façon

C	SINON
A1	A2

OU

C
A1

classique sous la forme d'une instruction conditionnelle :
si C alors A1 sinon A2 fsi ; ou respectivement
si C alors A1 fsi ;

5.37 Accepter des combinaisons des divers types de tables. Tables généralisées.

Ainsi, la table 12 est une écriture plus commode de la table 13.

TABLE 12

A > 2	V	F				V	F	S
B = 0	F	V				F	V	
A + B > 0	V	C = 0	C < 0	C > 10	0 < C ≤ 10	F	-	
A1	-	-	1	1	1	1	-	1
A2	-	1	-	2	2	-	1	2
A3	1	2	2	-	3	-	-	-

TABLE 13

A > 2	V	F	F	F	F	V	F	S
B = 0	F	V	V	V	V	F	V	
A+B > 0	V	V	V	V	V	-	-	
C = 0	-	V	+ F	+ F	F	-	-	
C < 0	-	+ F	V	+ F	F	-	-	
C > 0	-	+ F	+ F	V	F	-	-	
A1	-	-	1	1	1	1	-	1
A2	-	1	-	2	2	-	1	2
A3	1	2	2	-	3	-	1	-

Remarquons toutefois que la décomposition de cette table en deux tables, l'une correspondant au cas où A 2 et B = 0, l'autre aux autres cas permet également une écriture très commode ; mais elle oblige à écrire deux fois les actions, comme on peut le contater sur la table 14, où P est un appel d'une procédure contenant la table 15.

TABLE 14

A > 2	V	F	V	F	Sinon
B = 0	F	V	F	V	
A + B 0	V	V	F	–	
A1	–	–	1	–	1
A2	–	–	–	1	2
A3	1	–	–	–	–
P	–	1	–	–	–

TABLE 15

	C = 0	C < 0	C > 10	Sinon
A1	-	1	1	1
A2	1	-	2	2
A3	2	2	-	3

Après avoir examiné les diverses voies possibles dans lesquelles s'engager pour étendre la définition des tables de décision, nous allons proposer une définition générale d'une table de décision, dont tous les cas précedemment rencontrés ne seront que des cas particuliers.

5.4 TABLES DE DECISION DANS LE PROJET CIVA .

5.41 Contenu et valeur d'une table.

5.411 Notations.

Une table de décision utilisable en CIVA se présente sous la forme :

C_1	EC_{11}	EC_{12}			EC_{1m}
!	EC_{21}				
!			---------	----------	
!					
!					
C_n	EC_{n1}				EC_{nm}
A_1	EA_{11}				EA_{1m}
!					
!			---------	----------	
!					
!					
A_p	EA_{p1}				EA_{pm}

Une table peut avoir deux modes d'évaluation : table de décision ou table de sélection.

Nous noterons :

C_i la souche de la ième condition, $i \in [1, n]$;

A_i la souche de la ième action, $i \in [1, p]$;

R_j la jième règle, $j \in [1, m]$;

EC_{ij} l'entrée de la ième condition dans la jème règle, $i \in [1, n]$, $j \in [1, m]$

EA_{ij} l'entrée de la ième action dans la règle R_j, $i \in [1, n]$ $j \in [1, m]$

Les C_i sont des expressions booléennes. Les A_i sont des instructions CIVA, à l'exception des tables de décision non réduites à une instruction conditionnelle. Ce peut être en particulier un appel de module ou de procédure, ou une instruction conditionnelle, ou une suite d'instructions de ce type. Rappelons l'existence de deux instructions de bouclage particulières qui ont un grand intérêt pour l'écriture des tables REINST et RETABLE.

La rencontre de RETABLE dans la partie des actions demande un bouclage en tête de la table elle-même. Cela suppose bien entendu que dans chaque règle demandant cette action, une autre action doit être demandée auparavant, permettant de modifier les réalisations des conditions.

La rencontre de REINST demande de recommencer le traitement en tête de la plus petite unité contenant la table : un module ou une procédure.

5.412 Contenu et valeur d'une entrée de condition.

Une entrée de condition peut contenir une expression booléenne (en particulier V ou F), une marque de condition indifférente — ou une marque de condition imposée + V ou + F.

La valeur v (EC_{ij}) d'une entrée de condition EC_{ij} est du type booléen et elle est définie comme suit :

- si EC_{ij} contient une expression booléenne EB autre que F : $C_i \wedge EB$;
- si elle contient F : $\neg C_i$;
- si elle contient + F, + V ou — : VRAI.

La dernière entrée de la première condition EC_{1m} peut également contenir S ou Sinon : la règle m est alors une règle "sinon" (règle E de 5.24). Les EC_{im} pour i = 1 doivent contenir une marque de condition indifférente ou être vides. La valeur des entrées de condition de la règle sinon ne sont définies, la valeur de la règle sinon sera définie en 5.415.

5.413 Contenu et valeur d'une entrée d'action.

Une entrée d'action peut contenir :

- un numéro d'ordre n ;
- une instruction CIVA, autre qu'une table de décision ;
- les deux ;
- ou encore être vide.

Dans l'entrée d'action, n et I seront écrits dans un ordre quelconque, et séparés par un point virgule.

Si l'entrée EA_{ij} est vide, sa valeur est l'instruction vide (ne rien faire).

Si l'entrée EA_{ij} contient un numéro n, sa valeur est nA_i, où n numérote l'instruction A_i.

Si elle contient une instruction I, sa valeur est A_i ; I.

Si elle contient à la fois un numéro n et une instruction I, sa valeur est n (A_i ; I) s'ils sont écrits n ; I et elle est n (I ; A_i) s'ils sont écrits I ; n.

5.414 Valeur et exécution d'une règle normale.

Une règle "normale" est une règle autre que la règle "sinon".

La valeur v (Rj) de la règle normale Rj est la valeur de l'expression booléenne :

v (EC_{1j}) et v (EC_{2j}) et..... et v (EC_{nj})

Une règle R_j ne peut être exécutée, que si cette règle a pour valeur VRAI.

Pour exécuter une règle, on évalue chacune de ses entrées d'actions, ce qui conduit à une suite de couples numérotés par n_i d'instructions A_i et I_i. Chacun des constituants peut être absent. Il peut donc y avoir en particulier des instructions non numérotées : il y a une croix à la place d'un numéro. Si la suite est vide, l'exécution est terminée ; si elle n'est pas vide, on exécute

- la suite, dans l'ordre des numéros n_i des couples d'instructions A_i et I_i ; dans un même couple A_i et I_i sont exécutées dans l'ordre défini ci-dessus pour la valeur d'une entrée d'action ;
- les instructions non numérotées ; elles sont exécutées dans un ordre arbitraire.

5.415 Valeur d'une table .

Nous appellerons table positive associée à une table, la table obtenue en supprimant de celle-ci la règle "sinon", si elle existe.

Soit m' le nombre de règles de la table positive associée à une table à m règles : m' = m ou m-1.

La valeur de la table positive associée à une table de décision ou de sélection est la valeur de l'expression :

$E = v(R_1) \vee v(R_2) \vee \ldots\ldots\ldots\ldots \vee v(R_{m'})$.

La valeur de la règle "sinon" est le complément logique de E.

La valeur d'une table est celle de E, si m = m' (pas de règle "sinon") et toujours vraie, si m = m' + 1.

5.42 Exécution d'une table .

5.421 Table de sélection.

L'exécution d'une table de sélection comportant m règles est décrite par la procédure suivante, en supposant les règles rangées dans une file R composée de deux champs, leur valeur v et leur ensemble d'actions EA. Les instructions "pour chaque" et "sortir" sont précisées au chapitre 7. (resp. 6).

Procédure sélection (R) ;

Pour chaque I de 1 à m' faire

si v de R (i) alors exécuter E A de R (i) ;

sortir fsi fpc

si m' = m-1 alors exécuter E A de R (m) fin proc

5.422 Table de décision.

En prenant les mêmes conventions que ci-dessus, la procédure décision (R) est une description de l'exécution d'une table de décision. Deux descriptions peuvent différer par l'ordre d'évaluation des règles.

Procédure décision (R) ; I booléen ; I = vrai ;

pour chaque X de R (1 : m') tel que v de X faire

I = faux ; exécuter EA de X fpc ;
si I et m'=m-1 alors exécutr EA de R (m) fsi fproc.

5.423 Optimisation.

Ce qui précède définit l'exécution d'une table, mais la réalisation pratique peut être fort différente. En effet, chaque condition est évaluée chaque fois qu'on évalue une règle. Dans le cas d'une table à entrées limitées, on peut penser à n'évaluer les conditions qu'une seule fois et construire un vecteur booléen que l'on compare ensuite globalement aux entrées de condition de chacune des règles (techniques de "masques" (ICL, 69). Dans le cas général, ceci n'est pas applicable. Il convient donc de proposer des méthodes d'évaluation de ces tables un peu plus efficaces que celles des paragraphes précédents. Dans la partie "réalisation" et dans (Aubry B. 73) nous proposons une méthode de traduction permettant de minimiser le nombre de tests évalués dans chaque éventualité et une méthode permettant de minimiser le temps moyen d'évaluation d'une table, connaissant les probabilités des fréquences de réalisation des différentes conditions.

5.43 Présentation d'une table.

5.431 Dessin d'une table.

Une table est une instruction et peut se trouver, comme telle, dans le texte d'un module ou d'une procédure.

Les entrées des conditions et des actions ont été décrites en 5.41.

Une souche de condition contient une expression booléenne ou un identificateur de condition. Celui-ci désigne une expression booléenne à laquelle il a été déclaré équivalent par une déclaration :
identificateur de condition condition expression booléenne (cf. Chap. 4).

Une souche d'action contient une instruction CIVA autre qu'une table.

Une table de décision commence par le symbole décision, une

table de sélection par le symbole sélection.

Les entrées se terminent par un trait simple. La partie actions commence par le mot action. Elle peut être séparée de la partie conditions par un trait horizontal double. Les règles sont séparées par des traits verticaux. La table se termine par fin table.

Si elle existe, la règle E sera représentée par le symbole sinon dans une de ses entrées.

La marque de condition indifférente — peut être remplacée par un blanc, tout comme la marque qui indique, dans une entrée d'action, que l'action correspondante ne doit pas être exécutée. Le symbole fin table peut être écrit ft.

Exemple.

TABLE 16 (cf. TABLE 12)

Décision	A + B > 0	V	C = 0	C < 0	C > 10	0 < C ≤ 10	F		Sinon
	B = 0	F	V	V	V	V	F	V	
	A = 2	V	F	F	F	F	V	F	
Actions	A 1			1	1	1	1		1
	A 2		1		2	2		1	2
	A 3	1	2	2		3			
Fin table									

5.432 Simplifications d'écriture.

a) Si plusieurs entrées de condition voisines dans une même ligne sont identiques, on peut enlever les traits verticaux qui les séparent et ne les écrire qu'une fois.

Ainsi la table 16 pourrait être dessinée selon le modèle de la table 17.

TABLE 17

Décision						
A+B 0	V			F		Sinon
B = 0	V			F		
A = 2	F	F	V	V	F	
C = 0	V	F	F	V	F	
Actions						
A 1	-	1	1 ; A=2			
A 2	-	2	2	2 ; B = 0	1	
A 3	1	3		1		
Fin table						

b) Si les expressions booléennes de toutes les entrées de condition d'une ligne de la table commencent par le même début d'expression (terminé par un opérateur de relation ou un opérateur logique), alors celui-ci pourra n'être écrit qu'une seule fois dans la souche. Si cette souche contient déjà une expression booléenne, celle-ci sera suivie d'un <u>et</u>.

A + B > 0 <u>et</u> C = | 2 | 3 | 4 |

La 2ème entrée a pour valeur A + B 0 <u>et</u> C = 3.

Ainsi la table

<u>Décision</u>	A <	0	5	7	13
<u>Action</u>	I = I+1	1	1	1	1

<u>ft</u>

permet de calculer le rang de l'intervalle dans lequel se trouve A. (Si I a pour valeur initiale 0).

c) Si toutes les entrées d'une ligne d'action contiennent une instruction d'affectation ayant la même partie gauche, celle-ci pour être écrite dans la souche une seule fois. Les entrées d'action de cette ligne contiennent alors des expressions. Si la souche d'action correspondante contient plusieurs instructions, le début de l'instruction d'affectation est nécessairement à la fin de cette suite d'instructions. Ainsi la table

<u>Sélection</u>	A <	0	5	7	13	Sinon
<u>Action</u>	I =	1	2	3	4	

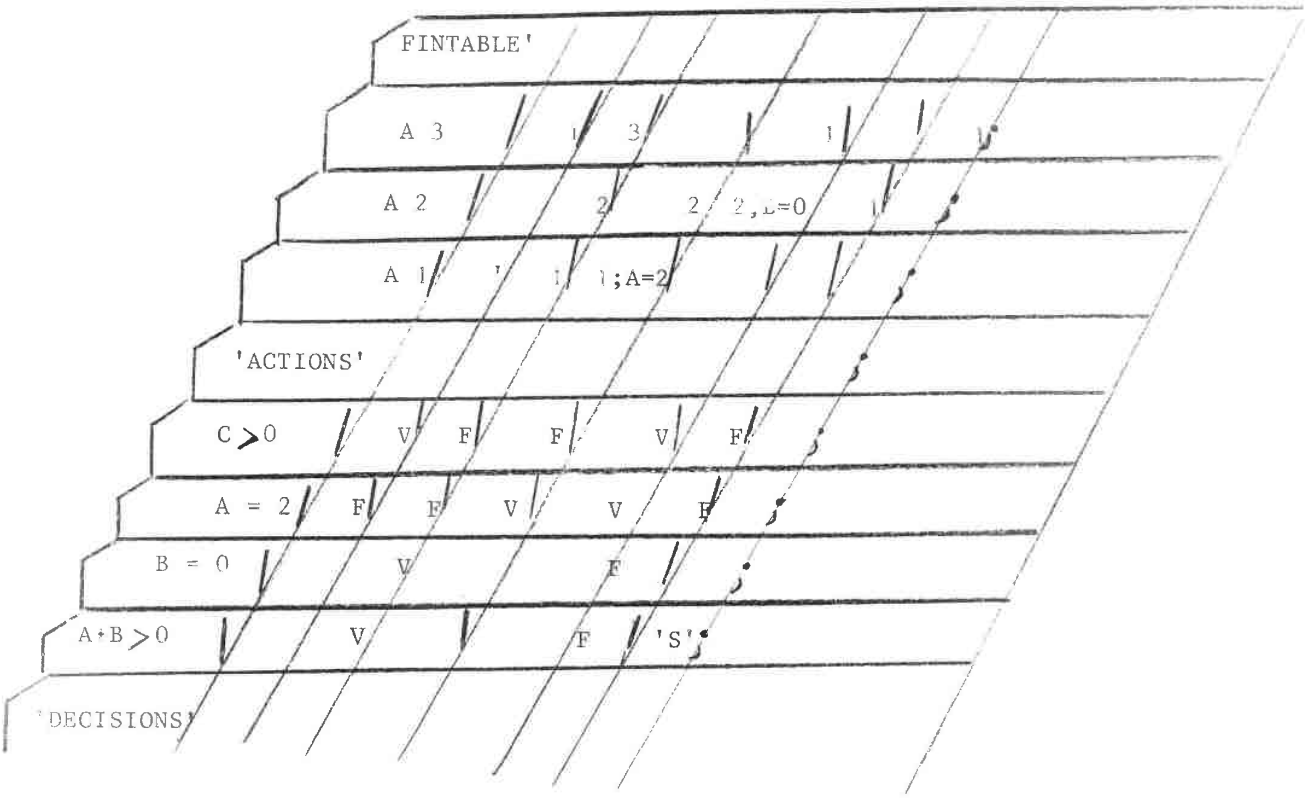
a le même effet que la table de l'alinéa b) ; son exécution sera certainement plus rapide.
Si une entrée d'action d'une ligne simplifiée de cette façon contient un numéro d'ordre, il suivra l'expression et en sera séparé par un point virgule.

5.433 Perforation.

Une table de décision ou une table de sélection sera perforée, ligne par ligne, en respectant la disposition adoptée pour son dessin. Une ligne pourra occuper plusieurs cartes.
Les traits horizontaux n'apparaîtront pas : une nouvelle ligne correspond à une nouvelle carte. Les traits verticaux seront représentés par une barre verticale : ces barres occuperont les mêmes positions sur toutes les cartes. La dernière barre verticale sera représentée pour chaque ligne par un point-virgule.

Exemple :

La table 17 pourrait être perforée comme suit



Un certain nombre de simplifications seront admises :

- Actions et fintable pourraient être perforés à la fin de la carte précédente ;
  le point virgule avant fintable pourra être omis ;
- si une ligne se termine par des entrées indifférentes, le point virgule de fin de ligne pourra être placé immédiatement après la dernière entrée non indifférente.

5.44 Exemples de tables.

1) Tables simples.

Les formes suivantes sont donc des tables utilisables en CIVA.

i) TABLE 18

Décision	C 1	V	V	V	Sinon
	C 2	F	V	V	
	C 3	V	F	V	
Actions		A 1	A 2	A 3	A 4

ft

ii) TABLE 19

Sélection	C 1	C 2	C 3	C 4	Sinon
Actions	A 1	A 2	A 3	A 4	A 5

ft

iii) En utilisant la remarque b) de 5.432, on peut écrire une table de la forme :

TABLE 19

Sélection	I =	1	2	3	4	5	Sinon
Action		A 1	A 2	A 3	A 4	A 5	A 5

ft

qui est à rapprocher d'un aiguillage des langages classiques.

2) La table 20 permet de faire l'itération $x_{n+1} = f(x_n)$ jusqu'à ce que l'écart entre deux valeurs consécutives soit inférieur à une quantité fixée ou que l'on ait fait N itérations.

TABLE 20

Décision		
	I ≤ N	V
	DIF > EPSILON	V
Action	I=I+1	1
	DIF=X	2
	X=F (X)	3
	DIF = |X-DIF|	4
	Retable	

Fin table

Table 21

Décision	C1	F	V	V	Sinon
	C2	F	F		
	C3	-	F	V	
Action	A1	1	1		1
	A2	1			
	A3	2		1	1

Fintable

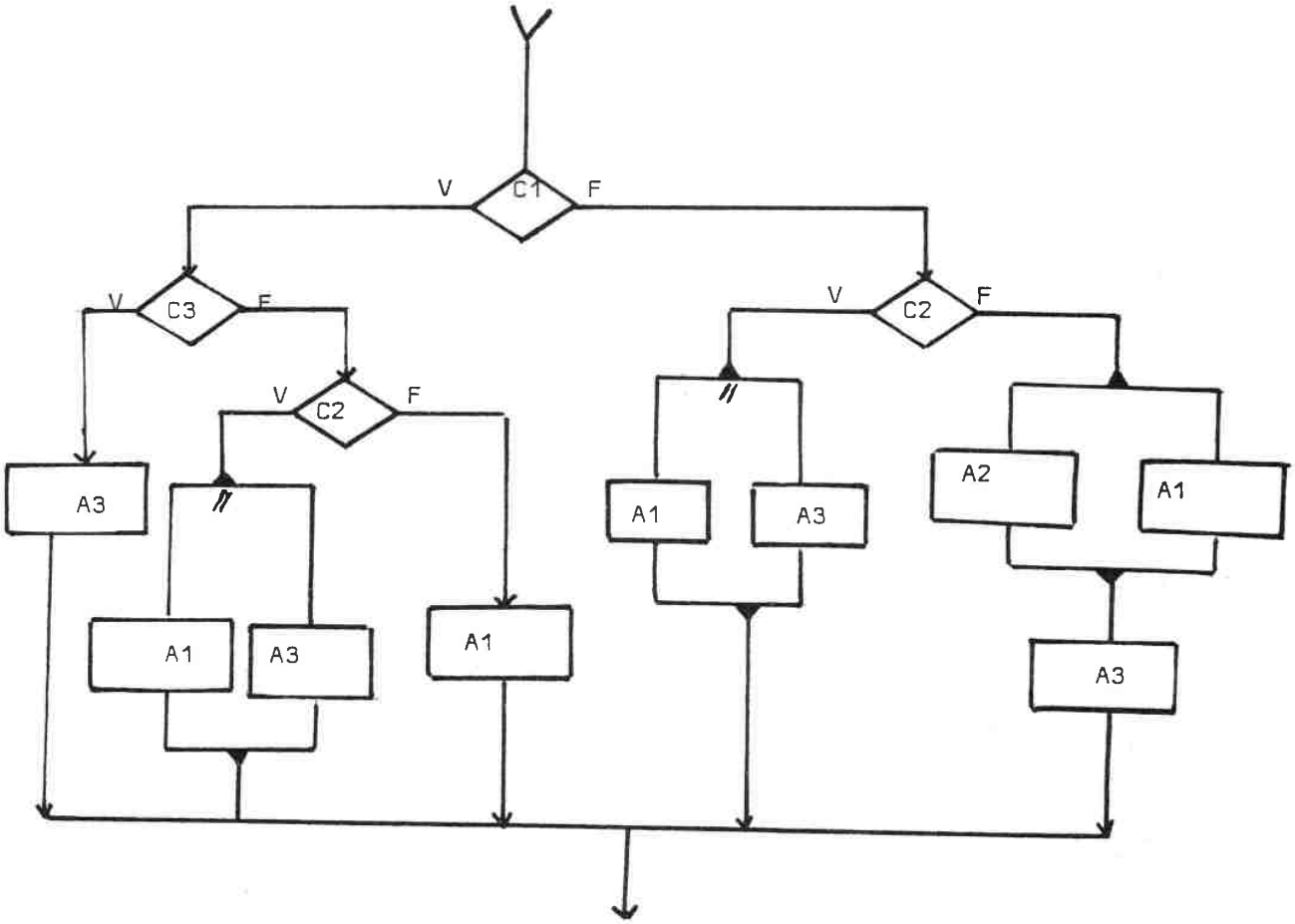
Cette table pourrait être perforée :

Décision	C1	F	V	V	sinon ;
	C2	F	F ;		
	C3	-	F	V ;	
Actions	A1	1	1	-	1 ;
	A2	1 ;			
	A3	2	-	1	1 fintable ,

C1, C2, C3 sont des variables booléennes ou des identificateurs de condition ;

A1, A2, A3 sont des noms de procédure ou de module (cf. 54312).

Elle pourrait correspondre à l'organigramme suivant :



Le symbole // indique que les branches qui suivent sont évaluées collatéralement et donc qu'elles peuvent l'être en parallèle. Le symbole ▼ indique que les actions situées sur les branches supérieures doivent toutes être terminées pour passer plus loin.

4) Envisageons les deux tables :

Table 22

Déc.	C1	-	F	V	Sinon
	C2	V	-	V	-
	C3	F	V	-	-
Act.	A1	1	-	-	2
	A2	-	1	-	-
	A3	-	-	1	1

ft

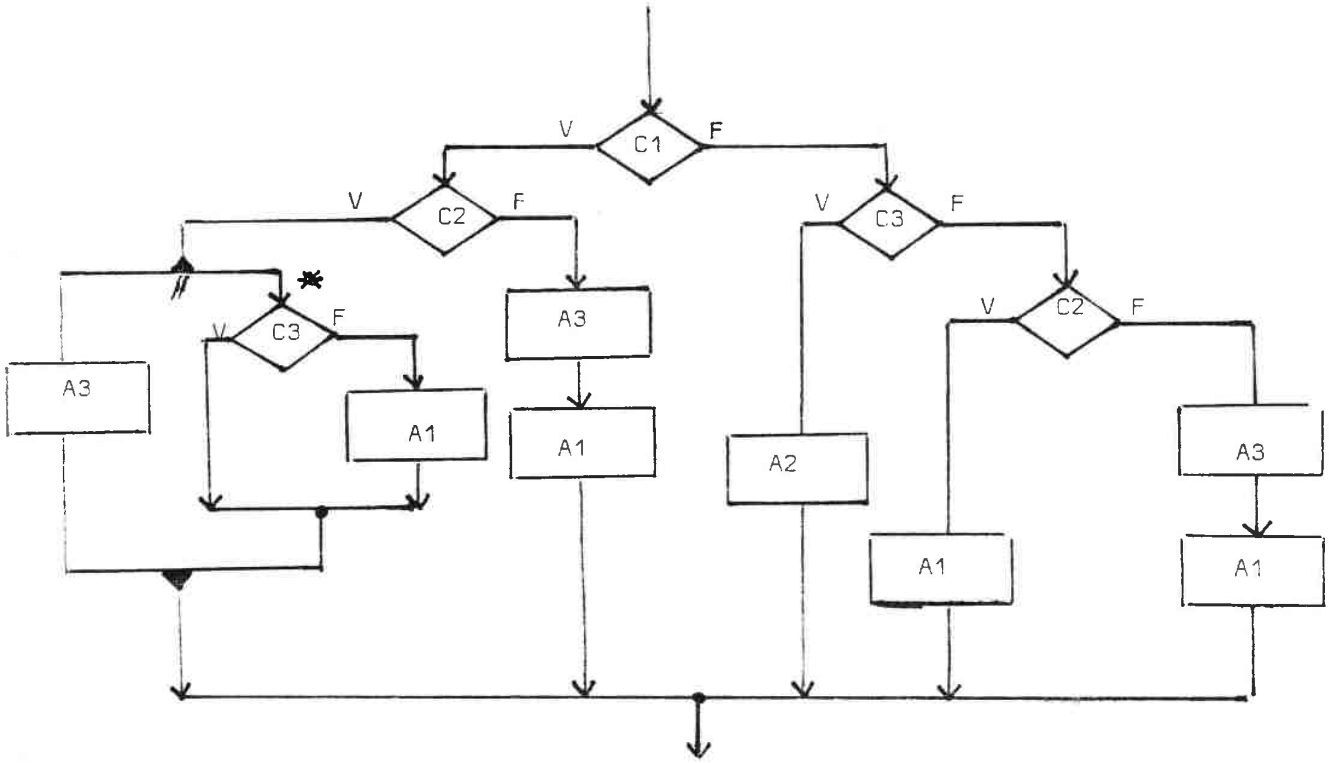
Table 23

Déc.	C1	+F	F	V	Sinon
	C2	V	+F	V	-
	C3	F	V	+V	-
Act.	A1	1	-	-	2
	A2	-	1	-	-
	A3	-		1	1

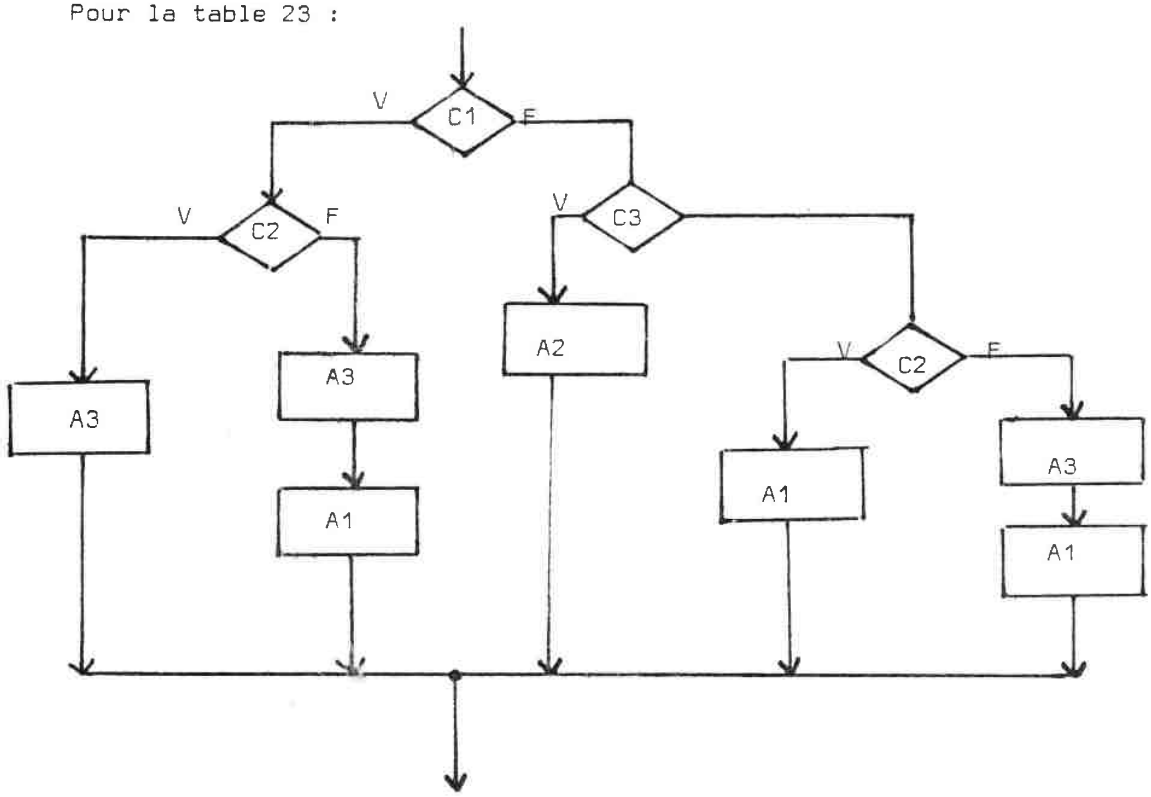
ft

L'introduction des conditions imposées + F et + V permet d'éviter un test (noté * sur le premier organigramme), comme on peut le voir sur les deux organigrammes suivants. Les équivalences proposées correspondent au cas où l'on cherche à minimiser le nombre de tests écrits dans l'organigramme.

Pour la table 22 :



Pour la table 23 :



5) La table 3 du paragraphe 5.21 pourrait être décrite exactement par une table CIVA, en appliquant les simplifications d'écriture b) et c) de 5.432

Sélection	Dimension de l'échelle =	4	4	4	8
	Unité à mesurer =	mv	mv	ma	ma
	Inf Plage mesure =	10	76	10	75
	Sup Plage mesure =	75	200	200	250
Actions	Nbre de tours =	24	45	24	50
	Nature du fil =	Cu	Cu	Cu	Al
	Diamètre =	16	8	16	16
	Numéro Schéma =	A1234	B5678	F9012	K7821
	N° Description =	10	17	10	7

5.45 - Tables de dimension 2.

5.451 - Exemple.

Une table de décision ou une table de sélection de dimension 2 est une écriture plus commode d'une table de décision ou de sélection telles que nous les avons définies. Cette simplification intervient dans le cas où la table contient des répétitions de groupes d'entrées de conditions. Ainsi la Table 24 pourrait être décrite plus aisément par la table 25.

Table 24

Décision													
C1	V				V				F				Sinon
C2	F				V				V				
C3	V		F		V		F		V		F		
C4	V	F	V	F	V	F	V	F	V	F	V	F	
Actions	A1	A2	A3	A4	A5	A6	A7	A6	A9	A10	A11	A12	A3

TABLE 25

Décision 2

	C1	V	V	F
	C2	F	V	F
Actions				
V	V	A1	A5	A9
	F	A2	A6	A10
F	V	A3	A7	A11
	F	A4	A8	A12
C3	C4			

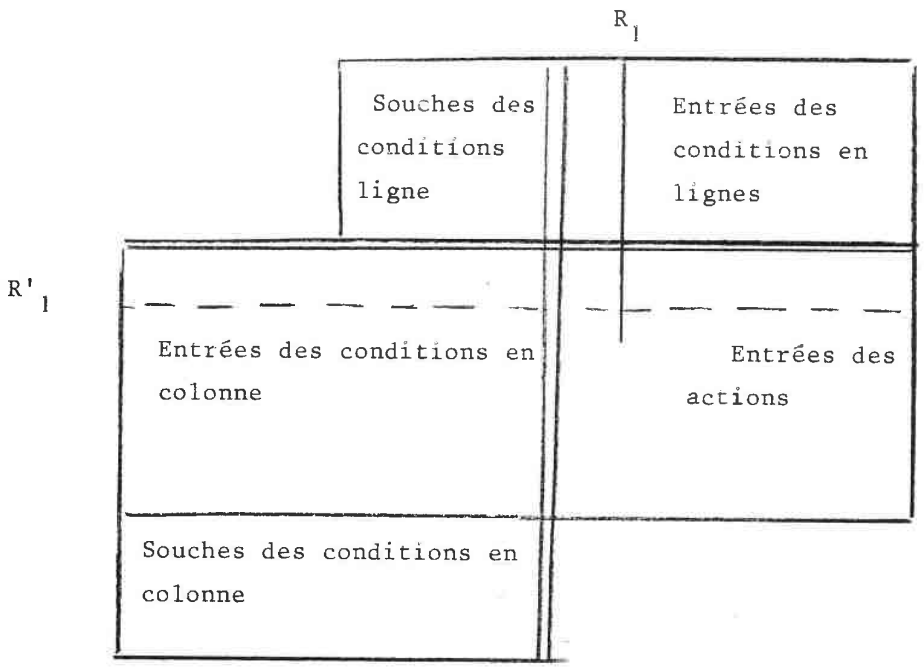
Fintable

Le double trait vertical évite d'avoir à introduire un nouveau symbole action. Ce double trait sera perforé.

5.452 Présentation d'une table de dimension 2.

Une table de dimension 2 commence par le symbole décision 2 ou le symbole sélection 2

Elle est constituée de 5 parties selon le schéma suivant :



Une souche de condition est définie comme en 5.411, une entrée de condition comme en 5.412. Les simplifications d'écriture a) et b) de 5.432 s'appliquent également : la simplification b) a été utilisée dans l'écriture de la table 25 pour la première colonne de condition.
Une entrée d'action est constituée d'une ou plusieurs instructions séparées par un point virgule.
Si une règle "sinon" est nécessaire, elle sera écrite après la dernière souche des conditions en colonne, sous la forme

<u>sinon</u> action <u>fintable</u>

Une forme particulièrement simple de table de décision ou de sélection de dimension 2 correspond au cas où toutes les souches sont vides ; les entrées de condition sont alors des expressions booléennes ;

Décision 2

	C1	C2	C3	--	Cn
CC1	A11	A12			A1n
CC2	A21				
CCm	Am1				Amn

Une table de dimension 2 sera perforée en utilisant les mêmes principes qu'en 5.433, c'est-à-dire en respectant la disposition adoptée par son dessin. Pour les conditions en colonne, communes à plusieurs colonnes (simplification a) de 5.432), la condition sera perforée dans la première ligne, les cases des lignes suivantes restant blanches.

5.453 Exécution d'une table de dimension 2.

Nous noterons R_i la ième règle portant sur des conditions en lignes (R_i est une colonne de la table) et R'_j la jème règle portant sur des conditions en colonne (R'_j est une ligne, et A_{ji} l'entrée d'action commune aux règles R_i et R'_j.

La valeur de la règle R_i est celle de l'expression :

$v\ (R_i) = v\ (ECL_{1i})$ <u>et</u> (ECL_{2i}) <u>et</u>.... <u>et</u> $v\ (ECL_{ni})$

où ECL_{ki} désigne la kième entrée de condition en ligne de la règle R_i.

La valeur de la règle R'_j est celle de l'expression :

$v\ (R'_j) = v\ (ECC_{j1})$ <u>et</u> $v\ (ECC_{j2})$ <u>et</u>..... <u>et</u> $v\ (ECC_{jm})$

où ECC_{jk} désigne la kième entrée de condition en colonne de la règle R'_j.

Exécuter une table de <u>décision de dimension 2</u>, c'est

- s'il en existe, exécuter toutes les actions A_{ij} telles que :

  $v\ (R_i)$ <u>et</u> $v\ (R'_j) = VRAI$;

- s'il n'en existe pas, c'est exécuter l'action de la règle "sinon" s'il y en a une, et s'il n'y en a pas, c'est exécuter une instruction vide.

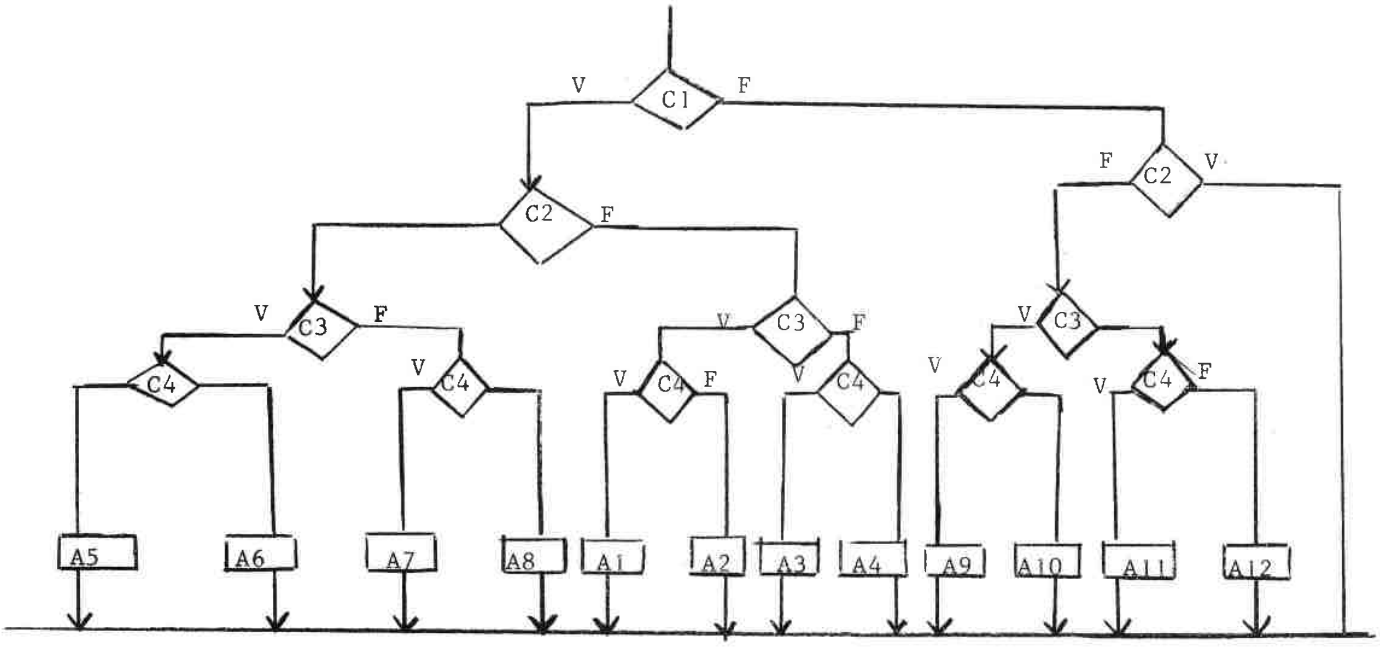
Exécuter une table de <u>sélection de dimension 2</u>, c'est

- si elle existe, exécuter l'action A_{ij} telle que

  $$i = \mathrm{Min}\left\{k \,\middle|\, v\ (R_k) = VRAI\right\} \text{ et } j = \mathrm{Min}\left\{l \,\middle|\, v\ (R'_l) = VRAI\right\}$$

- s'il n'existe pas de tel couple ij, c'est exécuter l'action de la règle "sinon" s'il y en a une, et c'est exécuter une instruction vide, s'il n'y en a pas.

Ainsi, la table 25 pourrait être évaluée selon le schéma suivant :



5.46 Expressions en table de sélection

Les tables proposées jusqu'à maintenant sont des instructions CIVA. Une forme de table couramment utilisée (commerce : catalogues de prix, administration : grille de salaire, etc...) consiste à déterminer la valeur d'une variable en fonction d'un certain nombre de conditions.

Ces tables apparaissent très tôt dans le dossier d'analyse et elles constituent le plus souvent des constantes de l'application. Il est donc souhaitable de les accepter dans le langage sous leur forme de présentation courante.

Les tables servent à **indiquer quelle valeur** attribuer à une grandeur, en fonction d'un certain nombre de conditions, ce qui peut s'écrire sous forme d'une instruction d'affectation, en considérant une telle table comme une expression. Les tables ayant valeur d'expression seront uniquement des tables de sélection : la valeur associée à la variable après évaluation de la table doit être définie et unique.

Exemple :

Table 26

Sélection	C1	V	V	F	Sinon
	C2	F	V	F	
	C3	V	V	F	
Action		G=1	G=3	G=4	G=52+B

Table 27

G=Sélection	C1	V	V	F	Sinon
	C2	F	V	F	
	C3	V	V	F	
Action		1	3	45	52 +B

Une entrée d'action dans une telle table ne peut contenir qu'une expression arithmétique.

Cette forme de table CIVA est bien adaptée à l'utilisation courante, même en dehors de l'informatique, comme on pourra le constater sur les exemples suivants.

Nous avons dit que cette forme de table apparaît très tôt dans un dossier d'analyse, et souvent bien avant leur utilisation : il convient donc de pouvoir les déclarer. Il suffit pour cela de déclarer dans une classe, une procédure de type fonction contenant une table de décision sous la forme :

<u>entier</u> <u>procédure</u> F = <u>sélection</u>.....<Table 27 par exemple>
<u>fin proc</u> ;

<u>Exemples</u> :

TABLE 28

(Extrait du catalogue "vacances 73" Touropa - Havas-Voyages)

PRIX PAR PERSONNE COMPRENANT : Le circuit et le séjour (voir dates départ des circuits, pages 80 à 83).

CIRCUIT Cat. Touriste*	2 semaines au Maroc			3 semaines au Maroc		
	VILLES IMPERIALES		Suppl. chambre indiv.	GRAND TOUR DU MAROC		Suppl. chambre indiv.
SEJOUR (1 semaine)	Demi-pension	Pension complète		Demi-pension	Pension complète	
	F	F	F	F	F	F
ANFA Hôtel Anfa Plage	**2 185**	**2 235**	**300**	**3 020**	**3 070**	**410**
MOHAMMEDIA Hôtel Samir	**2 340**	**2 560**	**205**	**3 170**	**3 400**	**315**
AGADIR (transport aérien CASABLANCA/ AGADIR compris)						
Base Hôtel KAMAL	**2 220**	**2 320**	**210**	**3 055**	**3 155**	**320**
Base Hôtel LA KASBAH	—	**2 090**	**215**	—	**2 925**	**325**
Base Hôtel ALMOHADES	**2 245**	**2 350**	**240**	**3 080**	**3 185**	**350**

TABLE 29

Calcul de la prime de salaire unique à compter du 1er Août 1971

POURCENTAGE D'ABATTEMENT	SALAIRE UNIQUE				
	1 enfant		2 enfants		3 enfants et plus
	Moins de 2 ans	De plus de 2 ans	Un ou deux de moins de 2 ans	Deux de plus de 2 ans	
0	97,25	38,90	97,25	77,80	97,25
1	96,50	38,60	96,50	77,20	96,50
2	95,50	38,20	95,50	76,40	95,50
3	94,50	37,80	94,50	75,60	94,50
4	93,50	37,40	93,50	74,80	93,50

(1) A l'exception de l'aîné des familles de moins de trois enfants

La table 28 pourrait être décrite facilement par une table de sélection de dimension 2. On supposera que durée, hôtel, circuit, séjour et chambre individuelle ont reçu une valeur avant l'appel de la table.

Prix = Sélection 2

	Durée =	2				3			
	Circuit =	'VILLES IMPERIALES'				'GRAND TOUR DU MAROC'			
	Séjour =	1/2 Pension		Pension complète		1/2 Pension		Pension complète	
	Chambre individuelle	V	F	V	F	V	F	V	F
Action	ANFA Plage	2185+ 300	2185	2235 + 300	2235	3020 + 410	3020		3070
	SAMIR	2340 + 205	2340	2560 + 205	2 560	3170 + 315	3170		3400
	KAMAL	2220 + 210	2220	2320 + 210	2320	3055 + 320	3055		3155
	LA KASBAH	2220 + 215	2220	2090 + 215	2090	3055 + 325	3055		2925
	ALMOHADES	2245 + 240	2345	2350 + 240	2350	3080 + 350	3080		3185
	HOTEL =								

ft

Quant à la table 29, sa description dans une procédure appelée prime de salaire unique, permettrait son utilisation dans une expression arithmétique.

Salaire = Salaire + prime de salaire unique.

Décimal procédure prime de salaire unique = Sélection 2

Salaire Unique	V				
Nbre enfants	= 1		= 2		> 2
Age 1 < 2	V	F	-	-	-
Age 1 > et Age 2 > 2	-	-	F	V	-
Actions					
0	97,25	38,90	97,25	77,80	97,25
1	96,50	38,60	96,50	77,20	96,50
2	96,50	38,20	95,50	76,40	95,50
3	94,50	37,80	94,50	75,60	94,50
4	93,50	37,40	93,50	74,80	93,50
Pourcentage d'abattement =	Sinon 0 Fintable ;				

On constatera également l'utilité de telles tables dans l'application décrite au chapitre 15.

REFERENCES DU CHAPITRE 5

AUBRY B., 73 - Traduction des tables de décision
Thèse de 3ème Cycle - Université de NANCY 10/3/73

BUFFET J., 72 - BUFFET J., ARNAL P., QUERE A.
Définition du langage algorithmique Algol 68
Hermann - PARIS 72.

CALLAHAN H. D., 67 - and CHAPMANN A. E.
Description of basic algorithls in DETAB 65
preprocessor
CACM - Vol 10, n° 7 - 1967. pp 441-446

IBM 65 - IBM TF 2 0007 - 11/8/65
Application des tables de décision

IBM 66 - IBM TF 2 0014 - 30/3/66
Vers l'automatisation des études de production

IBM 68 a - IBM-TF 2 0036 - 6/2/68
Le point sur les tables de décision

b - IBM-H 20 0492 - 1/10/68
System 360 Decision Logic translator (360 A-LX -32 X)
Application description manual

ICL 69 - ICL 1900 Series - Jan 69
Decision table with Cobol

KIRK H. W. 65 - Use of decision tables in computer programming
CACM vol 8 n° 1 1965 - pp 41-43

MARKIA A. 71 - Mea culpa - Mea maxima culpa
OI Informatique n° 1 1971 p. 23-28

PICARD 69 - Les tables de décision - Informatique et gestion
n°s 3 - 4 - 5.

CHAPITRE 6

INSTRUCTIONS SIMPLES

6.1 INTRODUCTION.

6.2 EXPRESSIONS ARITHMETIQUES ET BOOLENNES.

6.3 INSTRUCTION D'AFFECTATION.

6.31 Préambule.

6.32 Affectation de valeur à un objet de type simple.

6.33 Affectation de valeur à un objet de type structuré.

1 Emetteur de type file d'objets de type arithmétique.

2 Emetteur de type file de caractères.

3 Emetteur de type structuré.

6.4 INSTRUCTION "SORTIR".

6.5 BOUCLES CLASSIQUES : INSTRUCTION "POUR".

6.51 Syntaxe.

6.52 Remarques sémantiques.

6.53 Simplifications d'écriture. Instruction "tant que".

6.6 CONTROLES PENDANT L'EXECUTION.

6.61 Introduction.

6.62 Le mécanisme des contrôles.

6.63 Déclaration d'une variable contrôlable.

6.64 Déclaration d'un contrôle.

6.65 Durée de définition d'un contrôle.

6.66 Libération, suspension, remplacement d'un contrôle.

1 Etats d'une variable contrôlable.

2 Instructions de changement d'état d'une variable contrôlable.

3 Graphe des états d'une variable contrôlable.

6.67 EXEMPLES D'UTILISATION.

1 Utilisation de "libérer".

2 Asservissements réciproques de deux variables.

3 Modification d'un contrôle.

4 Réalisation de contraintes.

5 Remarque sur l'utilisation de "sortir".

CHAPITRE 6

INSTRUCTIONS D'AFFECTATION, D'ITERATION CLASSIQUE ET DE CONTROLE.

6.1 INTRODUCTION.

Nous avons regroupé dans ce chapitredes constituants divers du langage dont l'importance n'est pas primordiale pour la présentation du projet. Nous traitons d'abord rapidement les expressions arithmétiques et booléennes et les instructions d'affectation et d'itération classique. Ces concepts sont bien connus dans les langagesde programmation et nous nous sommes contentés de présenter ici quelques aspects particuliers. Les choix opérés nous ont été inspirés par un souci de clarté et de sécurité de la programmation, plutôt que par la recherche d'une plus grande généralité des notions qui conduit à un langage où tout est possible et rien n'est plus tout à fait sûr pour l'utilisateur.

Restreindre l'étendue des possibilités du langage de façon très stricte n'est certainement pas souhaitable. Nous ne le faisons que dans le langage de base. L'introduction de fonctions prédéfinies, l'utilisation du métalangage (cf. chap. 9) permettent de décrire toutes les combinaisons souhaitables tant du point de vue des conversions autorisées que des formes de l'instruction d'affectation elle-même. Mais elles sont alors demandées explicitement par l'utilisateur et non implicitement ce qui nuirait à la sécurité.

Les instructions de contrôles sont moins courantes dans les langages de programmation. La nécessité du contrôle des informations traitées est évidente pour l'analyste, et nous devons lui proposer des outils permettant de décrire et réaliser facilement les contrôles qu'il souhaite :

- contrôle de validité des données à leur entrée en ordinateur, c'est ce que nous verrons en étudiant les problèmes d'acquisition des données,

- contrôle des résultats produits (il s'agit de contrôles à postériori, demandés pendant l'édition des résultats par exemple).

- contrôle continu pendant l'exécution : chaque fois qu'un objet du langage reçoit une valeur, il peut être nécessaire d'effectuer un contrôle sur la valeur qui lui est donnée et éventuellement d'entreprendre un traitement particulier dans les cas défectueux.

C'est le but des instructions de contrôle qui seront vues ici. Outre la description des contrôles pendant l'exécution, elles permettent également la mise au point aisée des chaînes de traitement comme nous le verrons au chapitre 10.

6.2 EXPRESSIONS ARITHMETIQUES ET BOOLEENNES

Il s'agit des expressions classiques, voir par exemple (Genuys, 60), (Colliet,) ou (Veillon, 72).

Les opérandes d'une expression arithmétique sont des constantes, des identificateurs simples ou indicés ou des indicateurs de fonction suivies des paramètres effectifs de la fonction. Il n'y a pas de commentaire dans les paramètres comme en Algol 60.

Les valeurs des opérandes sont des valeurs binaires pour les opérandes de type entier, réel ou complexe (représentation des informations sous leur format interne cf. introduction) ou des valeurs décimales (type décimal).

Les opérandes d'une expression arithmétique doivent être de même type ou de type compatible : deux types seront dits compatibles si la conversion implicite d'une valeur d'un type en une valeur de l'autre type est définie.

Les seules conversions implicites sont définies ci-dessous :

- les types arithmétiques sont classés dans l'ordre de priorité croissante :

entier, décimal, réel, complexe

- si les deux opérandes d'une opération binaire sont de même type, il n'y a pas de conversion, le résultat est aussi du même type
- si les deux opérandes sont de type différents, le résultat est du type prioritaire, et il y a une conversion de l'opérande de type non prioritaire en une valeur du type prioritaire.

Des conversions peuvent être demandées explicitement par l'intermédiaire de fonction prédéfinie à un opérande, qui s'écrivent sous la forme :

nom du type demandé (paramètre)

Exemples :

entier (X), réel (X).

Ces fonctions sont également définies pour des paramètres d'un type non arithmétique.

Les expressions booléennes ne portent que sur des relations et des opérandes de type booléens. Les opérateurs utilisés sont :

$<$, $\leqslant$, $>$, $\geqslant$, $\neq$, = = pour les opérateurs de relation (= = désigne l'opérateur d'égalité).

NON, ET, OU pour les opérateurs booléens.

6.3 INSTRUCTION D'AFFECTATION.

6.31 Préambule.

Cette instruction permet d'attribuer une valeur à un objet. C'est une instruction classique, qui s'écrit presque toujours sous la forme d'un identificateur (le récepteur), suivi d'un symbole d'affectation (pour nous, c'est =), suivi d'un moyen de désigner une valeur (l'émetteur) notation de valeur, expression, ou identificateur (il y a alors un "dérepérage").

Il y a donc peu de chose à dire sur la syntaxe d'une telle instruction. Quant à la sémantique, elle dépend essentiellement des types de l'émetteur et du récepteur. Ceux-ci peuvent être particulièrement complexes, un champ d'une structure pouvant être une file de type quelconque, dont les éléments sont eux-mêmes une structure....Or, tant que les types de l'émetteur et du récepteur sont simples, sans chercher à définir ce terme maintenant, il est relativement facile de fixer la définition de l'affectation dans chacun des cas, les conventions possibles n'étant pas trop nombreuses. Il n'en est plus de même dès que l'on introduit des structures et la possibilité de combiner files et structures.

Une première solution consiste alors à multiplier le nombre d'instructions de transfert avec des conventions différentes (Move et Move corresponding de Cobol, affectation et affectation "by name" de PL 1, etc...) mais elle ne résout pas entièrement le problème.

Une autre solution consiste à ne définir qu'une instruction d'affectation entre objets de type simple et à donner la possibilité à

l'utilisateur de construire à partir de celle-ci toutes les autres formes de transfert qu'il peut désirer, en utilisant un métalangage. L'introduction d'un métalangage dans le projet étant également justifiée par ailleurs (métamodules (cf. 2.3), ou les problèmes d'acquisition et d'édition (cf. 2.33, chap. 1) et (Chabrier J. J., 73)), c'est cette seconde solution que nous avons choisie.

On trouvera donc dans ce paragraphe la définition de l'instruction d'affectation dans les cas simples. Pour la description des cas de transferts plus complexes, on trouvera la définition du métalangage au chapitre 9 et dans (Benamghar L. 73), et des exemples de métamodules de transfert aux chapitres 9, 13 et 14.

6.32 Affectation de valeur à un objet de type simple.

Nous désignons par objet de type simple un objet du type caractère, booléen ou type arithmétique. Il peut être :

- un élément d'une file : il est désigné par un identificateur indicé,
- un champ d'une structure : il est désigné par un identificateur, éventuellement qualifié par l'identificateur d'un de ses prédécesseurs dans la structure, s'il y a un risque d'ambiguïté : numéro de client, par exemple,
- un objet indépendant : il est désigné par un identificateur.

L'opérateur d'affectation est le signe = .
Nous appelons récepteur, l'objet désigné à gauche du signe = et émetteur, celui désigné à sa droite.

C'est le récepteur qui impose son type.
Si l'émetteur et le récepteur sont tous deux de type arithmétique, l'émetteur est converti dans le type du récepteur même si celui-ci est moins prioritaire. Toutefois la traduction d'une instruction d'affectation à un décimal, d'une valeur de type réel ou complexe entraîne l'émission d'un message d'attention à la compilation.

Nota : l'affectation de valeur à un objet de type file est une opération qui porte sur la totalité de la file et elle sera étudiée au chapitre 7 : traitement global des files et des ensembles.

Le cas où l'émetteur est une file de caractères sera examiné en 6.322.

6.33 Affectation de valeur à un objet de type structuré.

Il s'agit ici de l'affectation d'une valeur structurée et non pas de l'affectation d'une valeur simple à l'un des champs de la structure, cas qui relève du paragraphe précédent.

6.331 Emetteur de type file d'objets de type arithmétique.

La structure ne doit pas avoir de champ de type file de taille bornée ou variable.

La file émettrice est alors explicite, c'est-à-dire que l'affectation A = B est considérée comme une suite d'affectations.

MA (1) = B (1) ; MA (2) = B (2) ;..... MA (n) = B (n)
et sera possible si chacune d'elles l'est. MA désigne le mot des feuilles de la structure A et ne désigne le minimum de la taille du mot des feuilles de A et de la taille actuelle de B. B est une file ou plus généralement une désignation de files ou sous files.

6.332 Emetteur de type file de caractères.

Nous avons vu dans l'introduction de ce travail que les valeurs traitées par Civa sont conservées sous leur représentation "interne" pour favoriser les traitements. Les données destinés à un traitement doivent donc être soumises à un module d'acquisition qui fera le changement de présentation du format externe vers le format interne, et procédera aux contrôles souhaités. Cette technique est particulièrement utile dans le cas de données groupées en fichier. Pour les autres, elles pourront être lues sous forme d'une chaîne de caractères et converties ensuite sous une forme propice aux traitements. C'est le rôle de l'instruction d'affectation, lorsque l'émetteur est une file de caractères, de permettre ces conversions. Il n'est pas question dans une instruction de ce type de faire référence à un modèle de présentation externe, un "format" : nous considérerons que la file de caractères est une écriture de valeurs numériques en "format libre" séparées par des blancs.

Ainsi A = B, si A est entier et B une file de caractères permet d'attribuer à A la valeur décrite en décimal par les caractères contenus dans la file. La valeur est celle décrite par la chaîne de

caractères comprise entre le début de la file et le premier caractère non numérique ou la fin de la file.

Si A est une structure, l'affectation A = B est explicitée sous la forme :

MA (1) = B (bi : i_1) ; MA (2) = B (i_1 : i_2) ;.....; MA (n) = B (i_{n-1} : bs)

ou bi et bs représentent la borne inférieure et la borne supérieure de B. Les valeurs de i_1 , i_2,.....i_{n-1} sont déterminées par les champs de la structure réceptrice.

Si le jème champ est une file de caractères de taille fixe k : i_j = i $i_j = i_{j-1} + k$.

Si le jème champ est de type numérique, la portion de file émettrice considérée pour ce jème champ est limitée par le premier blanc suivant un caractère non blanc après B (i_j - 1). Soit ℓ la taille de cette sous file de la file émettrice, représentant un objet du type du champ récepteur ou d'un type compatible :

$i_j = i_j + \ell$.

Exemples :

1) sec soc struct (sexe car, année (2) car, mois (2) car, dept (2) car, com (3) car, num (3) car) ;

f (13) car ;

L'instruction sec soc = f est équivalente à la suite :

sexe = f (1) ; année = f (2 : 4) ; mois = f (4 : 5) ; dept = f (6 : 7) ; com = f (8 : 10) ; num = (11 : 13) ;

2) g (max = 30) car ;

h struct (a entier, b réel, c entier, d entier) ;

g = ' 125␣-9 . 5 E + 05 ␣ - 119 ␣2 ' ;

L'instruction h = g serait alors équivalente à la suite d'instructions

a = g (1 : 3) ; b = g (5 : 13) ; c = g (14 : 18) ; d = g (19 : 19) ;

c'est-à-dire qu'après h = g, a vaut125, b vaut -9.5 E + 5, c -119 et d vaut 2.

6.333 Emetteur de type structuré.

Si l'émetteur et le récepteur sont de type identique, l'affectation A = B est équivalente à la suite d'instructions d'affectation entre les champs de A et B. Elle ne pose aucun problème : il n'y a aucune conversion et A et B peuvent être des structures quelconques.

Dans le cas général, la structure réceptrice ne doit pas contenir de file de longueur variable ou bornée. L'affectation entre deux structures est assimilée à une suite d'instructions d'affectation entre chacune de leurs feuilles dans l'ordre du mot des feuilles de chaque structure : les files de taille fixe étant elles-mêmes décomposées en la suite de leurs éléments.

Exemple :

h struct (sexe car, date (année (2) car, mois (2) car), lieu (dept (2) car, com (3) car, num (3) car)) ;

g struct (sexe car, date (4) car, lieu (8) car) ;

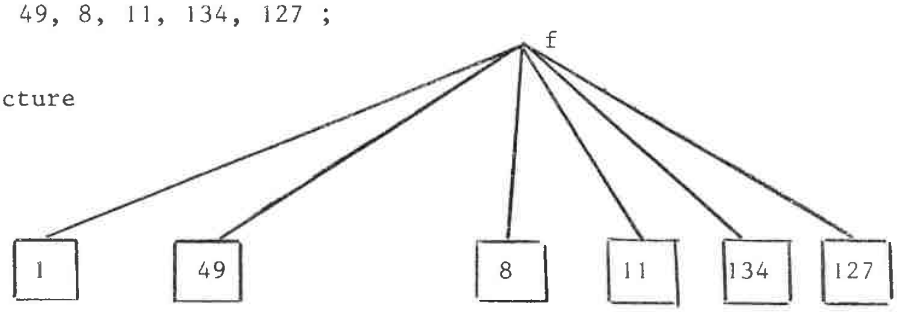
f struct (sexe entier, année entier, mois entier, dept entier, com entier, num entier) ;

sec soc struct (sexe car, année (2) car, mois (2) car, dept (2) car, com (3) car, num (3) car) ;

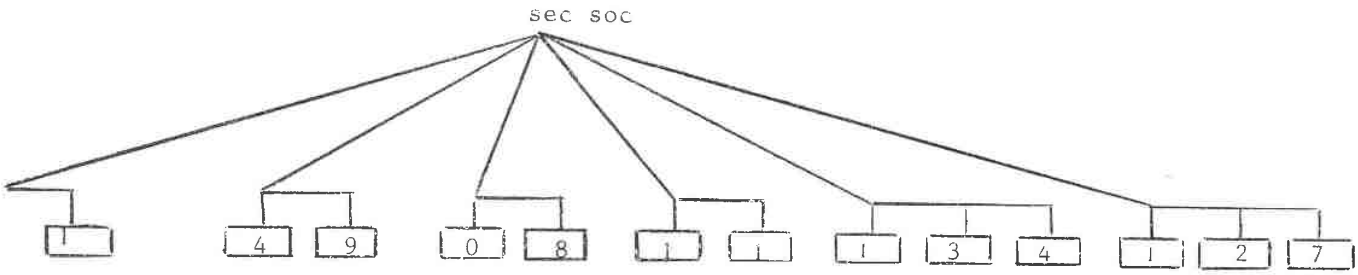
h = sec soc ; g = h ; f = h ; sec soc = f ; h = f ;
sont des instructions d'affectation permises.
supposons qu'on ait

f = 1, 49, 8, 11, 134, 127 ;

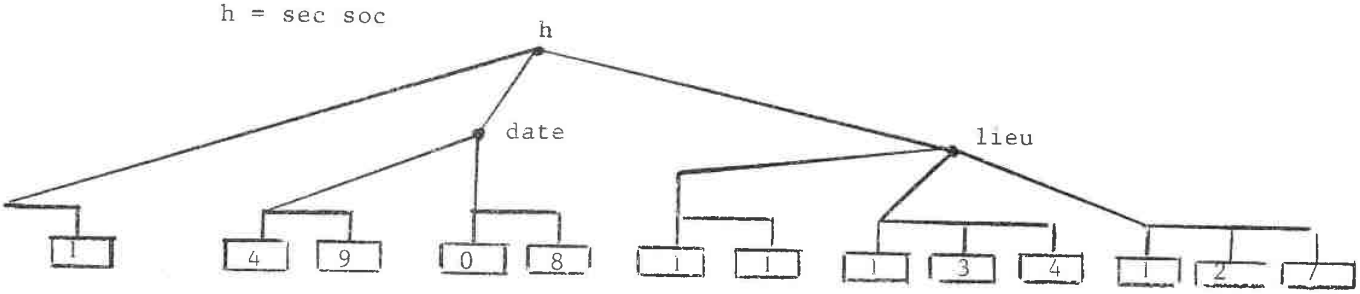
f a la structure



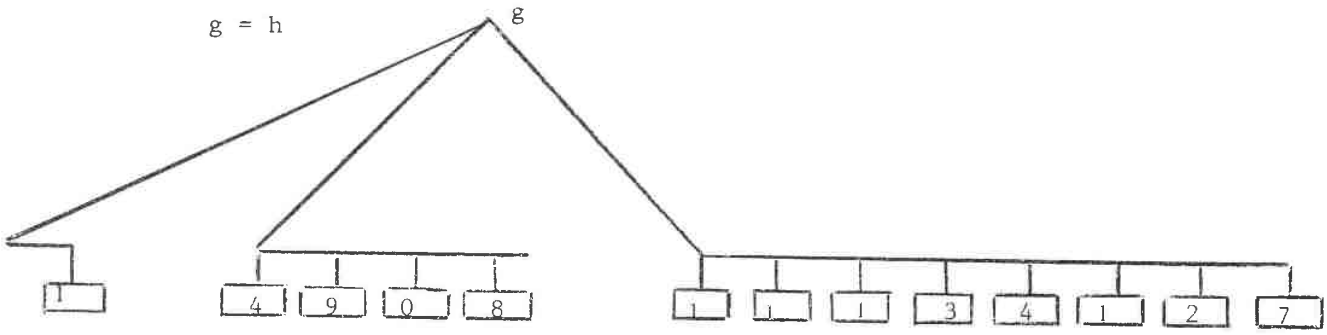
ses soc = f



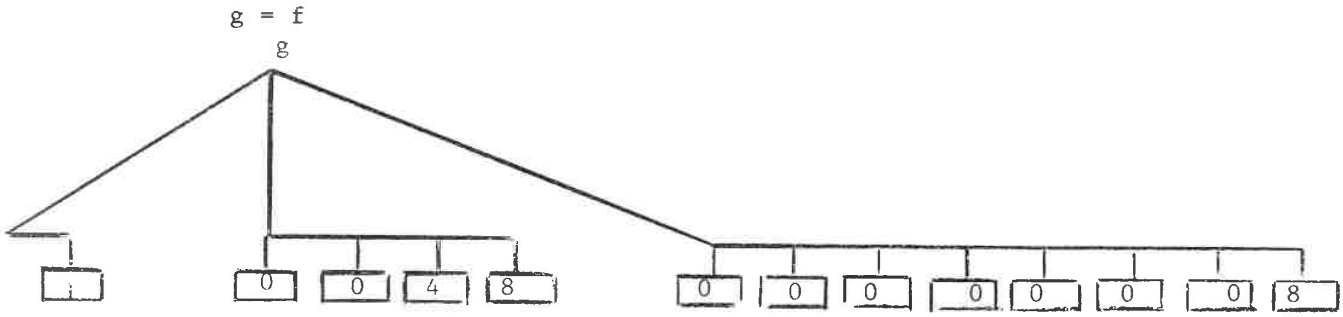
h = sec soc



g = h



Par contre l'instruction g = f conduit à un message d'attention à la compilation (structure incomplète) ; on aurait



6.4 INSTRUCTION "SORTIR".

Nous avons vu dans l'introduction que la modularité de la description des actions et les instructions introduits dans le langage permettent d'éliminer complètement les instructions de branchement. Il pourrait en être de même de l'instruction de retour au module appelant ("Return" de fortran, par exemple). Nous avons cependant préféré définir une instruction "sortir", qui permet de mettre fin à l'exécution d'un module ou d'une procédure : il y a alors reprise de l'exécution de la procédure ou du module appelant.

Il est assez commode et il nous semble assez naturel, qu'au cours de la description d'une action élémentaire (une procédure ou un module), on dise que cette action est terminée. Si, l'endroit où l'on veut indiquer cette fin d'éxécution coïncide avec la fin de la description, il n'est besoin de rien ajouter. Si elle ne coïncide pas, l'absence d'instruction "sortir" conduirait à nécessiter une réorganisation de la description pour qu'il y ait coïncidence. Ce qui ne nous semble pas souhaitable.

Elle s'écrit simplement : sortir.

6.5 BOUCLES CLASSIQUES : INSTRUCTION "POUR".

La plupart des calculs itératifs correspondent au traitement global d'une ou plusieurs files. Ils sont étudiés au chapitre 7. Il subsiste cependant des calculs itératifs qui échappent à cette règle, pour lesquels les boucles classiques conviennent bien. Nous définissons donc des instructions "pour".

6.51 Syntaxe.

<instruction pour> := pour <élément de pour> <fin de pour>
| pour <identificateur> <fin de pour>.
| <tant que> < fin de pour>

<élément de pour> : := <identificateur> de <exp. arithmétique>
[à <exp. arithmétique>] [pas <exp. arithmétique>]
[< tel ou tant que>] .

<tel ou tant que> : := <tel que> [<tant que>] | <tant que> [<tel que>].

<tel que> : := tel que <exp. arithmétique> .

<tant que> : : = tant que <exp. arithmétique> .

<fin de pour> : := faire <instruction> [;<instruction>]* fp.

6.52 Remarques sémantiques.

Pour I de m1 à m2 pas m3 faire proc fp ; avec I, m1, m2, m3 entiers est une instruction "pour" simple (1).

(1) On pourrait considérer que la suite "I de ms à m2 pas m3 " constitue la déclaration de l'élément courant I associée à la file des constantes entières : m1, m2, pas m3, et considérer le "pour" comme un cas particulier de l'instruction "pour chaque" du chapitre 7.

- I, m1, m2, m3 sont d'un type arithmétique.

- Les trois expressions doivent être calculables à la rencontre de cette instruction. Elles sont calculés une fois pour toutes à l'entrée dans la boucle, m1 représente la valeur initiale du compteur, m2 sa valeur finale et m3 le pas de la progression ; si m3 n'est pas mentionné, le pas vaut 1.

(*) - Si m1, m2, m3 sont tels que (m2 - m1)* m3 < 0, les instructions entre faire et fp ne sont pas exécutées et le traitement de cet élément de pour est terminé ; sinon, les instructions écrites entre faire et fp sont exécutées, la variable contrôlée I est augmentée de m3 et on recommence en (*).

Lorsque le traitement d'un élément de pour est terminé, on passe du traitement de l'élément suivant dans la liste d'éléments de pour s'il en existe un ; s'il n'en existe pas, l'exécution se poursuit à l'instruction qui suit immédiatement fp.

L'option <tel que> permet de qualifier explicitement les valeurs de la variable contrôlée pour lesquelles on veut exécuter la boucle. En (*), on évalue l'expression booléenne qui suit tel que : si elle est vraie, on effectue normalement la boucle, sinon on passe à la valeur suivante de la variable contrôlée, sans exécuter la boucle.

L'option <tant que> permet d'indiquer une condition d'abandon de la boucle : l'expression booléenne qui suit tant que est calculée en (*). Si elle est vraie, la boucle est exécutée normalement ; sinon, on passe à l'élément suivant de la liste de pour.

L'instruction : pour I de m1 a m2 pas m3 tel que E1 tant que E2 faire proc fp est donc équivalente à la séquence d'instruction :

I = m1 ;

Conditions	(m2 - m1 ✳ m3 > 0	V	V
	E1	V	F
	E2	V	V
Actions	proc	1	
	I = I + m3	2	1
	retable	3	2

ft ;

6.53 Simplifications d'écriture. Instruction "tant que".

a) L'expression m3 peut être absente : sa valeur est implicitement 1.

b) L'expression m2 peut être absente. Dans le test de fin de boucle, on considère alors que l'expression (m2 - m1)✳ m3 < 0 est toujours vraie. L'instruction doit donc compter nécessairement une option tant que.

c) Un élément de pour peut donc se réduire à une seule expression arithmétique et donc à un seul identificateur. Ceci n'a d'utilité que s'il existe plusieurs éléments de liste de pour.

d) L'instruction pour < identificateur > faire < instruction > fp qui n'aurait pas d'intérêt dans le sens général, est considérée comme une simplification d'écriture de l'instruction.

pour <identificateur > de 1 à max entier pas 1 faire < instruction > fp ou max entier désigne le plus grand entier accessible en mémoire lorsque l'identificateur est du type entier.

Si la valeur max entier est atteinte, il y a édition d'un message d'erreur à l'exécution (débordement entier).

Lorsque l'identificateur est réel max entier est remplacé par max réel.

e) Une instruction d'itération classique peut se réduire à la forme

tant que <expr. booléenne > < fin de pour> .

(✳) L'expression booléenne est évaluée, si elle est fausse l'exécution de "tant que" est terminée sinon on exécute la suite d'instructions écrites entre faire et fp et on recommence en (✳).

6.6 CONTROLES PENDANT L'EXECUTION.

6.61 Introduction.

Ce paragraphe s'inspire du travail effectué par J. Ducloy dans le cadre de ce projet (Ducloy J. 73).

Par contrôles pendant l'exécution, nous entendons la possibilité offerte à l'utilisateur de demander que des contrôles soient faits automatiquement sur les valeurs prises par les informations qu'il traite et de décrire des traitements à entreprendre lorsque les conditions imposées ne sont pas vérifiées ou, plus généralement lorsque survient un évènement.

Un évènement est associé à un changement d'état d'un objet. Il peut être caractérisé par l'état final de ce changement et par l'instant où il survient.

Exemples :

A = 3 ; est une instruction qui demande que A contienne la valeur 3.

L'éxécution de cette instruction est un évènement : il est caractérisé par l'instant du changement d'état de A et par le fait que A contienne 3.

Une nouvelle exécution de A est un autre évènement.

L'exécution d'un programme est donc une suite d'évènements. Le texte du programme prévoit habituellement, l'ordre de succession des instants d'apparition des évènements. En particulier, il prévoit que faire après la réalisation d'un évènement.

Dans de nombreux cas, l'utilisateur veut qu'une action spécifiée par lui soit entreprise après la réalisation d'un évènement, sans qu'il ait à se soucier de l'instant précis où il survient (il s'agit là d'une description en mode déclaratif du chapitre 2).

Pour cela, il faut que l'utilisateur définisse l'état final d'un objet tel qu'un changement aboutissant à cet état constitue un évènement : il doit décrire ainsi une classe d'évènements, et que le système détecte les instants ou un évènement de cette classe apparaît.

6.62 Le mécanisme des contrôles

Une classe d'évènements correspond à la modification d'un objet pour aboutir à un état déterminé. En Civa, il s'agira de la modification du contenu d'un objet pour aboutir à une valeur déterminée ou ayant des propriétés déterminées.

Une classe d'évènements est donc caractérisée par un objet, nous dirons une variable contrôlée et une condition portant sur les valeurs de cet objet.

Un contrôle consiste à associer, pendant une certaine durée, une action à entreprendre à chaque arrivée d'un évènement d'une classe, associée à une variable contrôlée.

Un objet susceptible d'être contrôlée est dit variable contrôlable.

Nous définirions des instructions permettant de définir la durée pendant laquelle un contrôle est associé à une variable contrôlable.

Un contrôle est donc caractérisé par :

- une variable contrôlable - c'est l'objet sur lequel porte le contrôle,
- une durée de validité de ce contrôle : la variable est dite contrôlée pendant cette durée,
- une condition - elle définit l'état final d'une modification de la variable contrôlée, donc une classe d'évènements,
- une action, à exécuter lorsque survient un évènement de la classe définie ci-dessus.

6.63 Déclaration d'une variable contrôlable.

Un objet est une variable contrôlable si un état de cet objet peut constituer la définition d'une classe d'évènements. Pour indiquer cette possibilité, on fait suivre sa déclaration du symbole contrôlable.

Une variable contrôlable peut être contrôlée dans tout module dans lequel une occurrence de son identificateur est valide (cf. 1.4).

Elle devient "contrôlée" par l'intermédiaire d'une instruction "quand".

Nous avons limité l'ensemble des objets pouvant être déclarés contrôlables, aux objets de type simple : booléen, entier, réel, décimal, champ d'une structure ou élément courant d'une file des types précédents.

Exemples :

i entier contrôlable ;

x de f (15) entier contrôlable ; déclare l'élément courant

x attaché à la file f (voir 7.3) et en fait une variable contrôlable.

De nombreux objets de la classe "système" (cf. 1.3) sont déclarés contrôlables : ils caractèrisent des évènements indépendants du programmeur tels que les évènements externes, les fins de fichiers, les erreurs d'entrée sortie etc... et toutes les "erreurs " détectées par le matériel ou par le système d'exploitation.

6.64 Déclaration d'un contrôle.

L'instruction "quand" permet de faire d'une variable contrôlable une variable contrôlée, en lui associant une condition pour définir une classe d'évènements et une action à entreprendre lorsque survient un évènement de cette classe. C'est une instruction : elle est exécutée à un moment déterminé par sa place dans un module, mais c'est également une déclaration d'un contrôle. C'est une instruction du mode déclaratif du chapitre 2, en ce sens qu'elle annonce ce qu'il faut faire, sans préciser comment.
Elle s'écrit :

quand <évènement> [, <évènement>]* faire <instruction>
[; instruction]* fqd

<évènement> : := <identificateur> [affecte]
| <identificateur> <comparateur> <expression arithmétique>

<comparateur> : := < | ≤ | > | ≥ | ≠ | = =.

<évènement> définit une classe d'évènements par une condition simple ou de façon inconditionnelle.
Lors d'une modification de la variable contrôlée (affectation, ou "en" de 7.43 1), si la condition décrite <évènement> est vérifiée la suite d'instructions qui suit faire est exécutée. L'exécution se poursuit ensuite à l'endroit suivant la modification de la variable contrôlée.

Nous appellerons <u>séquence de récupération</u> cette suite d'instructions écrites entre <u>faire</u> et <u>fqd</u>.

Les instructions <u>quand</u> i > j <u>faire</u> proc <u>fqd</u>

et <u>quand</u> j < i <u>faire</u> proc <u>fqd</u>

ne définissent donc pas le même contrôle car elles ne définissent pas la même classe d'évènements.

Si l'instruction "quand" ne comporte pas de condition, la séquence de récupération est exécutée à chaque modification de la variable contrôlée.

Si la variable contrôlée est de type booléen, l'introduction d'un symbole supplémentaire permet d'éviter des ambiguïtés :

<u>quand</u> I <u>faire</u> proc <u>fqd</u>

signifie "chaque fois que I devient vrai, faire proc", tandis que

<u>quand</u> I <u>affecté</u> <u>faire</u> proc <u>fqd</u>

signifie "à chaque affectation à I, faire proc".

Si la variable contrôlée n'est pas booléenne, les deux formes ci-dessus sont équivalentes.

<u>Exemple</u> : (Ducloy, 73)

```
module m ;
        i, j entier contrôlable ;  i = j = 0 ;
        quand  i  faire  imprimer  "i = " , i  fin ;
        quand  j > 0  faire  imprimer  " j = " , j  fin ;
j = 1 ; j = 0 ; i = 1 ; i = j ; j = i ; i = j = 1 ; i = j = -1 fmod
```

L'exécution de ce module conduirait aux impressions :

```
j = 1
I = 1
I = 0
J = 1
I = 1
I = -1
```

6.65 Durée de définition d'un contrôle.

La durée maximale de définition d'un contrôle est la durée de validité de l'unité, module ou classe, dans laquelle il se trouve déclaré, et non la durée de vie de la variable contrôlable.
Un contrôle est en effet l'association d'une variable contrôlée et d'une séquence de récup ration, et il est nécessaire que les deux existent pour que le contrôle soit défini. Notons cependant qu'un contrôle déclaré dans un module, sera effectif dans tout module appelé dans lequel une occurrence de la variable contrôlée est valide, s'il est appelé pendant la durée de définition de ce contrôle.

Exemple : (Ducloy, 73)

```
module m ;                          module m1 ;
      utilise c ;                         utilise c ;
      m1 ;                              quand i faire action fqd ;
      .....                              ......
      i = i + 1 ;                       fin mod ;
      ......
      fin mod ;
classe c ; i entier contrôlable ;..... fin classe ;
```

Le contrôle défini dans m1 ne porte pas sur l'instruction i = i + 1 de m, qui ne définit donc pas un évènement de la classe d'évènements définie dans m1.

Si l'on veut que le contrôle déclaré dans m1, soit ensuite valide dans m, il faut le définir dans c, en modifiant m et c comme suit :

```
module m1 ;  m                      classe c ; i entier contrôlable ;
     utilise c ;                         procédure contrôle i ;
     contrôle i ; m1 ;                      quand i faire action fqd fin proc
     ........                           fin classe ;
     fin mod
```

En règle générale, on aura intérêt à définir les contrôles dans les unités où sont définies les variables contrôlées.

La durée de définition d'un contrôle commence à l'exécution d'une instruction "quand" et se termine à la fin de l'exécution du module

exécutant cette instruction "quand" sauf si cette durée est limitée explicitement :

- momentanément, par l'intermédiaire d'une instruction "suspendre"
- définitivement, par l'intermédiaire d'une libération : instruction "libérer" ou déclaration d'un nouveau contrôle associé à la même variable contrôlable.

L'étude des états d'une variable contrôlable permettra de préciser le rôle de ces instructions.

6.66 Libération, suspension, remplacement d'un contrôle.

6.661 Etats d'une variable contrôlable.

- libéré si elle n'a jamais été contrôlée ou si elle a été explicitement libérée par une instruction "libérer". Aucun contrôle n'est effectué sur une variable contrôlable à l'état "libéré".
- contrôlé ; elle passe dans cet état à la suite d'une instruction "quand". Le contrôle déclaré par "quand" est effectué, lorsque la variable est dans cet état.
- suspendu ; elle passe dans cet état à la suite d'une demande de suspension explicite, c'est l'instruction "suspendre", ou implicite au début d'une séquence de récupération d'un contrôle portant sur cette variable. Elle quitte cet état à la suite d'une instruction de reprise ou de libération.

Une variable contrôlable ne peut être contrôlée que par un seul contrôle : une instruction "quand" fait passer une variable contrôlable à l'état contrôlé quelque soit son état précédent et, si un contrôle était déjà associé à la même variable contrôlée, celui-ci disparaît.

6.662 Instructions de changement d'état d'une variable contrôlable.

L'instruction "libérer" permettant de limiter définitivement la durée de définition d'un contrôle s'écrit :

libérer <identificateur> [, < identificateur >]*.

Elle a pour effet de faire passer à l'état "libéré" toutes les variables contrôlées désignées par les identificateurs de la liste qui suit "libérer".

On peut donc considérer que la fin du module dans lequel une instruction "quand" définit un contrôle sur une variable, comporte une instruction "libérer" implicite pour cette variable.

La manipulation des contrôles nécessitant certaines précautions, dues essentiellement à la difficulté de modifier une variable contrôlée dans une séquence de récupération, il nous a paru utile de joindre deux instructions, permettant de suspendre provisoirement un contrôle et de le reprendre ultérieurement. Elles s'écrivent :

suspendre <identificateur> [, < identificateur >]*.

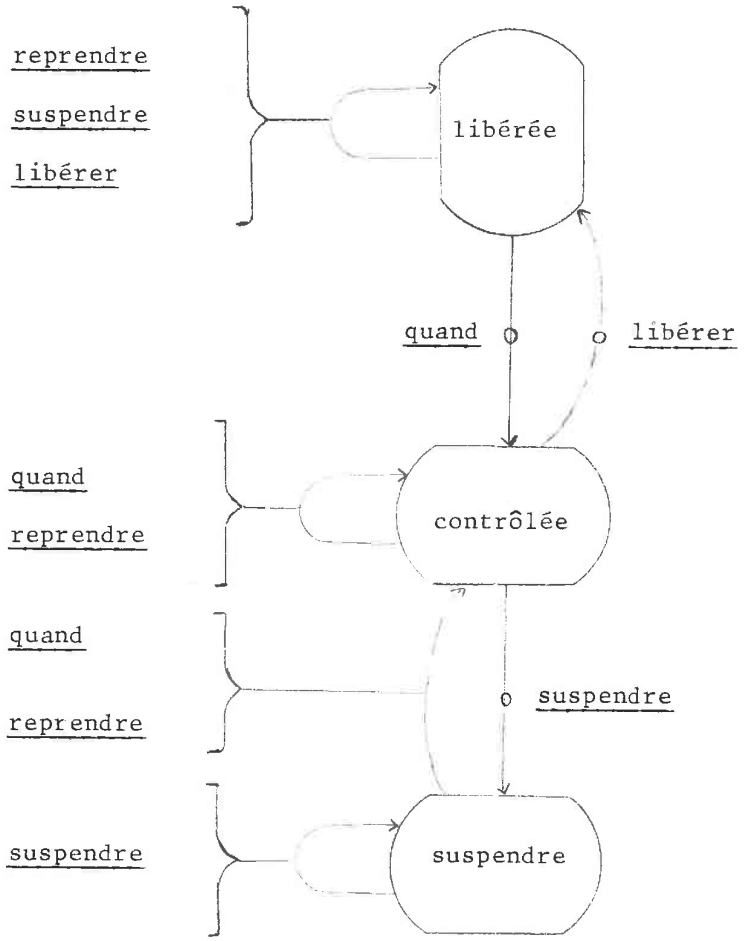
reprendre <identificateur> [, < identificateur >]*.

Une instruction "suspendre" fait cesser la durée de définition des contrôles associés à chacun des identificateurs de variable contrôlée de la liste et "reprendre" permet de redéfinir les mêmes contrôles.

Une instruction "suspendre" est implicitement insérée en tête d'une séquence de récupération d'une instruction "quand" et une instruction "reprendre" est insérée en fin de cette séquence. Elles portent toutes deux sur la variable contrôlée par "quand".

Une instruction "suspendre" sur une variable libérée ou suspendue n'a aucun effet, tout comme une instruction "reprendre" sur une variable libérée ou contrôlée.

6.663 Graphe des états d'une variable contrôlable.



6.67 Exemples d'utilisation.

Ils sont tirés de (Ducloy J., 73).

6.671 Utilisation de "libérer".

a) module n ;

```
j entier ; i entier contrôlable
quand i = 0 faire j = 1 ; libérer i fqd ;
i = 1 ;..........  ; j = 0 ;
i = 0 ;
imprimer j ; co provoque l'impression de 1 oc ;
j = 0 ; i = 0 ;
imprimer j ; co provoque l'impression de 0 ; la
                variable i ayant été libérée par
                l'exécution de la séquence de récu-
                pération lors de l'affectation
                précédente oc ;
fin mod ;
```

b) i entier contrôlable ;
quand i >10 faire i = 15 fqd

est une instruction correcte, car lors de l'exécution de i = 15, la variable contrôlable i a été suspendue et il n' y a aucun risque de "bouclage".

6.672 Asservissements réciproques de deux variables

On veut que les variables i et j aient toujours la même valeur.

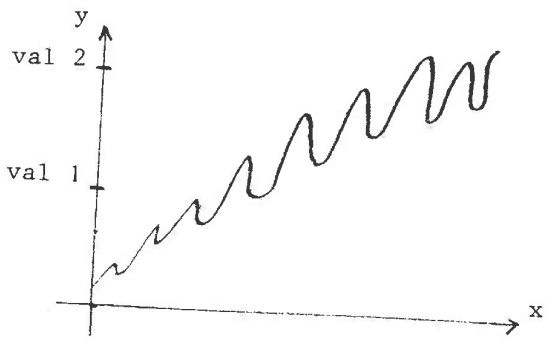
On pourrait écrire :

```
i, j entier contrôlable ;
quand i faire suspendre j ; j = i ; reprendre j fqd ;
quand j faire suspendre i ; i = j ; reprendre i fqd ;.....
```

6.673 Modification d'un contrôle.

On veut éditer les valeurs d'une fonction y = f (x) croissante en moyenne avec les contraintes suivantes :

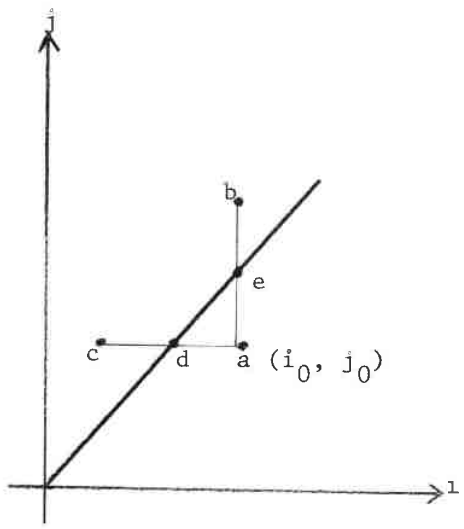
- x a un pas de 1 tant que l'on a pas rencontré de y val 1 ;
- x a un pas de 10 ensuite ;
- on s'arrête dès que n valeurs de y sont supérieures à val 2.



Cette application peut s'écrire :

```
module édition f (val 1, val 2, n) ; val 1, val 2 réel ;
pas, x réel ; y réel contrôlable ; i, n entier ;
    quand y >val 1 faire
            pas = 10 ; i = 0 ;
            quand y >val 2 faire i = i + 1 ;
                    si i = n alors sortir fsi fqd ;
    fqd ;
    pas = 1 ;
    pour x de 0 pas pas faire.
            y = f (x) ; imprimer y fp
    fin mod ;
```

6.674 Réalisation de contrainte



Etant donné deux variables i et j et une contrainte telle que i>j ; à partir d'un point a (i_0, j_0) la contrainte peut être rompue si j devient trop grand (tentative de rejoindre le point b) ou i trop petit (tentative d'atteindre c). Très souvent, le traitement d'anomalies est "borner la variable qui tente de déborder" : dans le cas où i devient trop petit, on fera i = j et dans l'autre j = i.

Il est extrêmement simple de programmer un tel problème.

```
i, j entier contrôlable ;
        quand i< j faire i = j fqd ;
        quand j >i faire j = i fqd ;
```

Cet exemple illustre le caractère dissymétrique des expressions de relation utilisées dans une instruction quand.

6.675 Remarque sur l'utilisation de "sortir".

Rappelons que l'instruction "sortir" permet de terminer l'exécution d'une procédure ou d'un module et de revenir à la procédure ou au module appelant.
La séquence de récupération d'une instruction "quand" n'est pas considérée comme une procédure. Une instruction "sortir" dans cette séquence fait donc sortir de la procédure ou du module contenant la déclaration de ce contrôle.

Exemple :

```
module x ;........ ;
        quand y faire sortir fq ; co l'affectation d'une valeur à
                                    y fera cesser l'exécution de x oc ;
```

```
quand z faire si t == 1 alors sortir
                        sinon imprimer t fsi
fqd ; co à chaque affectation de valeur à z, si t
        vaut 1, on sortira de x, sinon il y aura
        impression de t oc ;

fin mod
```

Nous reviendrons, au chapitre 10, sur l'utilisation des variables contrôlables pour l'aide à la mise au point des chaînes de traitement. Nous étudierons alors les variables contrôlables de la classe système.

REFERENCES DU CHAPITRE 6

BENAMGHAR L., 73 — Instruction d'affectation et définition d'un métalangage dans le projet Civa. Thèse Université Nancy 1. Juin 73.

BOLLIET

CHABRIER J. J., 73 — Acquisition des données et édition des résultats dans le projet Civa. Thèse Université Nancy 1. A paraître .

DUCLOY J., 73 — Compilation dans le projet Civa. Thèse Docteur-Ingénieur - Université Nancy 1. Mars 73.

GENUYS — Traduction du rapport Algol 60. Chiffres 3.1 Mars 1960.

VEILLON F., 72 — Cours de programmation en langage PL1. Armand Colin Ed. Paris 1972.

CHAPITRE 7

TRAITEMENT GLOBAL DES FILES ET DES ENSEMBLES

7.1. INTRODUCTION.

7.11. File et élément courant.
7.12. Ensemble d'emplacements.
7.13. Type file et type ensemble.
7.14. Opérations définies sur les files.
7.15. Opérations définies sur les ensembles.
7.16. Extension à un type des opérations définies pour l'autre.
7.17. Accès direct dans une file.

7.2. TRAITEMENT DES ENSEMBLES.

7.21. Déclaration d'un ensemble.
7.22. Réunion d'ensembles.
7.23. Suppression de valeurs d'un ensemble.
7.24. Opération de tri.
7.25. Extraction d'un sous-ensemble.
7.26. Affectation de valeurs à une variable de type ensemble.

7.3. TRAITEMENT DES FILES.

7.31. Déclaration d'une file.
7.32. Opérations élémentaires.
7.33. Opération de développement.
7.34. Opération de tri.
7.35. Désignation de file.
7.36. Expression de file.
7.37. Affectation de valeurs à une variable de type file.
1) Affectation de valeur à un élément d'une file.
2) Ecriture de l'affectation à une file.
3) File réceptrice de taille fixe.

4) File réceptrice de taille bornée.
5) File réceptrice de taille variable.
6) La file réceptrice est une sous-file.
7) La file réceptrice est une file de files.
8) Cas particulier d'une file de caractères.

7.4. ELEMENT COURANT D'UNE FILE.

7.41. Définition.
7.42. Déclaration.
7.43. Attribution du nom d'un emplacement de la file à l'élément courant.
1) Attribution explicite d'une valeur à l'élément courant d'une file.
2) Attribution d'un nom par qualification. Instruction "soit".
7.44. Rang et indices d'un élément courant.
1) Fonctions "rang" et "indice".
2) Rang et indices demandés à la déclaration d'un élément courant.
3) Fonctions "dernier", "premier", "sorti".

7.5. EXEMPLES D'UTILISATION.

7.6. CALCULS ITERATIFS ATTACHES A UNE FILE : INSTRUCTION "POUR CHAQUE".

7.61. Syntaxe.
7.62. Remarques.
7.63. Traitement d'une file simple.
1) Remarques.
2) Nombre d'itérations.
3) Exemples d'utilisation.
7.64. Traitement des files de dimension supérieure à 1.
1) Remarques.
2) Nombre d'itérations.
3) Exemples d'utilisation.

7.65. Traitement de plusieurs files.

1) Remarques.

2) Exemples.

7.66. Extension au cas des ensembles.

CHAPITRE 7

TRAITEMENT GLOBAL DES FILES ET DES ENSEMBLES

7.1. INTRODUCTION.

7.11. File et élément courant.

Une file est un ensemble d'emplacements totalement ordonné. Chaque emplacement (cf. 4.2.) "contient" une valeur. Une file est donc analogue à un tableau à une dimension ; cependant, file et tableau diffèrent quant à leur fonction d'accès.

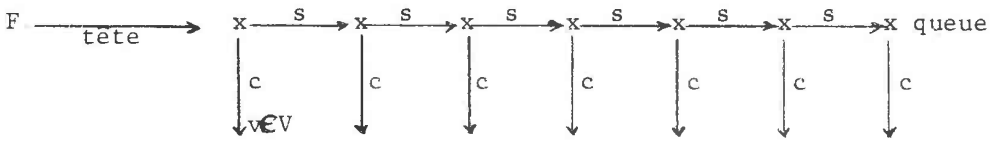
Un tableau à une dimension est une table (un ensemble I d'indicatifs ou "entrées dans la table", un ensemble V de valeurs et une fonction f de I dans V) (PAIR C., 71), dont l'ensemble des indicatifs est un intervalle de la suite des entiers positifs négatifs ou nuls. En général, on accède à un élément du tableau par une fonction d'adressage du type :

$$< A (i) > = < A (o) > + k (i - io)$$

où $< >$ signifie "adresse de", k est la taille d'un élément du tableau (elle est supposée constante), et io la borne inférieure de l'indice.

Une file est une liste au sens de (PAIR C., 71), c'est-à-dire un ensemble d'objets élémentaires contenant un ensemble P d'emplacements et un ensemble V de valeurs, et trois fonctions d'accès élémentaires :

- suivant : s, bijection de P - {dernier élément de P ou "queue"} dans P - {premier élément de P ou "tête"}.
  P est totalement ordonné par "suivant" : nous dirons que $\forall x, y \in P, x < y \Leftrightarrow \exists n > 1$ t.q $s^n (x) = y$.
- contenu : c, application de P dans V.
- tête : t, élément de P, fonction à 0 variable.



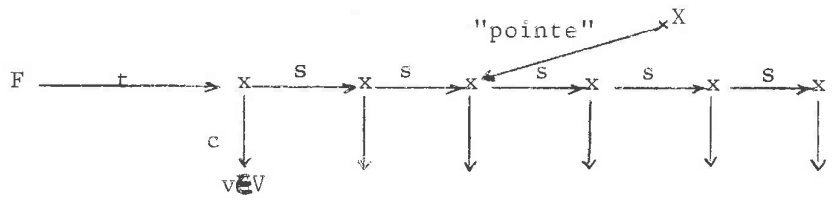
Pour désigner un élément d'un tableau à une dimension, on a l'habitude d'indiquer l'identificateur du tableau et le rang de l'élément à désigner : variable indicée de Fortran, Algol 60-68-PL1, etc... ou nom d'un fichier et clé de l'enregistrement dans les langages de traitement de fichiers, ces dispositifs favorisant l'accès direct. C'est d'ailleurs le seul accès utilisé dans les langages, que le mode de traitement soit séquentiel ou aléatoire.

Or les traitements séquentiels sont de très loin les plus fréquents - et favoriser les traitements séquentiels doit donc être un objectif impératif d'un langage de programmation efficace. D'autre part, si on cherche dans les modes d'expression courants en analyse, on y rencontre les notions de traitement global (traiter tout le fichier ou "pour chaque enregistrement faire..."), d'ordre des éléments (trier, ou "élément suivant"), mais on n'y rencontre que très peu celles de tableau et d'indice ou de rang d'un élément.

Il n'est donc pas très naturel, tout au long d'un traitement séquentiel, de devoir utiliser le rang ou la clé de l'élément à traiter. Nous sommes très tentés de donner un nom à l'élément courant, ce qui facilite la description du traitement, et de dire par exemple :

soit X l'élément courant de la file F
    initialiser X
    traiter X
    prendre un nouveau X
    recommencer.

On simplifie ainsi l'accès à un élément de la file :



où "pointe" (X) est une fonction à valeur dans P.

En reprenant la terminologie du chapitre 4, on peut dire que X est un identificateur qui "désigne" un objet O qui "contient" une valeur. Cette valeur n'est pas un objet du langage, mais le nom d'un emplacement de la file. Modifier la fonction "pointe", c'est-à-dire modifier le nom "pointé" par l'élément courant, peut être fait par l'intermédiaire d'instructions qui sont donc nécessairement différentes de l'instruction d'affectation. Elles seront vues en 7.43.

Le fait d'introduire un identificateur pour l'élément courant ne facilite pas seulement l'écriture, mais il évite également des calculs d'adresses assez longs pour accéder à un élément de la file : l'adresse d'un élément courant n'étant calculée qu'une fois, à la définition de la fonction "pointe", pour les utilisations ultérieures le calcul d'adresse est remplacé par un adressage indirect. De plus, l'introduction d'un accès séquentiel, c'est-à-dire la possibilité de définir la nouvelle valeur de "pointe", donc un nouveau nom d'emplacement, à partir de la valeur précédente et de la fonction s rend plus rapide ce calcul lui-même (PECCOUD D., 72).

Nous introduisons donc cette notion d'élément courant (1) et des instructions permettant de l'utiliser, pour permettre un <u>accès séquentiel</u> aux éléments d'une file, en plus de <u>l'accès direct</u>, qui est toujours possible puisqu'on peut accéder à chaque élément de la file par son rang comme dans un tableau à une dimension.

(1) Une file munie d'un élément courant est donc assez semblable à une file munie d'un index de langage A.T.F. (NOLIN L., 69 et 71). Cependant, ici l'accès à un élément ne se fait pas nécessairement par son index, ce qui assouplit un peu l'utilisation, et nous utiliserons des instructions particulières pour faire évoluer un élément courant, ce qui évitera toute ambiguïté. D'autre part, il nous semble qu'un index, c'est-à-dire le rang d'un élément, est une notion technologique, liée au rangement des objets dans la file, ce qui n'est pas le cas d'un identificateur d'élément courant.

7.12. Ensembles.

Au cours de l'analyse, et tant qu'il s'exprime en mode déclaratif, l'utilisateur décrit souvent des traitements à entreprendre sur des ensembles de valeurs indépendamment de tout ordre sur elles. Ainsi, lorsqu'il demande une mise à jour d'un fichier "stock" depuis un fichier "mouvement", il demande que, pour chaque article, la quantité en stock soit modifiée en tenant compte des mouvements de cet article décrits dans le fichier "mouvement".

Cependant, la description du travail à effectuer (qui peut être faite plus tard) fait intervenir un ordre de présentation des articles dans les deux fichiers : par exemple, le fichier "stock" et le fichier "mouvement" seront classés par ordre croissant des numéros de produits. Cet ordre ne définit pas un ordre total sur les articles de manière unique : par exemple un même produit peut figurer plusieurs fois dans le fichier "mouvement". Il définit plutôt un préordre total, c'est-à-dire une relation reflexive, TRANSITIVE ~~antisymétrique~~ telle que deux éléments soient toujours comparables.

On est donc conduit à considérer des ensembles de valeurs munis d'un préordre total, par exemple un ensemble d'articles muni du préordre $\leqslant$ défini par :

article i $\leqslant$ article j $\Longleftrightarrow$ n° de produit de l'article i $\leqslant$ n° de produit de l'article j

(dans la suite, nous désignons toujours un préordre par $\leqslant$).

En réalité, nous considérons plutôt des ensembles d'emplacements contenant de telles valeurs, ce qui permet d'y placer plusieurs exemplaires de la même valeur. Un préordre dans l'ensemble des valeurs susceptibles d'être contenues dans ces emplacements induit un préordre sur les emplacements : si x, y sont des emplacements :

$$n \leqslant y \Longleftrightarrow c(x) \leqslant c(y)$$

Un ensemble d'emplacements ainsi muni d'un préordre total (nous dirons seulement un ensemble) est différent d'une file :

- d'abord, il est préordonné totalement et donc pas nécessairement ordonné totalement (si on préfère, plusieurs ordres totaux peuvent être compatibles avec le préordre donné) ;

- ensuite, ce qui précède est défini par un préordre sur l'ensemble V de toutes les valeurs susceptibles d'être

contenues dans les emplacements. Cela rend possible la définition d'opérations comme l'adjonction d'un élément, la réunion de deux ensembles (ayant le même ensemble V), autrement dit la fusion ou l'interclassement, l'affectation d'un nouveau préordre (sur les valeurs), autrement dit le tri ; pour une file, au contraire, on ne pourrait, par exemple, pas définir l'adjonction d'un élément sans indiquer où le placer (et même, pour des raisons d'implémentation, on interdira les adjonctions) ; mais on peut définir, par exemple, la concaténation de deux files.

La notion d'ensemble au sens qui lui est donné ici n'est pas envisagée par les langages de programmation classiques et le problème de la représentation d'un ensemble de valeurs muni d'un préordre total par une suite d'emplacements est à la charge de l'utilisateur et sa solution est connue de lui seul.

Ainsi, c'est l'utilisateur qui impose l'ordre de rangement des valeurs dans une file en attribuant successivement une valeur à chacun des emplacements ou en imposant des tris qu'il décrira explicitement. C'est lui qui devra savoir, lorsqu'il voudra ajouter une valeur (un article) à son fichier, à quel endroit le placer pour respecter éventuellement son ordre de présentation.

Un <u>ensemble</u> est un ensemble d'emplacements susceptibles de contenir des valeurs d'un type donné t, muni d'un préordre total $P1^{\leqslant}$, défini par un préordre total $P2^{\leqslant}$ sur l'ensemble des valeurs du type t :

$$x \; \overset{\leqslant}{P1} \; y \iff c(x) \; \overset{\leqslant}{P2} \; c(y)$$

Dans l'exemple du fichier "stock" et du fichier "mouvement" cités ci-dessus, on peut définir un préordre sur l'ensemble des valeurs des articles à partir des numéros de produits, par exemple de la manière suivante :

article i $\leqslant$ article j $\iff$ n° produit de l'article i $\leqslant$ n° produit de l'article j

Pour les articles du fichier "stock", il s'agit d'un ordre total, mais pour ceux du fichier "mouvement", il s'agit bien, en général, d'un préordre total seulement, deux articles "mouvement" pouvant concerner le même produit.

L'ensemble "mouvement" est l'ensemble des enregistrements muni du préordre total défini par le préordre total sur les articles qu'ils contiennent.

Le fichier "mouvement", sous sa forme habituelle, est une représentation de cet ensemble : si c'est un fichier séquentiel, l'ordre total de ses emplacements est compatible avec le préordre total défini sur les enregistrements :

x précéde y $\Longrightarrow$ article x $\leqslant$ article y

$\Longrightarrow$ n° produit de x $\leqslant$ n° produit de y.

On peut convertir un ensemble en une file dont l'ordre est compatible avec le préordre de l'ensemble, c'est-à-dire :

$y = s(x) \Rightarrow x \leqslant y \quad (\Leftrightarrow c(x) \leqslant c(y))$.

Il y a en général plusieurs solutions possibles et celle qui est choisie est inconnue du programmeur. Une telle file est appelée file support de l'ensemble.

Inversement, on peut convertir une file en un ensemble, qui est son ensemble d'emplacements avec le préordre "plein" sur les valeurs, c'est-à-dire tel que, pour deux valeurs quelconques x et y, on ait :

$x \leqslant y$ (et $y \leqslant x$).

7.13. Type file et type ensemble.

Le type "file" est donc caractérisé par le type des valeurs que peuvent contenir ses emplacements et par le nombre de ces emplacements. Le type "file d'éléments de type t" peut être défini par $\{t\}^n$ dont les éléments sont des files au sens de 7.11., et une variable de type file peut donc contenir des valeurs de $\{t\}^n$; n est précisé par la taille de la file (cf. 4.24. et 7.31.).

Le type "ensemble d'éléments de type t" peut être défini par

$\{S\} \times \{P\}$ où $S \subset t$ et où les valeurs de P indiquent des préordres possibles.

$\{P\} = \{OE\}^m \cup \{sens\}$, où l'ensemble OE des ordres (1) élémentaires est formé de couples constitués d'un identificateur (celui d'un champ de la structure t quand t est une structure) et d'un

(1) Nous continuerons à parler d'ordre, pour ordre de rangement, bien qu'il ne s'agisse que de préordre (relation d'ordre au sens large).

sens et Sens est l'ensemble des "sens" : il est formé des deux valeurs "croissant" ou "décroissant".

(date, croissant), (n° client, croissant), (n° produit, décroissant)

est un ordre (une valeur) qui pourrait être utilisée pour l'ensemble des commandes d'une entreprise.
Chaque valeur élémentaire définit un préordre total sur t. La conjonction de ces préordres est également un préordre total sur t. Une variable de type ensemble contient donc des valeurs $\{S\} \times \{P\}$.

7.14. Opérations définies sur les files.

Nous nous intéressons ici aux opérations permettant un traitement global des files, c'est-à-dire celles pouvant être exécutées sur une file dans sa totalité par opposition aux opérations qui n'en concernent qu'un élément.

Le traitement global d'une file est toujours un traitement séquentiel : il porte sur une suite d'emplacements. Décrire un tel traitement consiste à associer à une file un ou plusieurs traitements qu'il faudra entreprendre pour chacun de ses éléments, ce qui amène à répondre à trois questions :

- comment désigner la file à traiter ?
- comment, dans le traitement, désigner l'élément en cours de traitement et comment passer d'un élément à un autre ?
- comment décrire cette association d'un ou plusieurs traitements aux éléments désignés.

Nous aborderons la désignation de la file à traiter (cf. 7.35.) en étudiant les opérations qui permettent de passer d'une expression de file quelconque à une file de dimension 1 de borne inférieure 1 :

- passage à une sous-file caractérisée par les rangs de ses extrêmités
- concaténation de files
- développement, qui est une opération faisant passer d'une file de dimension 2 (une file de files) ou d'une file de dimension 1 mais de borne inférieure différente de 1 à une file de dimension 1 de borne inférieure 1.

Le traitement à associer à chacun des éléments d'une file peut être particulièrement simple, il peut s'agir uniquement d'un transfert dans une file réceptrice : une instruction d'affectation

entre files permet alors de décrire complètement le traitement. Ainsi, l'instruction d'affectation

A = B

permet d'attribuer une valeur à la variable A. Si A et B sont du type file, l'identificateur B "désigne" une variable qui "contient" (cf. 1.1.) une valeur : une file, c'est une constante. L'instruction d'affectation modifie le "contenu" de A : A va contenir la même file que B.

La file désignée à droite du signe = peut l'être par une expression de file :

A = B (1 : 15), C, B (16 : .)

permet de demander le transfert des valeurs contenues dans la file obtenue par concatenation de la sous-file de B constituée de ses 15 premiers éléments, de la file C et des autres éléments de B. Le paragraphe 7.37. décrit l'instruction d'affectation appliquée à un objet de type file.

Nous précisons ensuite la notion "d'élément courant" utilisée pour "pointer" un élément d'une file, et les instructions qui permettent de fixer le nom de l'emplacement qu'il "pointe" ou de le modifier. Ces instructions permettent de contrôler très exactement les calculs itératifs en détaillant leur description (cf. 7.4.).

Il pourra être utile de connaître le rang de l'emplacement "pointé" par "l'élément courant": l'opération "rang" peut être définie pour les files puisque les emplacements sont totalement ordonnés.

L'instruction "pour chaque" permet de décrire de façon plus concise, dans le mode déclaratif (cf. 2.5.), le traitement global d'une file. Elle est, en quelque sorte, une écriture condensée d'une séquence d'instruction traitant un "élément courant". Son utilisation est rendue la plus souple possible, en l'étendant aux files définies par une expression de file quelconque (cf. 7.6.).

7.15. Opérations définies sur les ensembles.

Ce sont les opérations de réunion, suppression, passage à un sous-ensemble, tri et affectation de valeurs.

La réunion fait passer de deux ensembles à leur réunion munie du préordre associé au premier ensemble. Un cas particulier de cette réunion est celui de l'adjonction d'éléments à un ensemble, les

valeurs à ajouter étant fournies explicitement.

La suppression modifie l'ensemble des emplacements d'un ensemble et conserve le préordre associé, les emplacements à supprimer sont cités ou caractérisés par une propriété de leurs valeurs.

Ces opérations modifient l'ensemble des valeurs contenues par un ensemble, par contre l'opération de tri modifie le préordre associé à un ensemble sans en modifier les valeurs. Ainsi, A étant un ensemble d'éléments du type entier :

A <u>par ordre croissant</u>

est une opération qui fait passer de A à un ensemble contenant les mêmes valeurs que A (mais elles sont rangées par ordre croissant) et dont l'indicateur d'ordre contient la valeur "croissant".

Si B est un ensemble d'objets de type structure dont C, D, E sont des champs

B <u>par</u> C <u>croissant</u>, D, E <u>décroissant</u>

est une opération qui trie B selon les critères indiqués pour faire passer à un ensemble contenant les mêmes valeurs que B et dont l'indicateur d'ordre contient la valeur (C, croissant), (D, décroissant), (E, décroissant).

L'instruction d'affectation permet d'attribuer une valeur à un ensemble :

A = B

si A et B sont du type ensemble, A désigne une variable qui contiendra le même ensemble (constante) que B.

7.16. <u>Extensions à un type des opérations définies pour l'autre.</u>

Syntaxiquement, certaines opérations définies pour un type seront autorisées pour l'autre grâce à des conversions.

L'exemple le plus simple est celui de l'affectation. Si A est une file et B un ensemble les opérations A = B et B = A sont permises :

- A = B : il s'agit d'une affectation de valeur à une file. B est alors converti en une file qui est sa file support ;
- B = A : il s'agit d'une affectation de valeur à un ensemble. A est converti en son ensemble équivalent qui est l'ensemble des valeurs contenues dans A muni du

préordre "plein". (1).

De même une opération de tri pourra être appliquée à une file :

A <u>par ordre croissant</u>

fait d'abord passer de la file A à son ensemble équivalent puis à l'ensemble qui en est déduit par l'opération de tri. Ce qui permet de demander le tri d'une file en écrivant :

A = A <u>par ordre croissant</u>

A est convertie en son ensemble équivalent ; celui-ci est trié puis converti en une file avant d'effectuer l'affectation.

Nous étendrons également la définition de l'instruction "pour chaque" au cas des ensembles : l'élément courant "pointe" un emplacement de la file support.

La conversion d'un ensemble en une file conduit à plusieurs résultats possibles : nous avons vu, en 7.12., qu'un ensemble peut être représenté par une file dont l'ordre est compatible avec le préordre de l'ensemble ; les solutions diffèrent pour les éléments tels que $x \leqslant y$ et $y \leqslant x$ où $\leqslant$ est le préordre de l'ensemble.

7.17. <u>Accès direct dans une file</u>.

Le volume important de ce chapitre consacré aux traitements séquentiels ne doit pas faire oublier qu'ils ne sont pas les seuls possibles : l'accès direct est toujours possible. Il peut donc être utilisé, si cela est nécessaire, sous la forme d'un identificateur d'élément de la file. Considérons une file d'éléments simples. Chacun de ses emplacements peut être numéroté par un élément d'un intervalle de l'ensemble des entiers. Un identificateur d'élément de file est un couple : identicateur de file, désignation d'un numéro d'emplacement. Cette désignation est une expression arithmétique quelconque (cas d'une variable indicée classique) ou une notation de chaîne de caractères, c'est alors une <u>clé</u>. La correspondance entre les clés et les numéros d'emplacement est établie lors de

(1) Après la suite d'instructions A = B ; B = A l'ensemble B ne contient plus la même valeur.

la donnée des valeurs contenues dans la file (en donnant les clés associées).
F ('clé 1') désigne donc l'élément de la file F correspondant à la clé 'clé 1'.

Si la file F est une file d'éléments de type structure dont S est un champ, l'écriture S de F (I) pourra être condensée en S (I) (DUCLOY J., 73, paragraphe 4.2.).

7.2. TRAITEMENT DES ENSEMBLES.

Nous avons vu en 7.15. que nous définissons sur les ensembles les opérations de réunion, suppression, passage à un sous-ensemble, tri et affectation des valeurs. Il nous reste donc à préciser ces opérations et définir leurs aspects syntaxiques.

7.21. Déclaration d'un ensemble.

< déclaration d'ensemble > :: = < identificateur> ensemble < type >
< taille> :: = (max expression arithmétique>)|< expression arithmétique>

< type> a été défini en 4.22. : il indique le type des éléments de l'ensemble. Nous ne définissons que des ensembles de dimension 1. Nous remarquerons également que la déclaration d'un ensemble ne précise pas sa taille : un ensemble est toujours de taille variable : sa taille peut varier de façon continue par des adjonctions ou suppressions d'éléments.

Ceci constitue une autre raison pour laquelle nous avons distingué syntaxiquement les deux types d'objets file et ensemble : de nombreux traitements ne font pas intervenir l'ordre de présentation des valeurs d'une file mais s'accommodent de files de taille fixe (de réalisation plus simple) ou de taille bornée sans nécessiter de file de taille variable, ou encore sont plus commodes à exprimer en introduisant des files de dimension 2.

Nous définissons une fonction "taille actuelle" (e) dont la valeur est celle du nombre d'éléments de l'ensemble e.

7.22. Réunion d'ensembles.

L'opérateur de réunion est u.

Le premier opérande est un ensemble ; si les éléments du premier opérande sont de type t, le second opérande est un ensemble d'objets de type t, ou un objet de type t.

Le résultat est un ensemble : c'est la réunion, au sens classique, de deux opérandes, munie du préordre associé au premier opérande.

Les deux ensembles A u B et B u A peuvent donc être différents.

Des opérations de conversion permettent d'étendre la définition de cette opération : le second opérande peut être un élément ou un ensemble d'éléments d'un type compatible avec t, ou encore il peut être une file d'éléments d'un type compatible avec t.

Lorsque cela est nécessaire, le second opérande est converti en un ensemble muni du préordre associé au premier opérande (il y a éventuellement un tri), et l'opération de réunion est toujours effectuée sur deux ensembles munis du même préordre : elle est donc effectuée comme un interclassement.

Ainsi, si A et B sont des ensembles d'entiers munis du préordre $\leqslant$ (croissant), les opérations de réunions suivantes peuvent être écrites :

A u B : la taille du résultat est la somme de celle de A et de celle de B.
Le préordre associé au résultat est celui de A. Les éléments sont ceux de A et B.

A u 15 : est un ensemble ayant un élément de plus que A et le même préordre que A.

A u (15, D, 17.3.) :
il s'agit de la réunion à l'ensemble A, d'une liste de valeurs ; D est supposé entier ; 17.3. sera d'abord converti en entier.

A u F : où F est une file, est autorisée si les éléments de F sont d'un type compatible avec le type entier.

7.23. Suppression de valeurs d'un ensemble.

L'opération de suppression fait passer d'un ensemble à un autre muni du même préordre, mais privé des éléments à supprimer. Elle est demandée par une instruction "supprimer" qui s'écrit :

supprimer <valeur à retirer> [dans <identificateur>] .

<valeur à retirer> :: = <identificateur d'élément courant> tel que < souche >

| <valeur> [, <valeur>]*.

<souche> :: = <identificateur de condition> | <expression booléenne>

(cf. chapitre 5)

<valeur> :: = <expression arithmétique> | < expression booléenne>

| <chaîne de caractères> .

Les valeurs à retirer peuvent être caractérisées par :

- une propriété : c'est la première altérnative de la règle, comme par exemple dans

  supprimer X tel que A>35 et A<42 dans F ;

  qui demande la suppression de F de toutes les valeurs telles que le champ A de X est compris entre 35 et 42 ; si X a été déclaré comme un élément courant attaché à F l'instruction peut être réduite à :

  supprimer X tel que A>35 et A<42 ;

- une liste de valeurs : chacune d'elles est décrite par une expression arithmétique (resp. booléenne) si les éléments de l'ensemble F sont de type arithmétique (resp. booléen), ou une file de caractères si ces éléments sont des files de caractères.

La taille de l'ensemble diminue du nombre de valeurs retirées. L'ordre de rangement est conservé : il y a un tassement de la file support sur elle-même.

L'opération de suppression de valeurs peut aisèment être étendue au cas d'une file : celle-ci est convertie en un ensemble, avec le préordre "plein", la suppression est effectuée, l'ensemble résultat est reconverti en file.

7.24. Opération de tri.

L'opération de tri est une opération unaire ; elle fait passer d'un ensemble à un autre contenant les mêmes valeurs mais possédant un préordre différent. Elle s'écrit en faisant suivre un identificateur d'ensemble d'une option d'ordre, qui précise le préordre à associer au résultat.

<option d'ordre> :: = par ordre <sens> par <critère> [, <critère>]* <sens>]+.

<sens> :: = croi | décroi

<critère> :: = <identificateur> .

L'utilisation d'une option d'ordre n'a de sens que si la variable de type "ensemble" désignée par l'identificateur précédant cette option contient déjà des valeurs et s'il est utilisé en "entrée", c'est-à-dire s'il est utilisé en position "d'emetteur" dans les instructions d'affectations, à droite du signe =. Si les éléments de l'ensemble sont d'un type structuré, il suffit que les champs servant de critères vérifient cette condition.

Si l'ensemble désigné a des éléments de type simple (entier, réel, décimal, booléen, caractère) ou du type file de caractères, il n'est pas besoin de critère de tri. L'option d'ordre s'écrit alors :

par ordre croi[ssant] ou par ordre décroi[ssant]

Un critère de tri est utilisé lorsque les éléments de l'ensemble sont des structures. Un critère est repéré par l'identificateur du champ de la structure sur lequel porte le classement à opérer. L'identificateur de champ peut être qualifié par l'un de ses prédecesseurs dans la structure : A de B, par exemple, si A désigne un objet composant de B.

S'il figure plusieurs critères, le préordre ainsi défini est la conjonction des préordres définis par chaque critère : le premier rencontré dans l'option d'ordre est le critère majeur, il y a d'abord classement selon ce critère, les éléments non ordonnés par ce critère sont classés selon le critère suivant, etc...

Un "sens", croissant ou décroissant, est associé à tous les critères qui sont écrits devant lui dans une option d'ordre, depuis par ou le "sens" précédent.

Exemple :

par année, mois, jour croi nom de client décroi

est une option d'ordre qui peut être utilisée pour demander le tri

d'un ensemble de commandes par exemple par ordre de date croissante ; les commandes émises à la même date seront classées par ordre alphanumérique décroissant des noms de clients.

Comme nous l'avons vu en 7.16., une opération de tri pourra être appliquée à une file : elle fait passer d'une file à un ensemble dont les valeurs sont celles de la file et le préordre associé est celui donné par l'option d'ordre.

7.25. Extraction d'un sous-ensemble.

Le passage d'un ensemble à un sous-ensemble est une opération qui fait passer d'un ensemble à un autre dont les valeurs sont celles du premier ensemble qui vérifient une propriété. Le sous-ensemble est muni du même préordre que l'ensemble dont il est issu.

L'opération d'extraction s'écrit telque. Il est suivi de la propriété que vérifient les éléments du sous-ensemble extrait : elle est décrite par une expression booléenne, dans laquelle peuvent intervenir les identificateurs des champs de la structure t ou l'identificateur "élément" qui désigne l'élément de l'ensemble.

Si A est un ensemble d'entiers muni du préordre $\leqslant$:

A telque élément $<$ 2 000

est le sous-ensemble extrait de A, limité aux éléments inférieurs à 2 000.

Si B est un ensemble d'éléments du type structuré dont numéro est un champ :

B telque numéro $\geqslant$ 2 500

définit un sous-ensemble de B.

La propriété que doivent vérifier les éléments du sous-ensemble peut également figurer dans la file support entre deux positions limites : borne inférieure et borne supérieure.

7.26. Affectation de valeur à une variable de type ensemble.

L'instruction d'affectation permet d'attribuer une valeur à un ensemble. Elle s'écrit :

```
< identificateur d'ensemble> = < désignation d'ensemble>
                                  [ < option d'ordre>]
<désignation d'ensemble> :: = <ensemble> u <désignation
                                  d'ensemble> | <ensemble> .
<ensemble> :: = < identificateur> | (< liste de valeurs>) |
                < exp. arithmétique> | <sous-ensemble>
<sous-ensemble> :: = <identificateur>( [<borne inférieure>,]
                     < borne supérieure>] |<identificateur>
                      telque <expression booléenne>
<borne inférieure> :: = <exp. arithmétique> | .
<borne supérieure> :: = <exp. arithmétique> | .
<liste de valeurs>:: = <exp. arithmétique>[, < exp.
                                  arithmétique>]*
```

L'ensemble récepteur (désigné par l'identificateur écrit à gauche de -) reçoit la valeur de l'ensemble émetteur : valeurs de ses éléments et la description de son préordre.

Si l'option d'ordre est absente, le préordre associé au récepteur est le préordre "plein".

Si la désignation d'ensemble commence par un identificateur, c'est celui d'un ensemble dont le préordre sera celui de l'ensemble récepteur.

Si cet identificateur est le même que celui de l'ensemble récepteur nous dirons qu'il s'agit d'une modification de l'ensemble. Le traitement d'une modification sera en général plus simple que celui de l'affectation dans le cas général.
En particulier l'instruction

E = E <u>par</u> prix <u>croissant</u> ;

consistera uniquement en un tri de l'ensemble E, sauf si le préordre associé à E était déjà défini par : (prix, croissant).

La définition de l'affectation de valeur à un ensemble pourra être étendue au cas où l'émetteur est une file (plus généralement une désignation de file).

La file est alors convertie en un ensemble (préordre "plein").

7.3. TRAITEMENT DES FILES.

Nous avons vu en 7.14. que nous définissions les opérations de passage à une sous-file concaténation de files, développement, affectation, prise d'élément courant et "pour chaque".

7.31. Déclaration d'une file.

Elle peut être plus complexe que celle d'un ensemble à cause des files de dimension supérieure à 1 et des différents types de files.

Une file dont les éléments sont des files d'éléments simples ou structurés sera dite file de dimension 2. Une file de files de dimension 2 est dite de dimension 3 ; etc...

Une déclaration de file s'écrit (cf. 4.22.) :

< identificateur > < type file >

< type file > :: = file <taille> < type >

< taille > :: = (max < expression arithmétique >) | (< paire de limites > [, < paire de limites >] *)

< paire de limites > :: = < exp. arithmétique > : < expression arithmétique > .

A file type t pourra s'écrire plus simplement A file t.

A file entier est la déclaration d'une variable de type file de taille variable d'entiers. Sa taille pourra être modifiée par des instructions d'affectation à une file. Une fonction "taille actuelle" permet de connaître le nombre d'éléments d'une file : taille actuelle (f) a pour valeur le nombre d'éléments de la file f.

Si <taille>commence par max, la variable déclarée est du type "file de taille bornée" sinon c'est une file de taille fixe (cf. 4.24.). Le nombre de paires de limites indique alors la dimension de la file. L'expression arithmétique définissant une limite inférieure doit être une constante (calculable à la compilation). L'expression définissant la limite supérieure doit être calculable au moment de l'élaboration de la déclaration de file.

7.32. Opérations élémentaires.

Une opération permet de passer d'une file à une sous-file dont les éléments sont ceux de la file compris entre deux bornes.

Cette opération s'écrit en faisant suivre l'identificateur de file de la paire de bornes définissant la sous-file, écrites entre paranthèses.

On peut étendre cette définition à une file de dimension supérieure à 1 :

F (15 : 25), G (1 : 10,1 : 5) définissant des sous-files.

La syntaxe de cette définition de file sera vue au paragraphe "désignation de file" en 7.

Une opération pas permet de passer d'une file à une autre dont les emplacements sont ceux de la première pris à intervalles réguliers dont la valeur est définie par le pas de la progression.

L'ensemble des files est muni d'une loi de composition interne la concaténation, notée ",". C'est une opération qui fait passer d'une file A et d'une file B à une file C obtenue en plaçant à la suite des emplacements de A, ceux de B. Elle n'est définie que pour des files d'éléments de même type.

Nous appelons file attachée à une désignation de file, la file obtenue après avoir effectué les opérations de "pas", de passage à une sous-file et de concaténation demandées par cette désignation.

7.33. Opération de développement.

Un traitement séquentiel porte toujours sur une file de dimension 1 et de borne inférieure 1. Une opération de développement fait passer d'une file attachée à une désignation à une file de dimension 1 et de borne inférieure 1, appelée file développée de la désignation.

Si une désignation de file porte sur une file de dimension 1, de borne inférieure 1, file attachée et file développée sont identiques.

Le développement d'une file de dimension 1 mais de borne inférieure différente de 1 consiste à en déduire une file développée (1), dans laquelle les éléments sont rangés dans le même ordre que dans la file attachée.

(1) Cette opération ne porte que sur les descripteurs de file, les files ne sont, en général, pas déplacées en mémoire.

Le développement d'une file de dimension supérieure à 1, consiste à en déduire une file (1) développée, contenant les mêmes valeurs, rangées dans le même ordre que dans la file attachée. Signalons que les files de dimension 2 (file de files) sont rangées ligne par ligne, et qu'en général, lors du rangement d'une file de dimension supérieure à 1, c'est le dernier indice qui varie le plus vite. Ainsi la file de files, attachée à la désignation F (2 : 3, 1 : 2) sera développée en une file de dimension 1 dont les éléments seront dans l'ordre :

F (2,1), F (2,2), F (3,1), F (3,2).

Les indices d'un élément seront toujours définis dans sa file attachée. Ainsi les indices du premier élément de la file attachée à F (2 : 3, 1 : 2) sont 2 et 1.

7.34. Opération de tri.

Nous avons défini en 7.24. une opération de tri faisant passer d'un ensemble à un autre. Syntaxiquement, elle a été étendue pour être applicable à une file qui est d'abord convertie en ensemble. Si la position de la désignation de file exige que l'on obtienne une file, l'ensemble résultant du tri sera converti en file.

Nous appelons file associée à une désignation la file résultant de cette conversion.

Nous appellerons file désignée par une désignation de file, une file ayant les mêmes valeurs que la file associée mais dont la dimension et les bornes sont celles de la file attachée.

7.35. Désignation de file.

Les traitements portent en général sur des files déclarées et il suffira, dansune instruction, de mentionner le nom de la file. Deux cas font exception à cette règle : lorsque le traitement envisagé ne concerne pas toute la file - il convient alors de désigner une sous-file d'une file déjà déclarée - ou lorsque le traitement envisagé concerne une file créée par un tri ou par concaténation d'objets (éléments simples, files ou sous-files) déjà déclarés ou de constantes :

<désignation de file> :: = <expression de file> [<option d'ordre> .]

<expression de file> :: = <désignation élémentaire> [, <désignation élémentaire>]*

<désignation élémentaire> :: = <identificateur> [<partie dimension>] | <constante>.

<partie dimension> :: = ([<paire de bornes>] [option pas] [, [<paire de bornes> [<option pas>]]*) | <option pas>

<option pas> :: = pas <expression arithmétique> .

<paire de bornes> :: = <borne> : <borne> .

<borne> :: = . | <expression arithmétique> .

<option d'ordre> (voir paragraphe 7.24.).

Remarquons que dans une désignation de file, les deux bornes d'une sous-file doivent figurer ou être remplacées par un point.

La sémantique d'une désignation de file a été précisée au cours des paragraphes précédents en définissant des opérations sur les files.

7.36. Expression de file.

La file attachée à une désignation est celle obtenue à partir de la (ou les) file (s) nommée (s) dans la désignation, en ne prenant que les éléments désignés par celle-ci (cf. 7.32.).

Lorsque la file attachée à une désignation est une sous-file d'une file existante, ses éléments sont caractérisés par la partie dimension, donc par les bornes de la sous-file. Les bornes permettent d'indiquer, pour chaque indice, son domaine de variation. Si, à la place correspondant à un indice dans la partie dimension, il n'y a pas de paire de bornes, ce sont les bornes effectives qui seront prises en compte pour cet indice (c'est-à-dire les bornes actuelles pour une file de taille variable ou bornée, et les bornes données à sa déclaration pour une file de taille fixe).

La notation . pour une borne indique que l'on prend pour celle-ci la borne effective de la file.

Si une borne est représentée par une expression arithmétique, celle-ci doit être calculable lors de l'exécution de l'instruction contenant la désignation de file. Elle est calculée une seule fois

pour toute l'exécution de l'instruction. La valeur obtenue est convertie en entier si cela est nécessaire.

Ainsi, si F et G sont des files déclarées par :

F (25) caract ; G (15 : 15) entier ; H file (25) entier ;

- la file attachée à la désignation F ou F ( ) ou F (.:.) est la file elle-même,
- pour F (.: 10), c'est la file constituée des 10 premiers éléments de F,
- pour G (.: 10, 2 : 7), c'est la file de 60 éléments désignés par les 10 premières files de 6 entiers de G, chacun d'eux étant pris au rang correspondant dans G.
- pour H (10 : .) c'est la file constituée des 15 dernières valeurs de H.

La partie dimension peut être absente dans une désignation de file : il s'agit alors de la file dans son entier. Un identificateur, sans partie dimension, peut également désigner un objet de type simple.

Lorsqu'elle est présente, le nombre de virgules dans la partie dimension, augmenté de 1, indique la dimension de la file attachée à la désignation. Ce n'est pas nécessairement la plus grande dimension associée à une file.

Ainsi, F étant une file de files de files de caractères :

- les éléments de la file attachée à F (,) sont des files de caractères ;
- ceux de la file attachée à F (,,) sont des caractères ;
- ceux de la file attachée à F sont des files de files de caractères.

Pour l'exemple précédent, G (. : 10) désigne un sous-ensemble de G, constitué des 10 premiers ensembles de 15 entiers qui constituent G, tandis que G (. : 10,) désigne un sous-ensemble de G constitué d'entiers.

L'option pas associée à un indice indique quelle progression suivie pour cet indice en prenant les éléments de la file désignée pour constituer la file attachée.

Un traitement peut porter sur une suite de files et de sous-files. Pour pouvoir les décrire dans une même désignation, on utilise la concaténation (cf. 7.32.).

Exemples :

- F (1 : 15)

  Les files attachée, développée, associée et désignée sont identiques.

- F (10 : 15)

  La file attachée a 6 éléments de rang 1 à 6, d'indice 10 à 15. La file développée a les mêmes éléments, dans le même ordre, de rang et d'indice de 1 à 6. Les files associées et désignées lui sont identiques.

- F (1 : 3, 1 : 5)

  La file attachée est une file de dimension 2, possédant 3 files de 5 éléments. La file développée est une file de dimension 1, possédant 15 éléments, obtenue de la précédente sans déplacement. La file associée à cette désignation est la file développée (elle est utilisée pour les traitements séquentiels). La file désignée est la file attachée (elle est utilisée dans les accès directs).

- F (1 : 3, 1 : 5) par ordre croissant

  File attachée :

1	4	7	10	13
2	5	8	11	14
3	6	9	12	15

  File développée :

  1 4 7 10 13 2 5 8 11 14 3 6 9 12 15

  Ces deux files sont les mêmes que dans l'exemple précédent.

  File associée :

  1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

  File désignée :

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15

- F (4 : 8) par ordre croissant

  File attachée :

12	5	3	17	7
4	5	6	7	8

  File développée:

12	5	3	17	7
1	2	3	4	5

File associée :	3	5	7	12	17
	1	2	3	4	5
File désignée :	3	5	7	12	17
	4	5	6	7	8

Lorsque la désignation de file figure dans une déclaration d'élément courant, nous dirons par extension, que les files attachée à, développée de, associée à, la file nommée par la désignation sont attachée à, développée de, associée à, l'élément courant.

7.37. Affectation de valeurs à une variable de type file.

7.371. Affectation de valeur à un élément d'une file.

Un élément d'une file peut être désigné par sa position dans la file : son indice. Si la file est de dimension n, il peut être désigné par ses indices. L'affectation de valeur à 1 élément d'une file fait passer d'une file à une autre qui ne diffère de la première que pour cet élément.
L'élément pourra également être désigné par un élément courant (cf. 7.14. et 7.4.).
L'instruction pourra alors être étendue au cas de l'affectation de valeur à un élément d'un ensemble, si l'élément courant attaché à cet ensemble (c'est-à-dire attaché à sa file support) désigne un emplacement de cet ensemble qualifié par une propriété, et si le type des éléments de l'ensemble est une structure.
L'affectation peut alors porter sur les champs de la structure qui n'interviennent pas pour définir le préordre de l'ensemble.

7.372. Ecriture de l'affectation à une file.

L'instruction d'affectation de valeur à une file ou une sous-file s'écrit :

<affectation à une file> :: = <file ou sous-file> =
<désignation de file> .

<file ou sous-file> :: = <identificateur> | <identificateur>
<partie dimension> .

Remarques :

1) Une instruction d'affectation à une file comporte donc, à gauche du signe "égale", un identificateur de file ou d'une sous-file que nous appellerons file réceptrice, et, à droite de ce signe, une désignation de file, telle qu'elle a été décrite en 7.32.
La file associée à la désignation de file sera appelée file émettrice, et nous appellerons résultat de l'instruction la partie de la file émettrice qui sera copiée, aux conversions près, dans la file réceptrice.

2) Une instruction d'affectation à une file est décomposée en un certain nombre (précisé ci-dessous) d'affectations élémentaires, chacune d'elles étant traitée selon le type de l'élément de la file réceptrice : affectation d'une valeur à un objet de type simple, affectation à une file ou à une structure. Ainsi, l'affectation à une file de N réels, d'une file de N entiers sera considérée comme une suite de N affectations d'un entier à un réel, avec conversion de l'entier avant l'affectation (cf. 6.32.).
La seule exception à cette règle est constituée par les files de caractères que nous traiterons en 7.378.

7.373. File réceptrice de taille fixe.

En aucun cas, la taille de la file réceptrice ne peut être modifiée. Si la file émettrice est plus grande que la file réceptrice, elle sera tronquée pour obtenir le résultat. Si elle est plus petite, seuls les éléments correspondants de la file réceptrice sont modifiés.

exemple :

```
A file (25) caract ; c (10) caract ;...
A = A (1 : 15), C ; remplace les dix derniers caractères
                    de A par ceux de C ;
A = A (1 : 20), 'X', C ; remplace les cinq derniers
     caractères de A par le caractère "X" et les quatre
     premiers de C ;
```

A = A, C ; est sans effet, l'instruction étant exécutée quand même ;

A = C ; remplace les dix premiers caractères de A par ceux de C ;

A = C, A (11 :.) ; cette écriture est équivalente à la précédente.

7.374. <u>File réceptrice de taille bornée</u>.

La taille du résultat est définie par :

inf (taille actuelle de l'émetteur, borne de la taille du récepteur).

La nouvelle "taille actuelle" de la file réceptrice est celle du résultat.

<u>Exemple</u> :

A <u>file</u> (25) <u>entier</u> ; B (<u>file</u> (<u>max</u> 50) <u>entier</u> ; C <u>réel</u> ; D (- 7 : 15) <u>réel</u> ;

Supposons que la taille actuelle de B soit 25.

B = B (1 : 10), A ; la taille actuelle de B devient 35 . il n'y a pas de conversion, les valeurs des 25 éléments de A sont affectées aux éléments de B de rang 11 à 35 ;

B = B, C ; la taille actuelle de B devient 36 ; la valeur de C est convertie en entier avant d'être attribuée à B (36) ;

B = B (1 : 15) ; la taille actuelle de B diminue (il est donc inutile d'introduire un opérateur de diminution d'une file) ;

B = B, D ; la taille actuelle de D passe à 28 ; la valeur de chaque élément de D est convertie en entier avant d'être affectée à l'élément correspondant de B ;

B = B, A ; la taille émettrice a pour taille 53 ; elle sera donc tronquée, la taille actuelle de B sera égale à sa taille maximum : 50.

Dans les deux cas d'une file réceptrice de taille fixe ou bornée, un indicateur est positionné si l'affectation a nécessité une troncature de la file émettrice. La variable booléenne réservée (c'est une variable de la classe "système" (cf. 1.3.)), $DEBFILE prend la valeur VRAI quand il y a un dépassement de capacité de la file réceptrice. Elle a pour valeur FAUX au début de l'exécution d'une chaîne de traitement. Elle est remise à FAUX lorsqu'elle est testée.

7.375. <u>File réceptrice de taille variable</u>.

C'est alors la file émettrice qui impose sa taille au résultat. La taille actuelle de la file réceptrice est modifiée, elle prend pour valeur la taille du résultat.

7.376. <u>La file réceptrice est une sous-file</u>.

Elle s'écrit par exemple ␣F (bi : bs) ou F (bi :.).
Si elle est de la première forme, elle sera traitée comme une file de taille fixe : sa taille est bs, si bs est inférieur ou égal à t, où t est la taille de la file F quand F est de taille fixe et la taille maximum de F si F est de taille bornée ; sinon sa taille est t. L'indicateur $DEBFILE est positionné en cas de débordement comme en 6.432.
Si la file réceptrice est de la seconde forme, elle sera considérée comme une file du même type que F.

7.377. <u>La file réceptrice est une file de files</u>.

Elle s'écrit par exemple
F (bi1 : bs1, bi2 : bs2) ou
F (␣, bi2 : bs 2) ou encore F (,).
L'affectation de valeurs à une telle file sera considérée comme une suite d'affectations aux files de dimension 1 composant F.
Ainsi F (,) = G (1 : 2, 1 : 2) sera équivalente à
F (,) = G (1, 1), G (1, 2), G (2, 1), G (2, 2)
soit F (1,) = G (1, 1), G (1, 2)
F (2,) = G (2, 1), G (2, 2)
et non F (1,) = G (1, 1), G (1, 2), G (2, 1), G (2, 2) si la taille de F le permettait.
Remarquons aussi que les écritures F (,) = G (,) et F = G sont toujours équivalentes.

7.378. Cas particulier d'une file de caractères.

Si la file émettrice et la file réceptrice sont toutes deux des files de caractères, l'affectation consiste alors à donner à la file réceptrice les valeurs de la file émettrice. Si la file émettrice n'est pas une file de caractère alors que la file réceptrice en est une, des conversions vont être nécessaires. Le choix de ces conversions est inspiré des problèmes posés par les entrées sorties. Il est lié également au fait que toutes les valeurs manipulées dans une chaîne de traitement sont en "format interne" (cf. introduction). Une file de caractères dont la valeur a été fournie par une lecture est une représentation fidèle de la chaîne de caractères externe.

Une affectation entre une file de caractères et un autre type d'objet doit donc s'accompagner d'une conversion faisant passer du "format externe" au "format interne" ou inversement.

Nous n'avons défini l'instruction d'affectation à une file de caractères, tout comme le cas de l'affectation d'une file de caractères à un autre type d'objet, que dans les cas simples et dont l'utilisation ne présente pas de risque d'ambiguïté. Pour les autres cas, pour que l'utilisateur reste conscient de ce qu'il écrit, nous définirons des fonctions prédéfinies ou des méta-fonctions selon la commodité d'emploi.

Un emetteur de type simple est converti en une file de caractères représentant sa valeur. La conversion a toujours lieu si la file réceptrice est de taille variable ; elle n'a lieu, dans les autres cas, que si la file est de taille suffisante. Si l'émetteur est entier ou décimal, la file de caractères est formée d'une chaîne de caractères représentant la valeur absolue de l'émetteur en décimal, précédée du signe "-" si l'émetteur est négatif. Si l'émetteur est réel, il est

écrit en "flottant normalisé" sous la forme :

± 0. mantisse sur 8 caractères ± exposant sur 2 caractères.

Si l'émetteur est complexe, il est écrit sous la forme de 2 réels séparés par un blanc (25 caractères).

Si l'émetteur est une structure dont les champs sont tous des files de caractères ou des caractères, leurs valeurs sont rangées dans la file réceptrice, sans séparation, dans l'ordre où elles figurent dans le mot des feuilles de la structure.

Ainsi, l'instruction d'affectation suivante, permet de donner à F la valeur d'une suite de caractères représentant un numéro de sécurité sociale.

```
sec soc struct (sexe car, année (2) car, mois (2) car,
      dépt (2) car, com (3) car, num (3) car) ;

f file car ;
    f = secsoc ; co cette instruction a le même effet que
                  la suite :
        f (1) = sexe ; f (2) = année (1) ; f (3) =
        année (2) ; f (4) = mois (1) ;

        f (5) = mois (2) ; f (6) = dept (1) ; f (7) =
        dept (2) ; f (8) = com (1) ;

        f (9) = com (2) ; f (10) = com (3) ; f (11) =
        num (1) ; f (12) = num (2) ;

        f (13) = num (3) oc ;
```

7.4 ELEMENT COURANT D'UNE FILE.

7.41 Définition.

Un élément courant est un identificateur qui "pointe" un objet dans une file, un emplacement. Dans le cas d'une file triée l'identificateur désigne en fait une valeur : celle contenue par l'emplacement qu'il pointe, mais cette différence n'a que peu d'importance quant à l'utilisation des éléments courants. Un élément courant est attaché à la file dans laquelle il peut "pointer" (1). On peut attacher plusieurs éléments courants à une même file.

Il peut être utilisé exactement comme l'identificateur (F (1) par exemple) d'un élément de la file :

- pour les conversions, il sera considéré comme étant du type des éléments de la file à laquelle il est attaché ;
- affecter une valeur à l'élément d'une file désigné par l'élément courant s'écrira X = 0 qui sera équivalent, si X désigne le 3ème élément de F, à F (3) = 0 ;
- utiliser la valeur de l'élément de la file désigné par l'élément courant s'écrira B = X qyi sera équivalent à B = F (3) ;
- X = X + 1 signifie donc : augmenter de 1 la valeur de l'élément désigné par X (2).

1) Un élément courant attaché à une file serait un repère de repère en Algol 68 (Van Wijngaarden) 69 et (Buffet, 72).

2) Ce qui aurait un sens bien différent en Algol 68 : X étant du mode rep rep réel par exemple, l'écriture X := X + 1, n'entraîne aucune modification à gauche de := (position "molle") et il y a un dérepérage de X à droite de := (position "forte") ce qui fait changer le nom désigné par X (passer à l'élément suivant), alors que la définition que nous donnons pourrait être écrite en utilisant un "forceur" à gauche de := c'est-à-dire décrire rep réel : X := X + 1.

Il est donc utilisé comme un objet du même type que les éléments de la file, et il y a en général une perte d'un niveau de référence (modification "dérepèrer" d'Algol 68). Il convient de remarquer que cette définition , syntaxiquement moins satisfaisante que celle d'Algol 68, reste cependant plus naturelle, mais qu'elle nous obligera à utiliser une notation différente de = pour affecter à un élément courant une valeur du type "nom d'emplacement d'une file", ce que nous verrons au paragraphe 7.431.

7.42 Déclaration.

Un élément courant devra toujours être déclaré. Sa déclaration pourra être implicite dans une instruction pour chaque. Elle pourra être faite explicitement sous la forme X de F, ou plus généralement, en suivant la règle :

<déclaration d'élément courant> ::= <identificateur> de <désignation de file>
[<option indice>] [option <rang>]
| <identificateur> de <déclaration de file>.

La désignation de file a été vue en 7.32. Les options "rang" et "indice" seront étudiées en 7.442. Le deuxième cas de la règle constitue une simplification d'écriture permettant de déclarer simultanément l'élément courant et sa file attachée.

i) Nous acceptons en fait au plus une option de chaque type (ordre, rang, indice) dans un ordre quelconque.

7.43 Attribution du nom d'un emplacement de la file à l'élément courant.

On a vu en 7.31 que l'affectation X = 0 permet d'attribuer une valeur à l'élément F (I) "pointé" par l'élément courant X.
F (I) désigne un emplacement de la file F, auquel on veut affecter une valeur, 0 par exemple. On veut que F (I) contienne la valeur 0. X "pointe" le nom de cet emplacement : pour que X = 0 soit équivalent à F (I) = 0, il faut faire un "dérepérage" à gauche du signe égal. Pour attribuer une valeur à X, c'est-à-dire pour modifier le nom de l'emplacement pointé par X, il ne faut pas faire ce dérepèrage. Il nous faut donc définir des instructions d'affectation particulières à ce cas, car elles modifient la fonction "pointer" et non la fonction "contenir". Elles permettront d'attribuer un nom d'emplacement à l'élément courant soit par l'intermédiaire de l'identificateur qui désigne cet emplacement, soit par une expression permettant de calculer cette nouvelle valeur, soit par une propriété : le nom attribué sera celui du premier emplacement de la file contenant une valeur possédant cette propriété.

7.431 Attribution explicite d'une valeur à l'élément courant d'une file.

a) Instruction "en"

L'instruction "en" permet d'attribuer à l'élément courant le nom d'un emplacement de la file par l'intermédiaire d'un identificateur de variable indicée ou par un déplacement par rapport à l'emplacement actuellement pointé. L'instruction "en" ne peut s'appliquer qu'aux files et non aux files triées, sauf pour passer d'un élément au suivant :
X en X + 1 ;
Elle s'écrit

<instruction en> ::= <identificateur d'élément courant> en
<identificateur d'élément courant> [+ <constante entière>]
| <identificateur d'élément courant> en <variable indicée>

Elle indique quel emplacement de la file doit pointer l'élément courant.

Exemples

X en Y ; X en X + 1 ; X en Y -3 ; X en F (15) ; X en F (12, 20) ;
sont des instructions "en" correctes si X et Y sont des éléments courants de la file F.
Remarquons qu'on retrouve alors le sens de X := Y d'Algol 68 quand X et Y sont du mode rep rep μ : il y a une modification "dérepérer" à droite de : =, pour passer au mode repμ.
D'autre part, dans X en F (X + 1), le premier X représente l'emplacement pointé par X, le second représente la valeur que contient cet emplacement.

b) Instructions "premier" et "dernier".

L'instruction "premier" s'écrit :

premier[<identificateur d'élément courant>].

Quand une instruction "premier" figure dans une instruction composée qui suit le "faire" d'un "pour chaque" attaché à un seul élément courant, l'identificateur d'élément courant peut être absent. L'instruction "premier" porte alors sur l'élément courant attaché à cette boucle.

Elle a pour effet de faire pointer à l'élément courant, le premier emplacement de sa file attachée. Ses indices prennent pour valeurs celles des bornes inférieures données dans la déclaration de la file ou de l'élément courant. C'est l'élément de rang 1 de la file associée.

L'instruction "dernier", qui s'écrit :

dernier[<identificateur d'élément courant>]

permet de faire pointer à l'élément courant le dernier emplacement de sa file associée.

Exemple :

X de F (10 :.) ;
premier X ; co X désigne alors l'élément F (10), son rang est 1 oc ;
---
dernier X ; co X désigne alors l'élément de rang t, où t est la taille actuelle de la file F - c'est donc l'élément F (t + 9) oc ;

Remarquons que l'instruction "premier" pourrait être décrite aisément en utilisant l'instruction "en" - on connaît en général la borne inférieure d'une file - ce qui serait moins commode pour l'instruction "dernier".

Nous conserverons donc "premier" pour des raisons de symétrie.

c) Ecritures condensées

Une écriture condensée permet de déclarer et initialiser simultanément l'élément courant d'une file : il suffira de faire précéder la déclaration de l'élément courant du symbole premier :

premier X de F (10 :.) ; est équivalent à X de F (10 :.) ; premier X ;

Dans le cas où l'on veut déplacer un élément courant par une instruction d'attribution de nom et l'utiliser immédiatement dans une instruction ou comme paramètre d'une procédure il sera plus commode d'inclure l'instruction d'attribution de nom dans l'utilisation de l'élément courant. L'utilisation de parenthèse évitera tout ambiguïté.

Exemples :

- X en X + 5 ; Y = X + 1 ; s'écrira Y = (X en X + 5) + 1 ;
- premier X ; Y = X + 1 ; s'écrira Y = (premier X) + 1 ;
- dernier X ; X = 3242 ; s'écrira (dernier X) = 3242 ou encore dernier X = 3242 ;
- X en F (15) ; X = 3242 ; s'écrira (X en F (15)) = 3242.

d) Instruction "en" et taille de la file associée.

Dans le cas des files de taille fixe, une instruction "en" n'a pas d'effet sur la taille de la file. L'attribution à un élément courant d'un nom d'emplacement qui ne figure pas dans sa file attachée entraîne l'émission d'un message d'erreur et l'arrêt de l'exécution.
Pour les files de taille maximum ou de taille variable, une instruction "en" peut entraîner une augmentation de la taille actuelle de la file, si le nouvel emplacement pointé par l'élément courant à un rang qui dépasse la taille actuelle de la file.

7.432 Attribution d'un nom par qualification. Instruction "soit".

<instruction soit> ::= soit <identificateur d'élément courant> <tel que>

<tel que> ::= tel que <souche> [sinon <instruction inconditionnelle>].(1)

1) Voir le chapitre 5 "tables de décisions".

<souche> (1) ::= <expression booléenne> | <identificateur de condition>.
L'instruction "soit" peut concerner l'élément courant d'une file ou d'une file triée.
Après l'exécution de l'instruction "soit", l'élément courant pointe le premier emplacement de sa file associée (c'est-à-dire d'une file éventuellement ordonnée), contenant une valeur qui vérifie la condition exprimée après tel que, si un élément existe. S'il n'existe pas, l'élément courant ne pointe plus un élément de la file - la fonction "sorti" aurait pour valeur VRAI (cf. 7.443) - et, si l'option "sinon" est présente, l'instruction qui suit sinon est exécutée.

L'instruction

soit X tel que C sinon module 1

dans laquelle X est un élément courant déjà déclaré et désigne une condition (elle pourrait être écrite dans un métamodule et C serait remplacé, lors d'une utilisation, par une expression booléenne), est équivalente à la suite d'instructions :

premier X ;

conditions	C	V	F	F
	dernier (X)	-	F	V
actions	X en X + 1	-	1	-
	Module 1	-	-	1
	retable	-	2	-

;

7.44 Rang et indices d'un élément courant.

Nous avons vu dans l'introduction de ce chapitre que la notion d'élément courant attaché à une file pour en désigner les éléments

(1) Voir le chapitre 5 "tables de décisions".

remplace avantageusement l'utilisation d'indices car, outre sa commodité, elle évite de calculer plusieurs fois la même adresse et favorise le traitement séquentiel. La notion d'indice disparaît dans ce cas.

Les indices peuvent cependant être utiles, et s'il faut les utiliser nous aurons recours à des fonctions calculant les indices ou le rang d'un élément, ou encore nous utiliserons les options "rang" et "indice" rencontrés à la déclaration d'un élément courant. Il ne s'agit pas là de la solution normale à adopter dans le cas d'un traitement aléatoire ; les variables indicées (élément d'une file désignée par ses indices) existent et sont prévues pour cela, mais de la solution à adopter pour un traitement essentiellement séquentiel, pour lequel l'utilisation du rang ou des indices de l'élément courant s'avèrent nécessaire dans quelques cas particuliers. Si l'utilisation du rang ou des indices est fréquemment nécessaire (pour accéder à d'autres files par exemple) au cours du traitement, on pourra le signaler à la déclaration de l'élément courant. Il sera alors associé à celui-ci des identificateurs d'entiers qui auront pour valeur le rang, ou les indices, de l'élément courant.

7.441 Fonctions "rang" et "indice".

a) Fonction "rang".

C'est une fonction à valeur entière. Elle n'est définie que pour un élément courant. Rang (X) a pour valeur celle du rang de l'emplacement de la file associée pointé par X au moment de l'appel. Il s'agit donc du rang actuel d'un élément, après l'opération de tri qui pourrait être demandée par une option d'ordre.

Si l'élément courant n'a pas été initialisé, la valeur de rang (X) est indéterminée.

b) Fonctions indices.

1) File de dimension 1.

C'est une fonction à valeur entière. Elle s'écrit indice (X).

Elle n'est définie que pour un élément courant déjà initialisé.

Sa valeur est celle de l'indice de l'emplacement pointé par X au moment de l'appel, dans la file désignée par X.

Exemple :

premier X de F (14 :.) ; I = rang (X) ; J = indice (X) ;

I vaut 1 et indice vaut 14.

ii) File de dimension supérieure à 1.

Les fonctions indices s'écrivent alors indice n (X) où X est un élément courant et n un chiffre indiquant le rang de l'indice désiré. Elles ont les mêmes caractéristiques que la fonction indice.

Exemples :

premier X de F (20, 11 : 12, 2) ; co il s'agit là d'une déclaration simultanée de X et de F : les bornes inférieures manquantes valent 1 oc ;

X en X + 3 ; I = rang (X) ; J = indice 2 (X) ; K = indice 3 (X) ;
L = indice 1 (X) ;
I vaut alors 4, J vaut 2, K vaut 2 et L vaut 1.

7.442 Rang et indices demandés à la déclaration d'un élément courant.

Dans le cas où l'utilisation du rang et des indices d'un élément courant est souvent nécessaire au cours du traitement d'une file, plutôt que d'appeler chaque fois la fonction "rang" ou une fonction "indice", il sera préférable d'indiquer à la déclaration de l'élément courant, quel (s) identificateur (s) de variable entière devra (ont) être associé (s) au rang, ou aux indices de l'emplacement pointé par l'élément courant.

Syntaxe (cf. 7.42 déclaration d'un élément courant) :

<option rang> ::= rang <identificateur>.

<option indice>::= indice <identificateur> | indice (<identificateur> [, <identificateur>]*).

L'écriture X de F (,,) rang L indices (I, J, K) ;
déclare l'élément courant X attaché à la file F déjà déclarée considérée comme une file de dimension 3. Elle demande en plus qu'à tout moment L ait la valeur de rang (X), I celle de indice 1 (X), J celle de indice 2 (X) et K celle de indice 3 (X).

Si les identificateurs cités dans une option rang ou indice ne sont pas déclarés, la déclaration d'élément courant qui les contient constitue leur déclaration. Ils sont implicitement du type entier . S'ils sont déjà déclarés, les objets correspondants seront pris pour recevoir les valeurs des fonctions rang et indice - (un message sera édité à la compilation).

7.443 Fonctions "dernier", "premier", "sorti".

Ce sont des fonctions à valeur booléenne. Elles s'écrivent

dernier (X), premier (X), sorti (X)

où X est un élément courant. Dernier (X) (resp. premier (X)) a pour valeur VRAI si X pointe le dernier (resp. le premier) emplacement de la file associé à X, et FAUX sinon. Sorti (X) a pour valeur VRAI si X désigne un élément de sa file associée, FAUX sinon.

7.5 EXEMPLES D'UTILISATION.

1 Recherche du nombre de mots d'une phrase.

F est une file de caractères contenant une phrase. Les mots sont séparés par des blancs, un point ou une virgule. On veut compter , dans Y, le nombre de mots de cette phrase. Y et F sont définis dans une classe C.

module compte mots ; utilise C ;

Y = 0 ;

soit X de F tel que X ≠ '␣' ; co s'il n'existe pas de tel élément X n'est pas défini ; sorti (X) vaut donc vrai oc ;

conditions	X = '␣'	F	+ F	+ F	V
	X = ' . '	F	+ F	V	F
	X = ' , '	F	V	+ F	F
	sorti (X)	F	F	F	F
actions	soit X tel que X≠ '␣'	-	-		1
	soit X tel que X= '␣'	2	1	1	
	Y = Y + 1				
	retable	3	2	2	2

ft fin module ;

2. Exemple 2.

E est une file d'éléments structurés dont le champ A est du type file 10 caract.C est une file de 10 caractères. On veut compter les éléments de la file E dont le champ A est classé alphabétiquement avant la valeur de C et copier les autres dans une file F tant que F n'est pas remplie. E, F, C et COMPTE sont déclarés dans une classe T.

Module E X 2 ; utilise T ;
COMPTE = 0
Premier X de E ; premier Z de F ;

conditions	A de X ≤ C	V	F
	sorti (X)	F	F
	sorti (Z)	F	F
actions	X en X + 1	3	3
	Z en Z + 1	2	2
	Z = X	-	1
	COMPTE + COMPTE + 1	1	-
	Retable	4	4

ft fin module ;

3. Classement d'une file sur elle-même.

F est une file d'éléments du type entier, réel, décimal, caractère ou file de caractères. A est du type des éléments de F. F est déclarée dans la classe E. On veut ranger F par ordre croissant.

Module classer ; utilise E ; A comme élément de F ;
premier X de F ; Y de F rang J ; X en X + 1 ;
Y en X ;

conditions	Y > F (J-1)	V	F
	sorti (X)	F	F
actions	intervertir		1
	X en X + 1	1	
	Y en Y + 1		2
	Y en X	2	
	si non premier (Y) alors Y en X		3
	retable	3	4

ft ;

procédure intervertir ; A = X ; X = Y ; Y = A fin proc fin mod ;

7. 6 CALCULS ITERATIFS ATTACHES A UNE FILE : INSTRUCTION "POUR CHAQUE".

Comme on le fait habituellement dans les langages de programmation, nous définissons des instructions de répétition permettant de décrire des calculs itératifs. En civa, en plus des boucles "pour" classiques, qui ont été vues au chapitre 6, nous introduisons des instructions spécifiques pour décrire le traitement global d'une file triée ou non : les boucles "pour chaque". Elles sont considérées comme une écriture condensée d'une suite d'instructions portant sur l'élément courant attaché à la file à traiter.

7.61 Syntaxe.

<instruction pour chaque> ::= pour chaque <liste d'éléments de pour chaque>
<fin de pour>.

<liste d'éléments de pour chaque> ::= <élément pour chaque> |
<élément pour chaque>, <liste d'éléments de pour chaque>.

<élément pour chaque> ::= <élément courant> [<tel ou tant que>].

<tel ou tant que> ::= <tel que> [<tant que>] | <tant que> [<tel que>].

<élément courant> ::= <déclaration d'élément courant> | <identificateur d'élément courant>.

7.62 Remarques.

Une instruction "pour chaque" peut être attachée à une file simple, une file de dimension supérieure à 1, ou à plusieurs files, ou à des sous-files, que ce soit des files triées ou des files simples.
Cette déclaration peut être explicite, avant une instruction pour chaque : (la déclaration d'élément courant a été vue au paragraphe 7.42 <tel que> et <tant que> ont été vus en 7.432). Dans celle-ci figure seulement l'identificateur d'élément courant déjà déclaré. Elle peut aussi être "implicite" l'instruction pour chaque contenant la déclaration de l'élément courant.

L'instruction générale

pour chaque X1 tel que C tel 1 tant que C tant 1, X2 tel que C tel 2 tant que C tant 2,..., Xn tel que C tel n tant que C tant n faire proc fp ;

est une écriture condensée de la séquence d'instructions :

premier X1 ; premier X2 ;.....; premier Xn ;

conditions	C tel 1	V	F	-		-
	C tel 2	V	-	F		-
	.	.	-	-		.
	.	.	-	.		.
	.	.	.	-	...	-
	C tel n	V	-	-		F
	cond 1	V	V	V	...	V
	cond 2	V	V	V	...	V
actions	proc	1				
	retable	3	2	2	...	2
	X1 en X1+1	2	1			
	X2 en X2+1	2		1		
		.				
		.				
		.				
	Xn en Xn+1	2				1

ft ;

dernier X1 ; dernier X2 ;....; dernier Xn ; (1)
booléen procédure COND 1 ; COND 1 = C tant 1 ou C tant 2 ou C tant 3....
ou C tant n f proc ;
booléen procédure COND 2 ; COND 2 = sorti (X1) ou sorti (X2) ou ... ou
sorti (Xn) f proc ;

7.63 Traitement d'une file simple.

7.631 Remarques.

- Il n'y a qu'un seul élément de pour, l'instruction étant attachée à une seule file.
- La partie dimension est réduite à 1 paire de bornes (cas d'une sous-file).
- Rappelons que si le rang ou l'indice de l'élément courant sont utilisés souvent, il est préférable de l'indiquer à la déclaration de l'élément courant : leur valeur sera entretenue à chaque itération (voir <option rang> et <option indice> dans <déclaration d'élément courant>, au paragraphe 7.442).
- Si une option d'ordre est présente, il y aura eu un classement avant l'exécution de la boucle (le classement étant fait à la déclaration de l'élément courant).
- Il est impossible d'associer à un "pour chaque" une file simple de taille variable qui n'aurait pas encore de valeur : sa taille actuelle est nulle et dès la première fois qu'on entre dans la boucle, celle-ci est abandonnée.

7.632 Nombre d'itérations.

Le nombre d'itérations à exécuter est déterminé par la taille de la file attachée à l'élément courant contrôlé par le "pour chaque" et par les options.

(1) Ces instructions sont utiles en cas de file contenant des enregistrements (fichier). Elles demandent une fermeture des fichiers.

La taille prise en compte est la taille de la file s'il s'agit d'une file de taille fixe (triée ou non), la taille actuelle s'il s'agit d'une (triée ou non), de taille bornée ou variable. Le rang R du dernier élément traité est celui correspondant à la borne supérieure indiquée dans la déclaration de l'élément courant ou, si elle lui est inférieure, à la taille actuelle de la file.

Le nombre d'itérations est donc égal à R, diminué du nombre d'éléments qui ne vérifient pas la condition tel que .

7.633 Exemples d'utilisation.

Exemple 1. Recherche de maximum

E est un ensemble d'objets structurés à 3 champs A, B, C ; A et C sont entiers et B une file de caractères. On veut chercher le champ B de l'objet dont le champ C n'est pas nul et A maximum.

MAX = MINENTIER ; Y de E ;
Pour chaque X de E tel que C ≠ 0 faire
si A > MAX alors Echange fp ;
F = B de Y
procédure échange ; MAX = A ; Y en X fproc ;

X et Y sont des éléments courants attachés à E. Il est inutile de préciser C de X ≠ 0 (de même pour A) car nous sommes dans une boucle "pour chaque" contrôlée par X. Par contre à l'extérieur de la boucle il est indispensable de qualifier B (B de Y).

Exemple 2. Somme

S = 0 ; pour chaque X de F faire S = S + X fp ;

où F est une file d'entiers ou de réels ou de décimaux et X l'identificateur d'élément courant ; X n'était pas encore déclaré. Cette séquence permet de calculer la somme des éléments de F.

Exemple 3. Autre recherche de maximum

On veut connaître la suite de caractères classés alphabétiquement la dernière parmi les files H de G (> est alors une comparaison alphanumérique entre files de caractères) ;

FILEMINI est une file de 9 caractères égaux au plus petit caractère alphanumérique. MAX est aussi une file de 9 caractères.

G est une file de files (H) de 9 caractères. Elle est traitée comme une file simple.

MAX = FILEMINI ;

Pour chaque X de G faire si X>MAX alors MAX = X fp ;

Exemple 4. Option d'ordre.

Pour chaque X de G (5 :. pas 2) par ordre croissant faire proc 1 fp ; où G est une file de taille connue à cet instant. La procédure proc1 sera exécutée pour les éléments de G compris entre l'indice 5 et la fin de la file, pris de deux en deux et traités dans l'ordre croissant de leur valeur. (Les éléments de G peuvent être de type entier, réel, décimal, file de caractères).

Exemple 5. Edition

G est une file de 150 éléments structurés,- dont un champ (file de caractères) s'appelle A.
Pour chaque Y de G par A décroissant faire éditer fp ; où éditer est une procédure.

Exemple 6. Options indice et tant que.

Pour chaque Z de F indice I tant que Z>0 faire proc 3 fp
La procédure proc 3 sera exécutée pour les éléments de F dans leur ordre de rangement dans F jusqu'au moment où l'on rencontre un élément courant Z tel que $Z \leq 0$. L'identificateur I pourra être utilisé dans proc 3 (si proc 3 est dans la portée de I, c'est-à-dire : si proc 3 est déclaré dans ce module et si I n'a pas de déclaration explicite (indice I constitue implicitement sa déclaration) ou si proc 3 est déclaré dans une classe utilisant (ou étant) la classe qui contient la déclaration de I).

Exemple 7. Classement d'une file sur elle-même.

Cet exemple reprend l'exemple 3 du paragraphe 7.5.
F est une file d'éléments du type entier, réel, décimal, caractère ou file de caractères. A est du type des éléments de F ; F est déclaré dans la classe C. On veut ranger F par ordre croissant.

```
Module classer ; utilise C ; A comme élément de F ;
Pour chaque X de F faire
     A = X ;
     Pour chaque Y de F (rang (X) - 1) :.) pas -1 tel que Y<X
                                faire glissery fp ;
     F (rang (Y)) = A fp ;
procédure glissery ; F (rang (Y) + 1) = Y fproc ;
```

7.64 Traitement des files de dimension supérieure à 1.

7.641 Remarques.

. Il n'y a qu'un seul élément de pour, l'instruction étant attachée à une seule file.

. La partie dimension esttoujours présente dans ce cas ; le nombre de virgules entre les deux parenthèses, augmenté de 1, est égal à la dimension de la file.

. Les instructions relatives au traitement d'élément courant portent sur la file associée à l'élément courant, c'est-à-dire après développement et mise en ordre de la file attachée.

. La file attachée est développée en utilisant les indices dans l'ordre de rangement en mémoire (ligne par ligne).

Pour tout traitement dans un ordre différent des indices, il faudra utiliser plusieurs boucles "pour chaque" simples et imbriquées, ou des boucles "pour" (cf. chap. 6).

7.642 Nombre d'itérations.

Le nombre d'itérations est déterminé par les tailles des files composant la file à traiter.

Soit F une file de dimension n.

Le premier élément susceptible d'être traité est F (O1, O2,...., On)
où Oi = sup (bidei, b idfi)
et bidei est la borne inférieure donnée à la déclaration de l'élément courant attaché à F pour l'indice d'ordre i,
bidfi la borne inférieure donnée à la déclaration de la file F pour l'indice d'ordre i.
Le dernier élément susceptible d'être traité est F (E1, E2,...., En)
où Ei = inf (bsdei, bsdfi)

et bsdei est la borne supérieure donnée à la déclaration de l'élément courant.
bsdfi la borne supérieure donnée à la déclaration de la file pour une file de taille fixe, la borne actuelle pour une file bornée ou variable.
Le nombre d'itérations est égal au produit des (Ei - Oi + 1) (c'est-à-dire le nombre d'éléments situés entre le premier et le dernier susceptible d'être traités), diminué du nombre d'éléments qui ne vérifient pas la condition "tel que".
Il peut encore être inférieur si une option "tant que" est présente.

7.643 Exemples d'utilisation.

Exemple 1. Somme de matrices.

A, B, C sont trois files (d'entiers par exemple), de même dimension.
Pour chaque Y de C(,) indice (I, J) faire
    Y = B (I, J) + A (I, J) fp
permet de calculer C = A + B. Si C est de dimension connue.

Nous verrons une forme plus concise et plus naturelle de cette expression dans l'exemple 1 du paragraphe 7.652.

Exemple 2. Produit de deux matrices.

Soient les matrices A et B représentées par les files A (N, M) et B (M, L).
On veut calculer C = A*B représentée par la file C (N, L), c'est-à-dire $C(i, j) = C(I, J) = \sum_{k=1}^{n} A(I, K) * B(K, J)$.
ce qui s'écrirait en Algol par exemple :

```
Pour I = 1 pas 1 jusqu'à N faire
Pour J = 1 pas 1 jusqu'à L faire
    début C (I, J) = 0 ;
    Pour K : = 1 pas jusqu'à M faire
        C I, J : = C I, J + A I, K   B K, J
    fin ;
```

Ce calcul demande 4*N*L*M + N*L calculs d'adresse d'une variable indicée de dimension 2, c'est-à-dire au minimum 2 sommes et un produit chaque fois.

On pourrait écrire :

Pour chaque X de C indice (I, J) faire
  X = 0 ;
Pour chaque Y de A (I) indice K faire
  X = X + Y B (K, J) fp fp ;

Ce calcul demande alors

1 + N + N*2*M calculs d'adresse d'une variable indice de dimension 2 et 3*N*L*M progressions de 1 d'une adresse d'élément courant, ce qui supprime approximativement 3*N*L*M additions et autant de produits.

7.65 Traitement de plusieurs files.

7.651 Remarques.

- Les files ainsi associées dans un "pour chaque" sont de type quelconque.
- Ces files sont décrites dans des éléments de pour, séparés par des virgules.
- Dans chaque élément de pour, on peut rencontrer des options "tel que", "tant que", ou "pas". Si une condition "tel que" n'est pas vérifiée, on passe à l'élément suivant uniquement pour la file correspondante.
- Le nombre d'itérations est le plus petit des nombres déterminés en 7.632 ou en 7.642 pour chacune des files figurant dans le pour chaque.

7.652 Exemples.

Exemple 1. Somme de deux matrices.

Il reprend d'exemple 1 de 7.643.

Pour chaque X de A, Y de B, Z de C faire
  C = X + Y fp

qui évite complètement tout calcul d'adresse de variable indicée de dimension 2.

Exemple 2. Produit scalaire.

V et W sont deux vecteurs représentés par les files V et W.

S = 0 ;

Pour chaque X de V, Y de W faire S = S + X*Y fp ;

Exemple 3. Copie d'une file dans une autre.

a) La file réceptrice est de taille fixe.

Soit à copier la file E dans l'ensemble F de taille fixe 2000. E est de type quelconque. L'élément de E et de F est de type structuré à deux champs : A qui est une file de 10 caractères et B qui est un entier On veut copier E par ordre alphabétique des noms A.

Pour chaque X de E par A croissant, Z de F faire Z = X fp ;

On aurait pu également écrire : par A de X croissant. Ce n'est pas indispensable : A sera implicitement considéré comme qualifié par X.

b) La file réceptrice est de taille quelconque, non encore fixée.

La forme précédente est inutilisable car l'ensemble F est vide au début de la boucle et dès le premier test on trouve qu'il faut quitter la boucle et on n'exécute rien. On pourrait écrire :

premier Z de F ;

Pour chaque X de E par A croissant faire

Z en Z + 1 = X fp ;

Exemple 4. Copie d'éléments d'une file dans une file.

a) Cas simple.

On veut cette fois copier les éléments de E dont le nom A est classé alphabétiquement après la file de 10 caractères C, sans rangement par ordre alphabétique. On suppose la taille de F fixée.

Pour chaque X de E tel que A de X > C, Z de F faire

Z = X fp ;

b) Avec comptage.

On veut cette fois compter les éléments dont le nom A est classé alphabétiquement avant C ou égal à C et copier les autres dans F. On pense que lorsqu'on rencontre un élément ≤C, il y a des chances d'en avoir d'autres derrière lui.

COMPTE = 0 ;

Pour chaque X de E, Z de F faire

si A de X >C alors Z = X sinon

pour chaque X de E (rang (X) :.) tant que A ≤ C faire COMPTE = COMPTE + 1 fp fp ;

On utilise là deux "pour chaque" imbriqués. Le rang de l'élément courant dans le second est initialisé à la valeur "rang" (X)" qui est le rang de l'élément courant dans le premier. Sans cette initialisation, le second "pour chaque" commencerait avec le premier élément de E.

Si on en veut pas tenir compte de cette probabilité de trouver des séquences d'éléments de E dont le nom A est C, on pourra écrire simplement ...

COMPTE = 0 ;

Pour chaque X de E, Z de F faire

si A de X >C alors Z = X sinon COMPTE = COMPTE + 1 fp ;

Remarquons que, si E est une file triée et si l'on veut copier, par odre alphabétique, les éléments de E dont le nom A est supérieur à C et compter les autres, l'écriture serait assez simple.

CC = 0 ;

Pour chaque X de E pas A croit tant que A ≤ C faire

CC = CC + 1 1 fp ;

Pour chaque X de E (rang (X) :.), Z de F faire Z = X fp ;

E étant une file triée aura été classée sur elle-même.

Si E est une file il faudrait d'abord la classer sur elle-même pour pouvoir utiliser une écriture analogue.

X de E ;

E = E par A croissant ;

Pour chaque X de E tant que A ≤ C faire CC = CC + 1 fp ;

Pour chaque X de E (rang (X) :.), Z de F faire Z = X fp ;

Exemple 4. Copie d'une file sur deux autres.

La file E contient quelquefois des éléments dont le nom A est égal à C. On veut copier les séquences comprises entre ces éléments, alternativement sur une file G et sur une file H. Les éléments dont le nom est C ne seront pas recopiés.

premier Y de G ; Z de H ; premier Z ;

Pour chaque X de F faire

```
    si A de X = C alors procédure 1 sinon Z en Z + 1 = X fp ;
procédure procédure 1 ; X en X + 1 ;
    pour chaque X de F (rang (X) :.) faire Y en Y + 1 = X fp
        fproc ;
```

Exemple 6. Interclassement de deux files triées.

Soient F et G deux files d'éléments courants X et Y, supposées classées par ordre croissant, et H une file réceptrice d'élément courant Z. X, Y, Z pourraient être de type entier, réel, caractère, décimal, ou file de caractères. X, Y, Z sont supposés initialisés. Le traitement pourrait être décrit par la table :

Décision	X > Y	V	V	F	F
	dernier (X)	-	-	F	V
	dernier (Y)	F	V	-	-
Actions	Z en Z + 1 = X			1	1
	Z en Z + 1 = Y	1	1		
	X en X + 1			2	
	Y en Y + 1	2			
	Pour chaque X de F (rang (X):.) faire Z en Z + 1 = X fp		2		
	Pour chaque Y de G (rang(Y):.) faire Z en Z + 1 = Y fp				2
	retable	3		3	

7.66. Extension au cas des ensembles.

Comme nous l'avons dit en 7.16., l'instruction "pour chaque" peut être étendue au cas d'un ensemble grâce aux conversions qui sont déterminées syntaxiquement. Une désignation de file qui figure après le nom d'un élément courant peut être remplacée par une désignation d'ensemble. L'ensemble ainsi désigné est converti en une file et c'est sur cette file équivalente qu'est définie l'instruction "pour chaque" ainsi que l'élément courant et les instructions de déplacement de l'élément courant.

REFERENCES DU CHAPITRE 7

BUFFET J. 72 BUFFET J., ARNAL P., QUERE A. (éd.)
Définition du langage algorithmique Algol 68.
HERMANN. PARIS, 72.

DUCLOY J. 73 Compilation dans le projet CIVA.
Doctorat Ingénieur. NANCY 1. Mars 73.

NOLIN L. 69 Formalisation des notions de machine et de programme.
GAUTHIERS-VILLARS. PARIS, 69.

71 ATF 71. Rapport interne. Institut de Programmation.

PAIR C. 71 Les structures d'information et leur représentation en mémoire. Cours à l'école d'été d'informatique d'ALES. Juillet 71. Université de NANCY.

PECCOUD D. 72 Un langage et une machine pour traiter les files.
Thèse de 3ème cycle. Université PARIS VI. Mars 72.

VAN WINJGAARDEN A. 69 Report on the algorithmic language Algol 68.
Math. Centrum AMSTERDAM, 69.
Traduction dans (BUFFET J., 72).

CHAPITRE 8

INSTRUCTIONS ITERATIVES ET CINEMATIQUE DES FICHIERS

8.1. INTRODUCTION.

8.2. SOUS-ENSEMBLE LOGIQUE ET INDICATIF.

8.21. Sous-ensemble logique.
8.22. Niveau d'indicatif.
8.23. Indicatifs et identification.
8.24. Indicatifs et préordre d'un ensemble.

8.3. INSTRUCTIONS D'ITERATION LOGIQUE SIMPLE.

8.31. Ecriture de l'instruction.
8.32. Déclaration d'un indicatif.
8.33. Evaluation d'une instruction d'itération logique simple.
8.34. Exemple d'utilisation.

8.4. INSTRUCTIONS D'ITERATION LOGIQUE MULTIPLE.

8.41. Définition.
8.42. Ensemble directeur.

8.5. EXEMPLES D'UTILISATION.

8.6. EXTENSION AU CAS DES FILES.

CHAPITRE 8

INSTRUCTIONS ITERATIVES ET CINEMATIQUE DES FICHIERS

8.1. INTRODUCTION.

Le traitement automatique de l'information est basé essentiellement sur la répétition d'actions à entreprendre :

- répétition de calculs sur un objet, ou un ensemble d'objets, chaque itération conduisant à de nouvelles valeurs utilisées pour l'itération suivante ;
- exécution d'un calcul pour un nombre important de groupes de valeurs, ces groupes et leur traitement n'étant décrits qu'une seule fois.

La notion de file est bien adaptée aux répétitions du second type : une file étant une suite d'emplacements contenant une valeur, les traitements peuvent être effectués successivement pour chacun des emplacements. Mais c'est une notion trop simple pour pouvoir décrire aisèment tous les cas de traitement : chaque emplacement reçoit un même et unique traitement et l'on peut vouloir, au contraire, associer des traitements pouvant être différents selon les valeurs contenues par les emplacements traités. Le cas le plus fréquent, celui que nous traiterons ici, correspond aux ensembles triés selon un certain nombre de critères et dont on veut traiter tous les éléments de la même manière avec en plus, un traitement complémentaire chaque fois qu'en passant d'un emplacement au suivant un critère change de valeur. Nous appellerons <u>point de rupture</u> du critère I, un élément d'un ensemble tel que la valeur du champ I dans cet élément est différente de celle qu'il a dans son prédecesseur (dans la file support de l'ensemble). Dans ce cas, nous appellerons <u>variable de rupture</u> ou <u>indicatif</u>, le critère I.

<u>Exemple</u> :

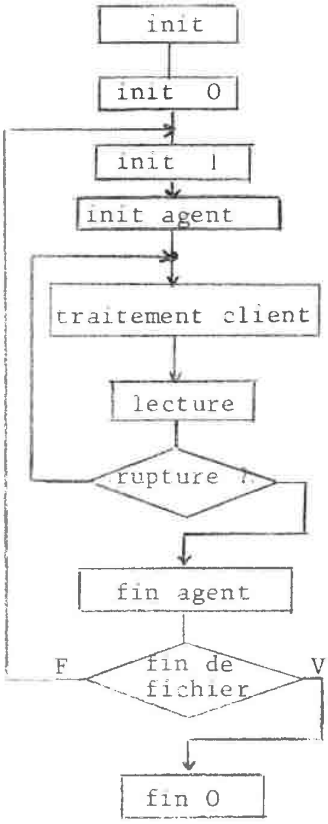
soit E un ensemble d'éléments structurés et l'indicatif I :

Le 3ème et le 5ème emplacements de E sont des points de rupture. Nous dirons que le 1er emplacement est également un point de rupture. Pour décrire des traitements particuliers aux points de rupture, nous constatons sur cet exemple qu'il serait assez commode de considérer E, non plus comme un ensemble de structure mais comme un ensemble d'ensembles de taille variable dont les éléments sont ceux de E et dont les tailles sont déterminées à partir des valeurs des indicatifs : chaque ensemble ne contenant que des indicatifs de même valeur.

La programmation des points de rupture est toujours un problème épineux à résoudre alors qu'il semble simple à exprimer.

Notons que les instructions "pour chaque" définies au chapitre précédent permettent de se dégager des problèmes de lecture et d'écriture des éléments d'un fichier. D'autre part, ce problème peut déjà être résolu simplement en CIVA, pour les informations issues de fichiers externes : à leur acquisition, la file interne obtenue serait alors constitutée de file, chaque file correspondant à une valeur d'indicatif.

<u>Exemple</u> : Considérons le fichier des clients d'une compagnie d'assurances. Chaque client est caractérisé par un matricule (tous les matricules sont différents). D'autre part, pour chaque client est indiqué le numéro de l'agent dont il dépend. On veut faire le cumul, pour chaque agent, des primes versées par les clients et le cumul pour la compagnie. Le problème est donc facile à énoncer. Sa résolution l'est moins. Elle conduit à une description telle que la suivante :




```
init : total général = 0 ;
init 0 : FFC = 0 ; lire (fich client) ;
         si $ debfile alors FFC = 1 ;
         co $ debfile indique une fin de file oc ;
init 1 : ω 1 = agent ;
init agent : total agent = 0 ;
traitement client : total agent = total agent +
                                  prime ;

lecture : lire (fichclient) ;
          si $ debfile alors FFC = 1 ;
rupture : si FFC == 0 ou agent == ω 1

          alors  aller à traitement client ;

fin agent : éditer (total agent) ;
            total général = total général +
                            total agent ;
fin de fichier : si FFC = 0 alors aller à
                            initagent ;
fin 0 : éditer (total général) ;
```

La solution est donc moins immédiate à décrire. On peut cependant remarquer que la plus grande partie de celle-ci est indépendante du problème précis posé : elle ne dépend que du nombre de variables de ruptures et de relations entre elles. Dans le cas plus général, l'organisation des traitements peut dépendre aussi du nombre de fichiers en entrée. Ainsi, l'exemple ci-dessus est entièrement déterminé par la donnée des "paramètres" suivants :

- le fichier traité est celui des clients ;
- la variable de rupture est "agent" ;
- init ;
- init agent ;
- traitement client ;
- fin agent ;
- fin 0.

Et il est naturel d'exprimer la solution sous une forme analogue à celle de l'énoncé :

```
init ; pour chaque agent faire
                init agent ;
                pour chaque client faire traitement client fpc ;
                fin agent fpc ;
fin O ;
```

Remarquons cependant que la solution du cas général est un peu plus complexe que dans l'exemple ci-dessus pour lequel on a fait l'hypothèse qu'une rupture du niveau "agent" est toujours détectée par une rupture du niveau "client" (les numéros de clients sont tous différents), ce qui n'est pas toujours le cas.

Avec les langages de programmation habituels, la mise au point d'outils permettant de décrire facilement la solution des problèmes de rupture permet de dégager le programmeur des problèmes de mouvements de fichiers. On dit qu'il réalisent la cinématique automatique des fichiers. Ces problèmes sont assez indépendants du langage utilisé par le programmeur.

On trouvera dans (HERTSCHUH N., 74) une étude plus complète de la cinématique des fichiers, des divers cas rencontrés et une justification approfondie de l'automatisation de cette cinématique. Nous sommes alors conduits à définir des instructions d'itération (telles que le "pour chaque" utilisé ci-dessus) permettant de décrire la cinématique des fichiers. Elles peuvent alors servir d'entrée à une génération de programmes dans un langage de programmation classique. Dans (HERTSCHUH N., 74) on trouvera la description et les règles d'utilisation d'un tel générateur de programmes en Cobol.

Dans CIVA, les informations sont toutes en format interne et les instructions d'itération "pour chaque", ou les instructions permettant de modifier le nom "pointé" par l'élément courant, dispensent déjà l'utilisateur d'avoir à se préoccuper des lectures et des écritures sur ses fichiers. La résolution des problèmes de rupture se résume alors à déterminer à quels moments déplacer un élément courant. Pour cela, nous étudierons l'organisation logique d'un ensemble. Nous préciserons alors la notion de sous-ensemble logique et celle d'élément courant attaché à un sous-ensemble logique (appelé habituellement "indicatif") : un ensemble à traiter est alors converti en un ensemble de sous-ensembles logiques, pour lesquels on pourra donc utiliser les instructions "pour chaque" rencontrées au paragraphe 7.6. Des exemples permettront d'illustrer la commodité d'emploi et le caractère naturel de l'outil obtenu.

8.2. SOUS-ENSEMBLE LOGIQUE ET INDICATIF.

8.21. Sous-ensemble logique.

Soit E un ensemble d'objets d'un type structuré t, et soit I le nom d'un champ de cette structure.

Nous appelons sous-ensemble logique dans E, d'indicatif I, un sous-ensemble de E dont tous les emplacements contiennent la même valeur pour le champ I, et muni du même préordre que E.

Les sous-ensembles logiques dans E, d'indicatif I, réalisent une partition de E, en prenant toutes les valeurs présentes de l'indicatif I.

Nous introduirons donc l'opération de partition selon un indicatif I qui fait passer d'un ensemble E, à l'ensemble de ses parties.

8.22. Niveau d'indicatif.

Soit F un sous-ensemble logique dans E, d'indicatif I. F étant un sous-ensemble de E, on peut définir de même un sous-ensemble logique dans E, d'indicatifs I, J comme étant un sous-ensemble logique dans F d'indicatif J. I est alors dit indicatif de niveau supérieur à J.

De même un sous-ensemble logique dans E d'indicatifs I1, I2, ... In est un sous-ensemble logique d'indicatif In dans un sous-ensemble de E défini comme un sous-ensemble dans E, d'indicatif I1, I2, ... In1.

I1 est parfois qualifié d'indicatif majeur, In d'indicatif mineur.

8.23. Indicatifs et identification.

L'emploi habituel des indicatifs en analyse correspond au cas où l'ensemble des indicatifs attachés à un ensemble permet d'identifier les éléments de cet ensemble. C'est-à-dire que les sous-ensembles logiques dans un ensemble E, d'indicatifs I1, ... In ne contiennent chacun qu'un seul emplacement.

Mais ce n'est pas le cas général : par exemple dans l'ensemble des mises à jour d'un stock, un même article pourra être rencontré plusieurs fois pour être modifié.

8.24. Indicatifs et préordre d'un ensemble.

Dans l'exemple de 8.1. les traitements décrits portent sur les sous-ensembles logiques de l'ensemble "fichier client", d'indicatif "numéro d'agent" puis sur chaque élément de ces sous-ensembles. D'une façon générale, les traitements demandés au cours de l'analyse portent sur des sous-ensembles logiques. Pour pouvoir les effectuer correctement, encore faut-il que tous les éléments d'un sous-ensemble soient présentés consécutivement dans la file support de l'ensemble. D'où la nécessité de trier l'ensemble de départ selon les indicatifs.

Soit E un ensemble et I un indicatif de E. E est muni d'un préordre et soit F une file support de E (les emplacements égaux selon ce préordre y sont ordonnés arbitrairement).

Si E est trié selon le critère I, on a la propriété :

(1) $\forall j \notin [bi, bs]$ (I <u>de</u> F (j) $\neq$ I <u>de</u> F (bi))

où bi et bs sont définis par :

$\forall j \in [bi, bs]$ (I <u>de</u> F (j) = I <u>de</u> F (bi)) ;

si F (bi - 1) existe, I <u>de</u> F (bi - 1) $\neq$ I <u>de</u> F (bi) ;

si F (bi + 1) existe, I <u>de</u> F (bi + 1) $\neq$ I <u>de</u> F (bi).

Ce qui nous donne une façon de mettre en oeuvre la recherche des sous-ensembles logiques d'indicatif I, en cherchant les changements de valeur de l'indicatif lors de l'examen successif des différents emplacements de l'ensemble (recherche des ruptures).

8.3. INSTRUCTION D'ITERATION LOGIQUE SIMPLE.

8.31. Ecriture de l'instruction.

Soit I un indicatif attaché à un ensemble E. L'instruction d'itération logique simple :

Pour chaque I faire m fpc ;

permet de demander l'exécution du module m successivement pour chacun des sous-ensembles logiques dans E, d'indicatif I.

On peut considérer que la rencontre de cette instruction provoque la conversion de l'ensemble E en l'ensemble de ses sous-ensembles logiques d'indicatif I (partition de E) et la création implicite d'un élément courant XI attaché à l'ensemble des parties E/I. L'instruction d'itération logique simple est alors équivalente à l'instruction :

Pour chaque X I de E/I faire m fpc.

Le nom XI de l'élément courant étant inutile dans la description des traitement, celui-ci n'apparaît jamais (sauf pour l'indicatif mineur comme nous le verrons en 8.33.

Une instruction d'itération logique simple est associée à un seul indicatif. Comme les autres instructions d'itération, les instructions d'itération logique simple peuvent être imbriquées sans se chevaucher. L'instruction externe correspond à un indicatif de niveau supérieur à celui de l'instruction interne.

8.32. Déclaration d'un indicatif.

La forme d'expression proposée ci-dessus paraît assez naturelle, mais il convient alors de différencier les instructions d'itération logique simple des instructions "pour chaque" de traitement global d'une file ou d'un ensemble du paragraphe 7.6., et de déterminer l'ensemble sur lequel elles portent.

Pour distinguer les deux types d'instructions, deux solutions sont possibles : donner deux formes différentes ou introduire des déclarations complémentaires précisant qu'un identificateur est un indicatif.

Nous avons choisi la seconde solution car la notion d'indicatif est une notion première en analyse : c'est la définition des inficatifs qui va permettre de déterminer la structure des fichiers d'entrée dans une application et l'organisation des traitements. On préférera en général donner la liste des indicatifs d'un ensemble dans la même classe que la déclaration de l'ensemble : les indicatifs étant liés à l'organisation de l'ensemble ne changeront pas, en général, pendant l'existence de cet ensemble. Cette liste doit donc pouvoir être donnée indépendamment des traitements.

D'autre part, nous définirons en 8.4. des instructions d'itération logique dont l'indicatif peut être attaché à plusieurs files et il semble préférable pour la facilité d'expression, de séparer, dans ce cas, l'association d'un indicatif à un ensemble de la description des traitements.

On pourrait donc songer à définir une déclaration statique d'indicatifs : une instruction d'indicatifs portant sur tous les ensembles ayant I pour indicatif. Mais tous les traitements ne considèrent pas toujours les indicatifs dans le même ordre, ou n'associent pas toujours les mêmes indicatifs à un même ensemble, et il est préférable de faire de la déclaration d'indicatif un objet modifiable.

Une déclaration d'indicatifs s'écrit :

[< indicatif> [<sens>]] [, < indicatif> [<sens>]]* <u>indicatif de</u> <identificateur d'ensemble> .

< indicatif > :: = <identificateur> .

<sens> :: = <u>croi</u> | <u>décroi</u>.

La déclaration d'indicatifs a pour but d'annoncer que I, champ d'une structure déjà déclarée (I a donc un type précisé par ailleurs) est un indicatif pour l'ensemble désigné et qu'il sera donc utilisé pour diviser cet ensemble en sous-ensembles logiques. Un sens peut être associé à chaque indicatif pour indiquer éventuellement dans quel ordre on voudra traiter ces sous-ensembles. Si l'indicatif n'est pas un critère dans le préordre associé à l'ensemble désigné, ou si le sens ne correspond pas, celui-ci sera trié en prenant l'indicatif comme critère.

Une déclaration d'indicatifs peut se trouver comme toute déclaration dans une classe ou dans un module (elle est alors locale à ce module).

Une déclaration d'indicatifs pour un ensemble annule et remplace la déclaration précédente.

La forme simplifiée de la déclaration :

indicatif de <identificateur d'ensemble>

correspond à la déclaration d'une liste vide d'indicatifs en remplacement des indicatifs déjà déclarés pour l'ensemble désigné ; cette suppression est locale à l'unité dans laquelle se trouve cette déclaration.

Ceci permet donc d'attacher un indicatif à un ensemble pour toute la durée de vie de celui-ci (c'est le cas le plus général) en le déclarant dans la même classe que cet ensemble ou dans une classe ayant même durée de vie, ou, au contraire de déclarer localement des indicatifs différents pour le même ensemble.

8.33. Evaluation d'une instruction d'itération logique simple.

Elle commence en s'assurant que l'ensemble est rangé selon un préordre compatible avec les indicatifs utilisés : à la rencontre d'une instruction d'itération logique simple externe, la liste des indicatifs d'un ensemble est examinée : si ces indicatifs sont aussi des critères de tri et si les sens indiqués sont identiques il n'y a pas de tri de l'ensemble, sinon un tri est effectué pour obtenir un ensemble rangé selon le préordre défini par les indicatifs et leur sens associé.

L'ensemble est alors partitionné en ses sous-ensembles logiques comme il a été dit en 8.31.

Pour l'instruction la plus interne, à chaque passage d'un emplacement au suivant la valeur d'un élément courant qui aura dû être associé à l'ensemble est incrémentée.

On trouvera en 24.2. la description détaillée de l'évaluation d'un ensemble d'instructions d'itération logique simple imbriquées.

8.34. Exemple d'utilisation.

Reprenons l'exemple de 8.1.

On écrirait, "fichier" étant l'ensemble des clients de la compagnie :

```
agent croi, client croi indicatif de fichier ;
total général = 0 ;
pour chaque agent faire
          total agent = 0 ;
          pour chaque client faire
                total agent = total agent + prime  fpc ;
          éditer (total agent) ;
          total général = total général + total agent
          fpc ;
éditer (total général) fin module ;
```

Dans cet exemple le traitement décrit ne nécessite pas l'emploi d'un nom d'élément courant : "prime" désigne "prime" de "l'élément courant".

Si cela avait été nécessaire on aurait pu faire précéder ces lignes d'une déclaration d'élément courant.

D'autre part, tous les clients ayant des matricules différents, l'instruction interne aurait pu être un "pour chaque" classique (client ne serait pas un indicatif dans ce cas).

8.4. INSTRUCTION D'ITERATION LOGIQUE MULTIPLE.

8.41. Définition.

Nous disons qu'une instruction d'itération est une instruction d'itération logique multiple, si son indicatif définit des sous-ensembles logiques dans plusieurs ensembles, c'est-à-dire s'il a été déclaré comme indicatif de plusieurs files.

Elle suppose que les différents ensembles attachés aux indicatifs d'un groupe d'itération logique multiple imbriquées sont triés selon des préordres compatibles, c'est-à-dire qu'ils vérifient les conditions suivantes, en prenant la notation :

$e \in E(Ij) \Leftrightarrow$ e est un ensemble ayant Ij comme indicatif

1) $\forall e \in E(Ij)$, Ij est un critère de tri de e ;

2) $\forall e, e' \in E(Ij)$, le sens (croissant ou décroissant) associé à Ij dans e est le même que celui associé à Ij dans e' ;

3) $\forall e, e' \in E(Ij) \cap E(Ik)$, si Ij est un critère de niveau supérieur à Ik dans e alors Ij est de niveau supérieur à Ik dans e'.

Si les préordres ne sont pas compatibles, les tris nécessaires seront entrepris : les critères de tri sont les indicatifs avec l'option d'ordre par défaut croissant, l'ordre des critères étant l'ordre de présence des indicatifs dans les instructions imbriquées.

Si l'utilisateur désire un traitement dans un ordre différent, c'est à lui de faire en sorte que les ensembles soient dans des préordres compatibles.

Une instruction d'itération logique multiple s'écrit exactement comme une instruction d'itération logique simple : les déclarations d'indicatifs permettent de déterminer sur quels ensembles portent les instructions.

On peut considérer que la rencontre d'une instruction d'itération logique multiple (ou simplement de la plus externe dans un groupe d'instructions imbriquées) provoque la conversion des ensembles concernés en l'ensemble obtenu par réunion de ceux-ci (cf. 7.22.), ou encore qu'il y a interclassement de ces ensembles.

8.42. Ensemble directeur.

Dans de nombreux cas, un ou plusieurs ensembles de E (Ij) peuvent jouer un rôle particulier, en ce sens que toute valeur de l'indicatif de l'ensemble résultant doit figurer dans un de ces ensembles. Nous appellerons un tel ensemble "ensemble directeur pour l'indicatif Ij". C'est le cas, par exemple, d'une mise à jour d'un fichier permanent par modifications de ses enregistrements, sans adjonction de nouveaux enregistrements :

l'ensemble à mettre à jour serait un ensemble directeur, ce qui permettrait de détecter la présence d'indicatifs erronés dans l'ensemble des mises à jour. C'est le cas encore d'une édition des éléments d'une liste à partir d'un ensemble de valeurs d'indicatifs d'éléments à lister : le second ensemble est alors un ensemble directeur.

Soit T l'ensemble obtenu par réunion des ensembles de E (Ij). L'ensemble T, privé des éléments dont la valeur de Ij ne figure dans aucun des ensembles directeurs pour Ij, est dit <u>réduit par les ensembles directeurs</u> de Ij.

L'effet d'une instruction d'itération logique multiple d'indicatif Ij est identique à celui de l'instruction d'itération logique simple d'indicatif Ij dans l'ensemble T réduit par les ensembles directeurs de Ij.

L'indication des ensembles directeurs est donnée dans les déclarations d'indicatif plus précisèment les ensembles non directeurs sont explicitement repérés par le mot <u>non directeur</u>. Si ce mot suit le nom de l'ensemble dans une déclaration d'indicatif, cet ensemble n'est directeur pour aucun de ses indicatifs. Si ce mot suit un indicatif, l'ensemble 8.16. désigné n'est pas directeur pour cet indicatif.

On trouvera en 24.3. la description détaillée de l'évaluation d'un ensemble d'instructions d'itération logique multiple imbriquées, dans le cas général et dans un cas particulier.

8.5. EXEMPLES D'UTILISATION.

Exemple 1 : mise à jour d'un fichier sans adjonction.

Soit "stock" un ensemble d'articles indiquant par produit la quantité en stock et "mouvement" un ensemble d'opérations effectuées sur ce stock. On supposera pour simplifier qu'il ne s'agit que d'opérations d'achats. Stock et mouvement ont un seul indicatif, le numéro de produit. Tous les numéros de produits sont supposés exister dans "stock" : "stock" est directeur et non mouvement.

```
module mise à jour simple ; co déclar est une classe contenant la
        description de stock et mouvement oc ;
        utilise déclar ;
        produit croi indicatif de stock ;
        produit croi indicatif de mouvement non directeur ;
        X de stock ; Y de mouvement ;
        nouveau stock type stock ;
        premier Z de nouveau stock ;
        pour chaque produit faire
                si produit de X = produit de Y
                            alors quantité de X = quantité de X +
                                        quantité de Y fsi ;
                Z = X ; Z en Z + 1 fpc ;
fin module ;
```

L'ensemble Z représentera donc le stock "à jour". Si dans l'ensemble mouvement figure un produit n'existant pas dans stock, il sera ignoré et un message d'erreur sera édité.

Exemple 2 : mise à jour d'un fichier avec adjonction.

Reprenons le même problème, mais en supposant cette fois que l'ensemble "mouvement" peut contenir des numéros de produits ne figurant pas dans stock et qu'il s'agisse alors d'éléments à insérer dans le nouveau stock : stock et mouvement sont alors tous deux directeurs.

La boucle s'écrit alors :

```
pour chaque produit faire
        si produit de X = produit de Y
           alors quantité de X = quantité de X +
                   quantité de Y fsi ;
        si produit de X ≤ produit de Y
           alors Z = X sinon Z = Y fsi ;
        Z en Z + 1 fpc ;
```

Exemple 3 : 3 niveaux, 2 fichiers en entrée.

On considère le fichier des ventes réalisées par l'ensemble des représentants d'une société. Les indicatifs de ce fichier sont un code de région, un code de représentant et un code de produit. On suppose le fichier trié par ordre croissant des indicatifs. Un fichier des représentants contient par représentant son nom, adresse, situation familiale et tous les renseignements nécessaires à l'établissement de sa paie.

On veut établir un fichier contenant pour chaque représentant le chiffre d'affaires qu'il a réalisé et le montant de son salaire, avec un récapitulatif par région.

```
module traitement paie ;
      utilise fichiers ; co la classe fichiers contient la déclaration
      des fichiers références, ventes et sorties oc ;
      région, représentant indicatif de références ; X de références
      région représentant, produit indicatif de ventes ; Y de ventes ;
premier Z de sortie ; premier W de récapitulatif ;
            total général = 0 ;
      Pour chaque région faire
            total région = 0 ;
         Pour chaque représentant faire
            total représentant = 0 ;
            Pour chaque produit faire
                  total représentant = total représentant + chiffre
                                                      de Y fpc ;
            nom de Z = nom de X ;
            total de Z = total représentant ;
            constitution Z co c'est une procédure qui construit les
                              éléments du fichier de sortie oc
            Z en Z + 1 ;
            total région = total région + total représentant
            fpc ;
         total de W = total région ;
         région de W = région de X ;
         W en W + 1 ;
         total général = total général + total région ;
         fpc
fin module ;
```

8.6. EXTENSION AU CAS DES FILES.

Les instructions d'itération logique peuvent aisèment être étendues au cas des files, les files traitées sont alors converties en ensemble. Il faut pour cela indiquer le préordre associé à cet ensemble : dans le cas d'une file, un sens doit être associé à chaque indicatif lors de sa déclaration.

REFERENCES DU CHAPITRE 8

HERTSCHUH N. 74 Problèmes d'analyse dans l'utilisation du projet CIVA.
Thèse 3ème cycle. NANCY 1, 1er trimestre 74.