Université de NANCY I
U.E.R. Mathématiques

LE PROJET CIVA

UN SYSTEME DE PROGRAMMATION MODULAIRE

TOME 2

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

Université de NANCY I
U.E.R. Mathématiques

LE PROJET CIVA

UN SYSTEME DE PROGRAMMATION MODULAIRE

TOME 2

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

CHAPITRE 9

LE METALANGAGE CIVA

9.1 INTRODUCTION.

9.2 MACRO ET META-PROCESSEURS.

9.21 Macro-instructions et macroprocesseurs.

9.22 Méta-instructions et métaprocesseurs .

9.3 OBJETS DU METALANGAGE CIVA .

9.31 Validité des méta-identificateurs.

1 Déclaration d'un méta objet .

2 Identificateurs locaux à un métamodule ou une méta-fonction.

9.32 Métavariables .

1 Liste des valeurs.

2 Métafonction associée aux listes.

9.33 Méta-expressions.

1 Méta-expressions arithmétiques.

2 Méta-expressions de chaînes.

3 Méta-expressions booléennes.

9.34 Métafonctions prédéfinies.

9.4 META-INSTRUCTIONS.

9.41 Méta-instruction d'affectation.

9.42 Méta-instruction de saut.

9.43 Méta-instructions d'itération.

1 Méta-instruction '$ pour'.

2 Méta-instruction '$ pour chaque'.

9.44 Méta-instruction conditionnelle.

9.45 Edition de message d'erreur et métacommentaire.

9.5 METAMODULES ET METAFONCTIONS.

9.51 Introduction.

9.52 Métafonctions.

9.53 Métamodules.

9.54 Exemples de métamodules.

1 Métamodule de déclaration.

2 Métamodule d'initialisation.

3 Métamodule de transfert.

4 Transfert par nom.

5 Itération logique de 7.74.

6 Exemple de métafonction.

CHAPITRE 9

LE METALANGAGE CIVA

Tout ce chapitre s'inspire largement de la présentation du métalangage qui a été faite dans Benamghar (73).

9.1 INTRODUCTION.

Nous avons vu, au chapitre 7, que les instructions d'affectations auraient pu être définies avec beaucoup d'autres conventions que celles qui ont été adoptées. De même, au chapitre 8, pour les instructions d'entrée-sortie. Nous avons vu aussi, en 2.31, qu'il est important de pouvoir décrire une action sur un ou plusieurs enregistrements d'un fichier (un tri, une fusion, un éclatement, etc...) sans connaître précisément ce fichier (fichier en paramètre d'un module). Faire la liaison entre le fichier virtuel, connu du module, et le fichier réel pendant l'exécution est particulièrement peu efficace. Il est préférable de faire cette liaison à la compilation d'un appel de module utilisant ce fichier.

Dans ces trois cas, on veut pouvoir, partant du texte paramétré d'un module d'un utilisateur, construire un autre texte, toujours en langage Civa, tenant compte des particularités des paramètres. C'est ce que nous ferons, en utilisant un métalangage. Les métamodules permettront de décrire des actions par une suite d'instructions, de déclarations et de méta-instructions indiquant comment adapter le texte à chaque référence à ce métamodule.

Les métamodules permettent de décrire les actions à entreprendre pendant le métatraitement sous forme de petites unités indépendantes. Ils ont donc un rôle analogue à celui des modules qui décrivent les actions à entreprendre pendant l'exécution du module. Les métamodules sont définis dans une classe d'application - ils sont donc équivalents aux modules -, ou dans une classe ou un module quelconque - ils sont alors à rapprocher des procédures. Déclaré dans une classe c, un métamodule a les mêmes propriétés que tout autre objet déclaré dans cette classe : dans toute classe ou module m tel que $m \hat{\partial} c$, une occurrence

de l'identificateur de ce module désigne le même métamodule, sauf si une déclaration locale a redéfini cet identificateur. Les métamodules peuvent donc communiquer des valeurs par les mêmes mécanismes que ceux vus au chapitre 1.

D'une façon générale, nous avons essayé de conserver le même esprit pour définir le métalangage que pour définir le projet lui-même : modularité , simplicité, clarté en pensant toutefois que le métalangage s'adresse à des utilisateurs privilégiés, particulièrement au fait des problèmes de programmation puisque ce sont ceux qui rédigent des "services" pour les autres. En cas de conflit, nous avons donc plutôt songé à la puissance d'expression qu'à la commodité d'emploi.

Ce chapitre définit le métalangage utilisé et traite des choix qui ont été effectués. Dans une première partie, nous revenons sur les notions de macroprocesseurs et de méta processeurs. Ce ne sont pas des notions nouvelles (Mc Ibroy - 60), (Stratchey - 65), (Leavenwork - 66), (Leroy - 66), (Waite - 67), (Brown -), et nous passerons donc rapidement mais nous essaierons toutefois d'en faire une présentation synthétique et ordonnée.

Nous préciserons ensuite les détails du métalangage adopté : les objets traités, les méta-instructions. Les facilités offertes par un métalangage sont en fait largement indépendantes du langage sur lequel il s'appuie et nous aurions pu faire appel à un métalangage existant (Waite - 67 par exemple), mais nous verrons que l'adaptation du métalangage au langage support offre de nombreux avantages. Nous nous sommes essentiellement inspirés pour la définition du métalangage de Métasymbol (C. I. I).

9.2 MACRO ET META-PROCESSEURS.

9.21 Macro-instructions et macroprocesseurs.

Une macro-instruction (ou simplement une macro), dans un langage de programmation L, est une suite finie d'instructions du langage L, à laquelle on a donné un nom. La donnée d'un nom de macro-instruction et de la suite d'instructions, formant le corps de la macro, constitue la définition de la macro.

Le nom de la macro figurant dans un programme du langage L

constitue une référence à la macro. Une référence à une macro peut en général être écrite aux lieu et place d'une instruction.

Le rôle d'un macro-processeur est de procéder au remplacement de toute référence à une macro par le corps de cette macro-instruction donné dans la définition. Le définition de la macro peut autoriser l'usage de paramètres : lors de la génération d'une version de la macro, toute référence à un paramètre (par un paramètre formel, ou, comme nous le verrons, par un emplacement dans une liste) est remplacée par le paramètre effectif correspondant. Les paramètres effectifs d'une référence à une macro peuvent être désignés dans la définition de la macro :

- par des paramètres formels ; ce sont des identificateurs ; la correspondance entre un paramètre formel et un paramètre effectif est assurée par leur place identique dans la définition et dans la ligne de référence de la macro.

- par un identificateur pour désigner la liste des paramètres effectifs d'une ligne de référence et par le rang du paramètre à désigner.

Ainsi la macroinstruction "intervertir" pourrait être définie en utilisant des paramètres formels par :

Defmacro intervertir (Z, U, V) ;

V = Z ; Z = U ; U = V finmacro ; et, en utilisant la liste des paramètres effectifs, que nous appellerons ℓpe, par :

Defmacro intervertir ;

ℓpe (3) = ℓpe (1) ; ℓpe (1) = ℓpe (2) ; ℓpe (2) = ℓpe (3) finmacro ;

Un macroprocesseur est donc un processeur de construction de texte, opérant uniquement par substitution. Le texte en langage L obtenu peut être traduit.

Pour préciser la notion de macroprocesseur, il peut être utile de revenir sur celle de processeur (ou traducteur) d'un langage. Prenons l'exemple simple d'un assembleur.

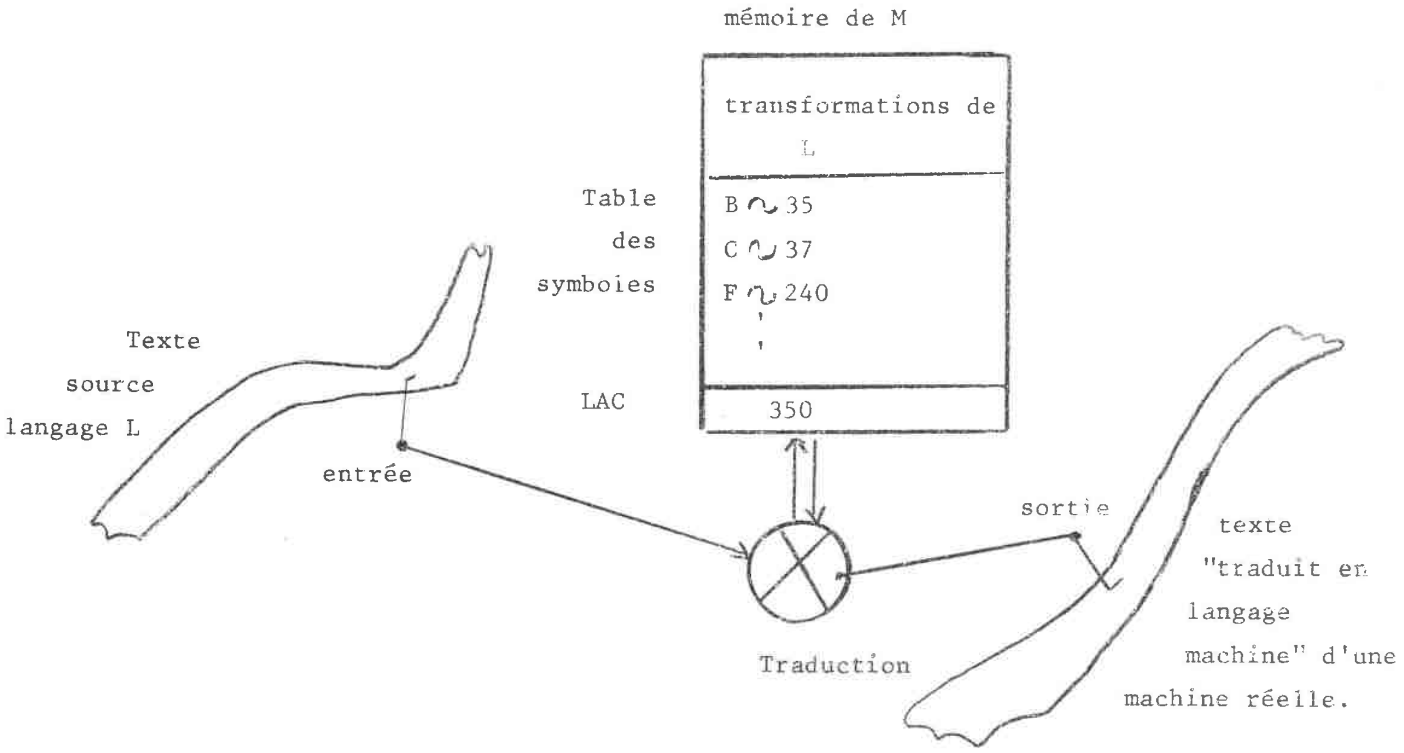
Un assembleur peut être considéré comme une pseudo-machine M possédant :

- une mémoire : elle contient la "table des symboles" et comporte un mot particulier, que nous appellerons LAC, pour "Load Assignment counter", qui indique le

nombre de mots construits depuis le début de l'assemblage, augmenté de 1.

- un ruban d'entrée lu séquentiellement, sur lequel on lit le programme source sous forme d'ordres (instructions et directives d'assemblage).

- un ruban de sortie sur lequel on range le texte produit.



pseudo-machine M

La mémoire et les positions des rubans caractérisent l'état instantané de l'assembleur. A chaque type d'ordre rencontré sur le ruban d'entrée est associé un changement d'état de l'assembleur.

Ainsi le changement d'état associé à la ligne L W, 5␣B serait :

"sortir" X' 3200 <ad (B)>' où ad (B) sera trouvé en mémoire

augmenter LAC de 1.

et pour A RES 1, on aurait

"définir ad (A)" = contenu de LAC

augmenter LAC de 1.

Dans un assembleur, "sortir" ne peut être suivi que d'une seule instruction-machine ou de constantes.

L'introduction de macros permet de lever cette restriction. De plus, les paramètres permettent de "sortir" des textes différents.

La mémoire de la pseudo-machine contient alors les définitions des macro instructions.

La liste des définitions de macros peut être figée à l'écriture du processeur et, conceptuellement, on a toujours affaire à un processeur simple. Dans ce cas, le macrotraitement n'apparaît pas à l'utilisateur : par exemple, il importe peu à celui-ci de savoir si une instruction "pour" d'Algol est traitée comme une macro ou non.

On a pu prévoir, au contraire, de donner à l'utilisateur la possibilité de définir de nouvelles macros dans un langage L en utilisant le langage L lui-même.

On dit alors que l'on a défini un macrolangage. Il suffit pour cela d'introduire dans L deux nouveaux ordres : une directive de début de définition de macro et une directive de fin. Le macro-processeur d'un langage L peut être considéré comme une pseudo-machine ML possédant :

- une mémoire : elle est destinée à recevoir les définitions de macros, et tous les objets du métalangage.
- un ruban d'entrée, qui porte le texte source ; il est lu séquentiellement.
- un ruban de sortie, qui reçoit un texte en langage L.

Le ruban de sortie de ML est le ruban d'entrée de la pseudo-machine M. Les transformations sont les suivantes :

- si la ligne entrée est une instruction de L, la sortir.
- si la ligne entrée est la marque de début de définition de macro, ranger en mémoire tout ce qui suit jusqu'au symbole de fin de macro.
- si la ligne entrée est une référence à une macro déjà rencontrée, (sinon c'est une erreur), sortir les lignes de la définition de cette macro, éventuellement en faisant les substitutions de paramètres.

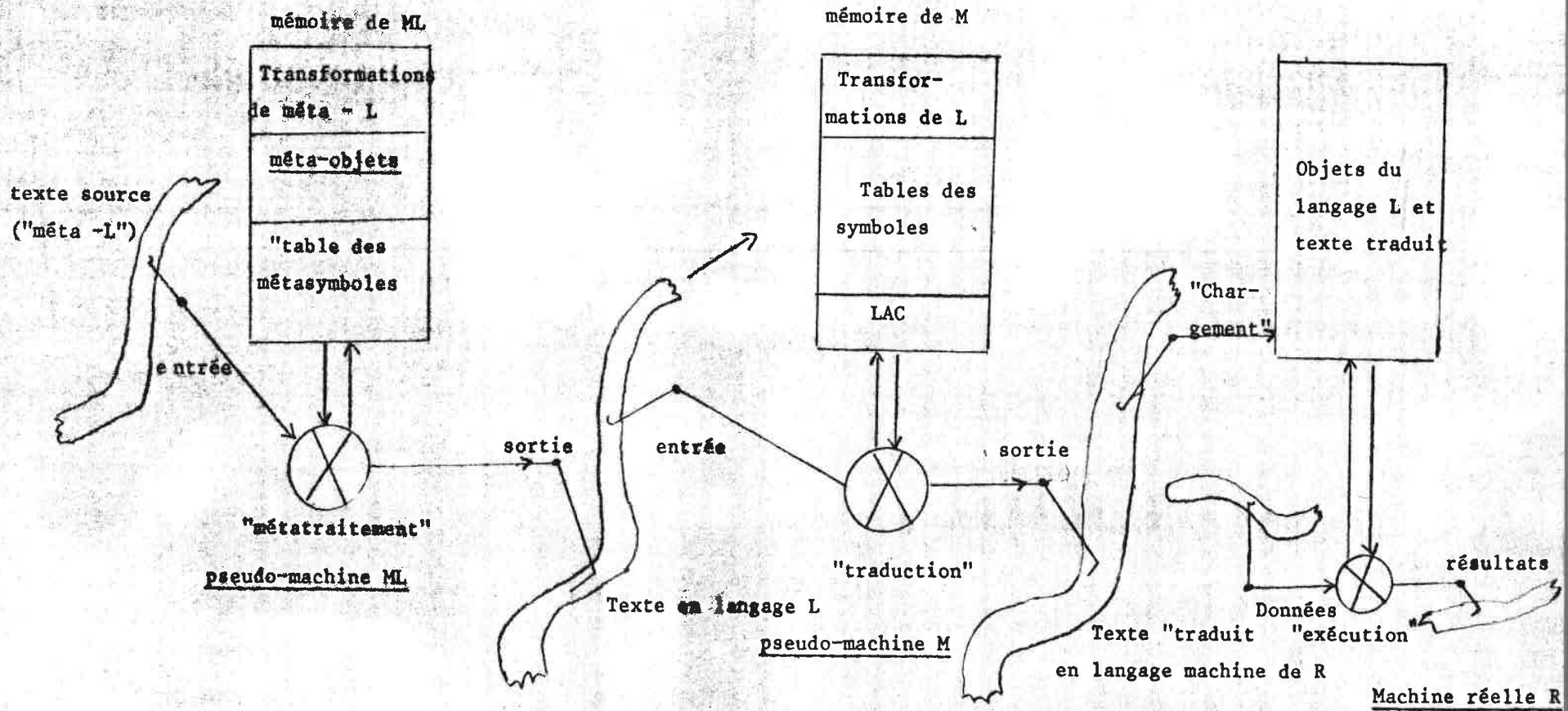


Schéma général de l'exécution des calculs décrits dans un texte de "méta-L".

On notera dans le schéma ci-dessus la différence importante entre les pseudo-machines ML et M : les objets du méta-langage "méta - L" sont rangés dans la mémoire de ML et les métacalculs sont interprétés, alors que les objets du langage L, se trouvent dans la mémoire de la machine réelle R et n'existent que pendant l'exécution du programme obtenu.

9.22 Méta-instructions et métaprocesseurs.

Dans un langage de macroprocesseur, deux lignes de référence à la même macro engendrent en général deux textes semblables.

L'introduction de paramètres est un premier perfectionnement permettant de faire varier les textes obtenus. Cela peut être insuffisant et on peut vouloir générer, ou non, une ligne de texte selon certaines conditions réalisées au cours du macrotraitement. Pour cela, il faut pouvoir décrire des calculs qui seront exécutés pendant la phase de métatraitement. Il suffit donc de généraliser la définition de la pseudo-machine ML en introduisant des transformations portant sur la mémoire de ML. Il faut donc donner à l'utilisateur accès à cette mémoire. D'où la définition de méta-variables, qui désignent des cases de la mémoire du métaprocesseur, et de méta-instructions qui permettent de décrire des calculs sur ces métavariables.

Un métalangage de support L est donc un langage de production de textes du langage L.

Cette distinction du traitement d'un texte source en deux phases - métatraitement et traduction proprement dite-facilite la définition d'un métalangage. Elle peut être utilisée effectivement pour réaliser un métaprocesseur. Dans ce cas, les deux pseudo-machines étant séparées, le métaprocesseur ignore tout du programme qu'il génère, et le métalangage est indépendant du langage L. Il est donc possible de définir un métalangage universel et de l'utiliser pour construire des textes d'un langage quelconque (Waite - 67).

Il peut être intéressant au contraire de ne pas distinguer pratiquement les deux phases de traduction et de métatraitement, c'est-à-dire ne pas passer explicitement par le ruban intermédiaire, car le métaprocesseur a alors accès à la mémoire de la pseudo-machine M et à la "table des symboles". L'introduction de méta-fonctions prédéfinies permet alors de mettre dans la mémoire de ML des informations en provenance de M, c'est-à-dire d'affecter à des métavariables des valeurs caractéristiques des objets du langage. C'est ce qui explique qu'il existe de nombreux méta-lan-

tages (Métaalgol, Métafortran, Métacobol, Métasymbol, etc....). "Méta Civa" est traité de cette façon.

9.3 OBJETS DU METALANGAGE DE CIVA.

Nous venons de voir que le métaprocesseur de Civa est incorporé au traducteur, (au codifieur, plus précisément), ce qui permet à l'utilisateur d'avoir accès dans les calculs du niveau "méta" aux caractéristiques des objets déclarés dans le texte source : type, longueur, etc... Les calculs décrits par le métalangage portent sur des valeurs, chacune d'elles pouvant être :

- un entier ;
- une chaîne de caractère ;
- '$ nul', elle caractérise l'absence de valeur dans un emplacement d'une liste de valeurs ;
- une liste de valeurs, telles que celles qui viennent d'être décrites.

Ces valeurs sont désignées dans un texte par des identificateurs (nous parlerons alors de métavariables), par des méta-expressions, par des métafonctions définies par l'utilisateur, ou encore par des métafonctions prédéfinies - ce sont celles-ci qui permettent l'accès aux caractéristiques des objets du langage.

9.31 Validité des méta-identificateurs.

Tout identificateur et symbole de base du métalangage commence par $ pour distinguer ces objets de ceux du langage.Un identificateur du métalangage commence par le caractère $ suivi d'au plus 8 caractères tous différents de $.

9.311 Déclaration d'un méta-objet.

Une métafonction décrit une façon de calculer une valeur pendant le métatraitement. Cette description constitue sa déclaration (cf. 9.34 et 9.52).

Un métamodule décrit une façon de calculer pendant le métatraitement quel texte produire. Cette description constitue sa déclaration (cf. 9.53).

La déclaration d'une métafonction ou d'un métamodule lui associe un identificateur. La validité de cet identificateur dans un

texte est régie par les mêmes règles que celles vues en 1.4 pour les identificateurs du langage : une occurrence d'un identificateur "id" de métamodule ou de métafonction dans un module ou une classe m est valide si "id" est déclaré dans m ou dans une classe liée à m par une chaîne de "utilise" ou dans une procédure déclarée dans m. Le domaine de validité d'un identificateur de métafonction ou de métamodule est l'ensemble des textes dans lesquels une occurrence de cet identificateur est valide. Comme nous l'avons vu pour les identificateurs du langage, un identificateur du métalangage peut désigner des objets différents dans son domaine de validité : la déclaration d'un métamodule ou d'une méta-fonction $ A dans un métamodule ou une méta-fonction $ B fait de l'objet désigné par $ A un objet local à $ B, et si l'identificateur $ A est valide à l'extérieur de $ B, il désigne un autre objet.

Une métavariable est un objet qui peut contenir une valeur quelconque du métalangage. Il n'y a aucune contrainte restreignant le champ des valeurs possibles d'une métavariable, il est donc inutile de déclarer explicitement les métavariables. L'occurrence d'un identificateur du métalangage qui est rencontrée la première au cours du métatraitement , à gauche d'un symbole :=, ou comme variable contrôlée d'une méta-instruction d'itération "$ pour" ou "$ pour chaque" (cf. 9.43), constitue une déclaration de métavariable.

Le domaine de validité d'un identificateur du métalangage est défini par les mêmes règles que celui d'un identificateur du langage. Une métavariable déclarée dans un métamodule n'est pas un objet local à ce métamodule, son domaine de validité est celui du métamodule plus le texte du métamodule lui-même. Il était assez important de ne pas prévoir de déclaration explicite des métavariables, de ne pas faire d'un méta-identificateur rencontré la première fois dans une déclaration de métamodule un méta-identificateur local à ce métamodule. En effet, la génération conditionnelle de texte nous amène (cf. 9.42) à négliger au cours du métatraitement des portions de texte source et il serait gênant de devoir s'assurer que dans le texte sauté il n'y a pas de déclaration dont il faudrait tenir compte. Ceci permet également aux métamodules d'un même module ou d'une même classe de se communiquer des valeurs de métavariables sans qu'il y ait à se préoccuper de savoir si l'identificateur correspondant a déjà été rencontré ou pas.

En particulier, le module de commande pouvant initialiser une métavariable, il eut été gênant de devoir s'assurer si une métavariable a été déclarée ou non dans le module de commande pour savoir s'il faut la déclarer ou pas.

9.312 Identificateurs locaux à un métamodule ou une métafonction.

Deux métamodules peuvent donc se communiquer la valeur d'une métavariable si une occurrence d'identificateur de cette métavariable est valide dans chacun des métamodules et si elle désigne le même objet. Les objets non locaux aux métamodules permettent donc des communications entre eux. Cependant, ces deux métamodules ont pu être écrits indépendamment l'un de l'autre, s'ils sont déclarés dans deux classes différentes par exemple, et les identificateurs désignant dans chaque métamodule des objets différents doivent pouvoir être choisis de façon quelconque, et éventuellement être les mêmes. D'où la nécessité d'identificateurs ne suivant pas les règles définies ci-dessus et pouvant être locaux à un métamodule, ou une métafonction.

Un identificateur de métavariable ou d'étiquette est local à une métafonction ou un métamodule, s'il en est un paramètre formel ou s'il est déclaré explicitement sous la forme :

$ LOCAL <liste d'identificateurs>

L'identificateur d'un métamodule ou d'une métafonction déclaré dans un autre métamodule ou métafonction est local à celui-ci.

La portée d'un identificateur local à un métamodule ou à une métafonction M est limitée au texte de ce métamodule à l'exception du texte des métamodules éventuellement déclarés dans M et dans lesquels le même identificateur est aussi déclaré local. En particulier, elle ne comprend pas le texte des métamodules appelés dans M.

La durée de vie de l'objet désigné par un identificateur local à un métamodule ou à une métafonction est la durée d'élaboration de ce métamodule.

9.32 Métavariables.

Une métavariable est un objet dont la durée de vie n'excède pas la durée du métatraitement. Elle contient une valeur quelconque du métalangage : entier, chaîne de caractères, "$ nul" ou liste de

valeurs.

La valeur contenue par une métavariable peut être modifiée par une méta-instruction d'affectation. Ainsi, la méta-instruction d'affectation $ A : = $ B permet de donner à la métavariable $ A la valeur contenue par $ B.

L'identificateur d'une métavariable peut apparaître n'importe où dans un texte Civa :

- s'il apparaît dans une instruction du langage, il lui est substitué la valeur de la métavariable sous forme de chaîne de caractères. Si la valeur actuelle de la métavariable est une chaîne de caractères, c'est celle-ci qui est recopiée dans le texte produit. Si la valeur actuelle de la métavariable est un entier, il est converti en une chaîne de 8 caractères éventuellement précédée d'un signe ; les zéros non significatifs sont remplacés par des blancs.

- s'il apparaît dans une méta-instruction, c'est un constituant d'une méta-expression. Nous traiterons ce cas en 9.33.

Si la métavariable contient une liste de valeurs, on pourra désigner un élément de cette liste par le couple (identificateur de la métavariable, rang de l'élément dans la liste). Nous parlerons alors de métavariable indicée. Elle s'écrit $ L ($ I).

9.321 Listes des valeurs.

Une liste est un ensemble ordonné de valeurs du métalangage. Si ces valeurs sont uniquement des entiers, chaînes de caractères ou "$ nul", la liste est dite linéaire.

1, 3, ABCD, , 9

A, B, C, D, E

sont deux listes linéaires de valeurs. La notation , , indique que l'élément correspondant a pour valeur "$ nul". Dans une méta-expression arithmétique, "$ nul" est la valeur zéro ; dans les autres cas, "$ nul" est la valeur "vide". "$ nul" est donc différent d'une métavariable à valeur zéro.

Si la taille d'une liste $ L est n, toute métavariable indicée $ L (m) a pour valeur "$ nul" si m > n.

Un élément de liste de valeurs peut également être une liste de valeurs. La première liste est dite non linéaire. Si l'élément

$ L ($ I) est une liste de valeurs, la métavariable indicée $ L ($ I, $ J) désigne la $ Jème valeur de cette liste. Les parenthèses sont utilisées pour délimiter les listes constituantes.

Exemples :

1, 2, 3 est une liste linéaire de trois valeurs.

(1, 2, 3) est une liste non linéaire à un seul élément. Celui-ci est la liste linéaire 1, 2, 3.

1, , (5, 7, (9, 3)), 8 est une liste non linéaire à 4 éléments qui sont 1, $ nul, une liste L et S, la liste L est une liste non linéaire à 3 éléments 5, 7 et une liste linéaire à 2 éléments 9, 3. Si $ B a pour valeur la liste ci-dessous, $ B (3, 3, 1) a pour valeur 9.

9.322 Métafonction associée aux listes.

Nous définissons une métafonction (cf. 9.34) dite prédéfinie, $NUM ($ L) qui a pour valeur le nombre d'éléments non nuls et d'éléments nuls explicites de la liste $ L. Le nom spécifié peut être celui d'une liste ou d'un élément d'une liste.

Exemple :

$ S : = (A, (B, ((C, D))) ; $NUM ($ S) vaut 2. $ S (1) est A (une paire de parenthèses inutiles) et $ S (2) est B, ((C, D)).
$NUM ($ S (2)) vaut 2. $ NUM ($ S (2,1) vaut 1 $NUM ($S (2,1,1 aussi, mais $NUM ($ (2, 1, 2) vaut 0. S (2,2) est (C,D) et $NUM ($ S (2, 2, 1))vaut 2.

9.33 Méta-expressions.

9.331 Méta-expressions arithmétiques.

Ce sont des expressions arithmétiques dont les opérandes sont des métavariables, des constantes entières, des chaînes de caractères numériques désignant une valeur entière, des expressions de chaîne ou des références à des métafonctions.

Les opérateurs autorisés sont +, -, *, /, opérandes et résultats étant toujours des valeurs entières. Une chaîne de carac-

tères numériques, signée ou non, sera d'abord convertie en entier. Si un opérande est une chaîne de caractères non numériques, il y a édition d'un message d'erreur.

Une métaexpression arithmétique est interprétée au moment de sa rencontre par le métaprocesseur. Sa valeur est un entier. Dans le cas où la métaexpression figure dans une instruction du langage, sa valeur est convertie sous forme de chaîne d'au plus 8 caractères précédée éventuellement d'un signe, représentant la valeur décimale de l'entier.

9.332 Métaexpressions de chaînes.

L'opérateur de concaténation, noté "." dans une métaexpression, permet de construire des métaexpressions de chaînes. Une valeur de type chaîne s'écrit sous la forme d'une suite de caractères entre les symboles ' ou, simplement d'une suite de caractères alphanumériques.

L'expression 'ABCD'. 'EF' a pour valeur la chaîne 'ABCDEF'.

Si les métavariables A et B ont respectivement pour valeur : 'XZP' et 'LEV', l'expression A.B a pour valeur 'XZPLEV'.

Les opérandes de "." sont des chaînes de caractères ou des identificateurs de métavariables ou des entiers, les entiers étant convertis sous forme de chaîne de caractères avant la concaténation (cf. 9.31). Un identificateur de métavariable est remplacé par sa valeur.

Exemple :

$ Z : = 257 ;

$ T : = 'A␣VAUT␣' ;

$ MESSAGE : = $ Z.$ T a pour valeur 'A␣VAUT␣␣␣␣␣257'

Remarquons que 'ABC' est la notation de la valeur de chaîne ABC, et que lors d'une substitution, $ MESSAGE sera remplacé par A␣VAUT␣␣␣␣␣␣ 257.

9.333 Méta-expressions booléennes.

Ce sont des expression booléennes (cf. 6.24) pour lesquelles les primaires booléens sont uniquement des expressions booléennes entre parenthèses ou des relations dont les opérandes sont des méta-expressions arithmétiques. Il n'existe pas de métavariable à valeur booléenne. La valeur d'une expression booléenne est l'entier 1 si elle est vraie, et l'entier 0 si elle n'est pas vérifiée. L'opérateur de relation = = désigne l'égalité.

$ A >342

$ A >342 $et $ B = = 0 sont des expressions booléennes

Pour les opérations de comparaison entre chaînes, c'est l'ordre alphabétique des caractères qui est pris en compte. Ainsi, dans

$ A = = $ B > 'ABC'

si $ B vaut 'ZZZ', alors $ A vaut 1 et si $ B vaut 'AAB', $ A vaut 0. Les priorités entre opérateurs sont les mêmes que pour les expressions arithmétiques et booléennes (Cf. 6.24).

9.34 Métafonctions prédéfinies.

Nous verrons au paragraphe 9.6 que l'utilisateur peut définir lui-même des métafonctions. Nous n'étudierons ici que les métafonctions prédéfinies, c'est-à-dire celles dont la définition est incorporée au métaprocesseur. Il s'agit, en plus de la fonction $ NUM vue en 9.323, de fonctions permettant de caractériser des éléments du langage (essentiellement les structures). Elles permettent d'avoir accès aux renseignements constitués par le codifieur : tables des identificateurs, table de descripteurs, type d'un identificateur, taille d'une file, lien horizontal ou vertical d'un champ d'une structure, adresse d'un élément dans la zone statique, etc...On en trouvera la liste ci-dessous.

$ Type (X) a pour valeur le code du type de la variable X du langage. C'est un entier. Elle est surtout utilisée dans des tests d'égalité (tests de conformité).L'utilisateur n'a pas à connaître effectivement les codes des différents types : des identificateurs de métavariables sont définis dans la classe "système". Ce sont : typent, typréel, typbool, typdec, typcar, typff (file fixe), typfm (file de taille maximum), typfv (file de taille variable), typstr (structure),

typfich (fichier).

$ Nbre car (F) où F est une file. Cette fonction a pour valeur le nombre de caractères qui ont été pris dans une file émettrice de type caractère, lors de la dernière instruction d'affectation ayant cette file en position émettrice.

$ Taille (X) a pour valeur le nombre d'éléments de la file de taille fixe. C'est un entier. Elle vaut 1 si X est un objet de type simple, et 0 dans les autres cas.

$ Max (A, B) ; $ Min (a, b) ont pour valeur, respectivement, le maximum et le minimum des deux métavariables A et B. Ces fonctions ne sont définies que si A et B sont de même type.

$ Borne inf (X, I) a pour valeur celle de la borne inférieure du Ième indice dans la file X.

$ Borne sup (X, I) a pour valeur celle de la borne supérieure du Ième indice dans la file X. Borne sup n'est définie que dans le cas des files de taille fixe ou bornée.

$ AF (liste d'indices) est une métafonction permettant l'accès dans la liste des paramètres effectifs d'un métamodule ou d'une métafonction. Elle n'est définie que dans leur corps. $ AF spécifie qu'il s'agit de la liste d'arguments effectifs et liste d'indice spécifie de quel élément il s'agit. En cas d'absence de la liste d'indices, la fonction référence la liste d'arguments dans sa totalité.

L'utilisation de cette fonction appelle une remarque dans le cas ou un élément (par exemple le 1er) de la liste de paramètres effectifs est lui-même une liste. $ AF (1,2) ne désigne pas le 2ème élément de cette liste, mais dans ce cas serait nul. Nous en verrons un exemple en 9.542.

$ Adr (X) a pour valeur l'adresse relative de l'objet X dans la zone statique (cf. chap. 17). Cette fonction sera utilisée essentiellement en mode "mise au point" (cf. Chap. 10). X est un identificateur de type quelconque, ou une constante, déjà rencontré par le codifieur.

$ Rgsymbol (A, X) a pour valeur l'entier égal au rang du symbole désigné par X dans la liste A. X peut être une métavariable ayant pour valeur une chaîne de caractères, ou une dénotation de chaîne de caractères. Rgsymbol prend la valeur 0 si le symbole X ne figure pas dans A.

Exemple :

$ A := 1, '12', 'ABCD', '73', 'TRUC' ;

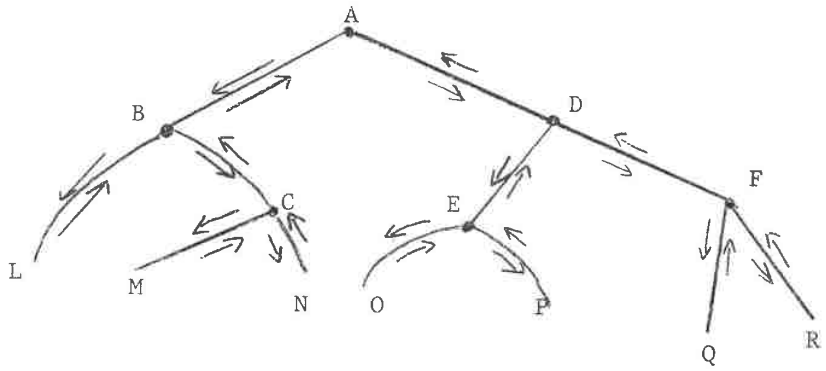
$ J := 'ABCD' ;

$ Rgsymbol (A, '73') vaut 4.

$ Rgsymbol (A, J) vaut 3.

$ Rgsymbol (A, 73) n'est pas défini.

Les métafonctions suivantes concernent toutes les structures. S et X désignent une chaîne de caractères (métavariable ou dénotation de chaîne) et I un entier. Nous nous appuierons pour les définir sur l'exemple de la structure A suivante :



$ Lh (S, Z), ou "lien horizontal" de Z dans la struture S. Soit G le prédécesseur de Z dans la structure S ; si G a un autre successeur après Z, soit H, Lh (S, Z) a pour valeur le nom du champ H, et "nul" sinon

$ Lh ('A', 'B') vaut 'D' ;

$ Lh ('A' ,'D') vaut "nul".

$ Lv (S, Z) ou "lien vertical" de Z dans la structure S. Si Z possède un successeur, Lv (S, Z) a pour valeur le nom de celui-ci, et "nul" sinon.

$ Lv ('A', 'B') vaut L, $ Lv ('A', 'C') vaut M et $ Lv ('A', 'M') vaut "nul".

$ Nbre feuil (S) a pour valeur le nombre de feuilles de la struture S.

$ Nbre feuil ('A') = 7

$ Nom feuil (S, I) a pour valeur une chaîne de caractères, c'est le nom de la Ième feuille de la struture S, dans l'ordre, de gauche à droite, du mot des feuilles de S. Si I > $ nbre feuil (S), $ nom feuil (S, I) a pour valeur "nul".

$ nom feuil ('A', 3) vaut N et $ nom feuille ('A', 9) vaut "nul".

$ Nbre champ(S) a pour valeur le nombre total de champ de la structure S (y compris la racine S)

$ Nbre champ ('A') vaut 13.

$ Rang cham (S, X) a pour valeur un entier égal au rang du champ de nom X dans la structure S. Ce rang est défini par l'algorithme d'analyse de l'arborescence associée à S. Il utilise une pile. les flèches sur le schéma ci-dessus indiquent comment il procède.

$ Rang cham ('A', 'D') a donc pour valeur 7.

$ Nom champ (S, I) a pour valeur la chaîne de caractères égale au nom du champ de rang I de la strututre S, s'il en existe un, et "nul" sinon.

$ Nom champ ('A', 7) vaut 'D', $ Nom champ (A, 15) vaut "nul".

$ Nom pred (S, X) a pour valeur le nom du prédecesseur immédiat (père) de X dans la structure S.

$ Nom pied ('A', 'P') vaut 'E', $ Nom pied ('A', 'A') vaut "nul".

$ Nbre pred (S, X) a pour valeur le nombre de prédecesseurs (ancêtres) de X, y compris la racine.

$ Nbre pred (A, P) vaut 3.

$ Mot feuil (S) a pour valeur une liste consitutée des feuilles de la structure (mot des feuilles).

$ Mot feuil (A) a pour valeur la liste de valeurs :

L, M, N, O, P, Q , R.

9.4 METAINSTRUCTIONS.

Ce sont elles qui permettent de décrire les calculs à effectuer pendant le métatraitement : elles sont donc interprétées. Elles s'écrivent comme les instructions classiques des langages de programmation. Une métainstruction est suivie d'un point-virgule, ou d'un symbole de fin : $ finmod , $ finfonc, $ fpc, $ finpour , fpc, finmodule, finclasse.

9.41 Métainstruction d'affectation.

<métainstruction d'affectation> ::= [<étiquette> :] <partie gauche> : = <partie droite>.

<partie gauche> ::= <identificateur de métavariable> | <métavariable indicée>.

<partie droite> ::= <métaexpression> [, <métaexpression>]*.

Une métainstruction d'affectation ne conduit à aucune génération de texte, elle permet d'attribuer une valeur à une métavariable.

Chaque métaexpression de la partie droite est d'abord évaluée : chaque identificateur de métavariable est remplacé par la valeur de celle-ci et les opérations décrites sont effectuées.

Un même identificateur peut donc désigner successivement un entier, une chaîne de caractères, une liste linéaire de valeurs, etc...

Exemples :

```
$ A : = 1 ;        $ A est une métavariable à valeur entière.
$ A : = 1, 2, 7 ;  Sa valeur est maintenant une liste d'entiers.
$ A : = 1, $ A, 7 ; $ A a pour valeur une liste non linéaire
                     de valeurs. $ A (2) vaut 1, 2, 7.
$ A : = ABCD ;
$ A : = 12, 3, 5 ;
$ A (3) : = 1, 2, 3 ; $ A a alors pour valeur la liste 12, 3,
                       (1, 2, 3).
```

9.42 Métainstruction de saut.

La métainstruction de saut demande au métaprocesseur de déplacer le ruban d'entrée de la pseudo-machine ML (cf. 9.21). Ce déplacement peut avoir lieu vers l'avant ou vers l'arrière. Cette instruction pourrait ne pas figurer dans le langage (cf. Introduction), on utiliserait essentiellement les appels de métamodules et les instructions conditionnelles, qu'il faudrait sans doute compléter en introduisant par exemple des tables de décisions dans le métalangage, mais il nous a semblé préférable d'introduire la métainstruction de saut. Le saut

est en effet naturel au niveau du métalangage ; il est utilisé la plupart du temps pour indiquer qu'une partie du texte source doit être négligée(saut "en avant"). D'autre part l'absence de saut conduirait l'utilisateur à structurer son texte de façon différente de celle que lui suggère le programme à générer, uniquement pour des raisons d'écriture du métalangage, ce qui ne nous semble pas une bonne chose.

Cette méta-instruction s'écrit [<étiquette>] : $ allera <étiquette> .

Lorsque le métaprocesseur rencontre cette méta-instruction, il poursuit son analyse à l'endroit repéré par l'étiquette qui suit "$ allera". Le texte écrit entre la méta-instruction de saut et l'instruction repérée est ignoré .

Quand le saut est effectué vers la fin du texte, la partie sautée est quand même analysée pour tenir compte des éventuels symboles de fin, si l'on est dans une méta-instruction qui en comporte ($ fsi, $ fpc, $ fin pour).

Depuis l'extérieur d'une méta-instruction conditionnelle ou d'itération, d'un méta-module ou d'une métafonction, on ne peut aller à l'intérieur qu'en allant au début de cette méta-instruction.

Les étiquettes n'ont d'existence que pendant le métatraitement. Une étiquette s'écrit comme un identificateur du métalangage (cf. 9.31). Une étiquette précède une instruction ou une déclaration quelconque du langage ou du métalangage. Elle peut également précéder certains symboles de fin qui seront précisés.

Une étiquette est locale à l'élément du langage ou du métalangage qui la contient : module, classe, métamodule, métafonction ou méta-instruction d'itération. Ainsi, il n'est pas possible de se rendre par une méta-instructionde saut depuis l'extérieur vers l'intérieur d'une métafonction, d'un métamodule, ou d'une méta-itération sans passer par son début.

Dans le cas d'une étiquette définie dans une métafonction ou un métamodule M, elle lui est locale et toute occurrence de cette étiquette est valide dans M sauf dans le texte des métafonctions et métamodules locaux à M. Ainsi, il n'est pas possible non plus de se rendre depuis l'intérieur d'un métamodule à l'extérieur, ce qui serait un effet de bord désastreux d'une référence à un métamodule.

9.43 Méta-instructions d'itération.

Nous définissons deux types d'instruction d'itération : une forme analogue à l'instruction "pour" et une forme analogue à l'instruction "pour chaque" associée cette fois à une liste de métavariables.

9.431 Métainstruction "Pour".

[<étiquette> :] $ pour <métavariable> : = <e1> , <e2> [, <e3>]
$ faire
<I 1> ; <I 2> ;.....<I n> [<étiquette :] $ fin pour

e1, e2, e3 sont des expressions à valeur entière ; I_1, I_2.... I_n sont des instructions ou des méta-instructions. Plusieurs méta-instructions d'itération peuvent être imbriquées, sans se chevaucher.

Evaluation :

e1, e2, e3 sont d'abord évaluées dans le mode entier. e1 est la valeur initiale :

- la métavariable à gauche de : = (appelons la MV) reçoit la valeur de e1 ;

(1) - le texte source étant écrit entre $ faire et $ fin pour est analysé par le métaprocesseur. MV reçoit la valeur de MV + e3.

- Si MV > e2, le métaprocesseur poursuit son traitement après $ Fin pour ; sinon on recommence en (1).

Notons que e1, e2, e3 sont évaluées une fois pour toutes à l'entrée de la boucle et que leurs valeurs ne peuvent pas être modifiées.

Si e3 est omis, sa valeur par défaut est 1.

Exemple 1 :

$ A : = ' XY', 'Z', 'T', 'U','V' ;
$ B : = 1, 342, 750, 2312, 17 ;
$ Pour $ I : = 1, 5 $ faire $ A ($ I) = $ B ($ I) ; $ fin pour ;
Cette séquence conduit à la génération du texte suivant :
XY = 1 ; Z = 342 ; T = 750 ; U = 2312 ; V = 17 ;

Cet exemple amène deux remarques :

Le point-virgule qui suit $ B ($ I) est nécessaire, si l'on veut que les instructions générées soient effectivement séparées

par des points-virgules.

La nécessité de deux symboles d'affectation différents pour l'instruction et la méta-instruction d'affectation apparaît bien ici : l'écriture

$ Pour $ I : = 1, 5 $ faire $ A ($ I) : = $ B ($ I) ; $ fin pour

ne conduit à aucune génération de texte ; elle est équivalente à la suite de méta-affectations $ A (1) : = $ B (1) ; $ A (2) : = $ B (2) ; $ A (3) : = $ B (3) ; $ A (4) : = $B (4) ; $ A (5) : = $ B (5).

Exemple 2 :

Si F est une file de borne inférieure 1 et de taille fixe, et S une structure :

$ Pour $I: = 1, $ min ($ bornesup (F, 1), $ nbrefeuil (S) $ faire

F ($ I) = $ nomchamp (S, $ I) ; $ fin pour ;

sera remplacé par le texte suivant :

F (1) = A ; F (2) = B ; F (3) = C ; F (4) = D ; F (5) = E ;

en supposant que F a au moins 5 éléments et que les champs de la structure S se nomment A, B, C, D, E.

9.432 Métainstruction $ pour chaque.

Elle s'écrit :

[<étiquette> :] $ pour chaque <méta él^{t} courant> $ de <métavariable> [(<métavariable>)] $ faire I_1 ; I_2 ..... I_n $ fpc ;

Ainsi $ pour chaque $ X $ de $ L (S I) $ faire I $ fpc ; est une méta-instruction "$ pour chaque". Le méta-élément $ X courant désigne successivement chaque valeur de la liste de valeurs désignée par $ L. $ I reçoit la valeur du rang de cet élément.

- Le métaprocesseur donne au méta-élément courant la valeur du premier élément de la liste et la valeur 1 à son rang, si celui-ci est demandé.

(1) - L'analyse est reprise après $ faire.

- A la rencontre de $ fpc, la métavariable désignant le rang

est incrémentée de 1. Si sa valeur est inférieure ou égale à $ NUM (<métaliste>), on recommence en (1) et le métatraitement courant prend la valeur de l'élément suivant de la liste des valeurs.

Sinon l'analyse se poursuit après $ fpc.

Exemple :

```
$ L : = A, B, C, D, E ; $ L 2 : = H, I, J, K, L
$ Pour chaque $ X   $ de $ L ($ I) $ faire  $ X = $ L 2 ($I) ;
$ fpc ;
```

permet de générer

```
A = H ; B = I ; C = J ; D = K ; E = L ;
```

9.44 Méta instruction conditionnelle.

Elle s'écrit :

[<étiquette>:] $ SI <méta-expression> $ Alors <texte> [$ Sinon <texte>]

[<Etiquette>] $ fin si.

<méta-expression> désigne une méta-expression arithmétique, ou booléenne, ou une méta-expression de chaîne ; elle est évaluée en mode entier.

<Texte> désigne une suite quelconque d'instructions, de méta-instructions ou de métavariables (une métavariable pouvant avoir pour valeur une chaîne de caractères représentant une instruction).

Si la valeur de la méta-expression est différente de zéro, le métaprocesseur poursuit son travail en analysant d'abord le texte qui suit "$ Alors" ; si celui-ci ne contient pas de méta-instruction de saut, l'analyse se poursuit par le texte qui suit "$ Sinon".

Si la valeur de la méta-expression est zéro, le métaprocesseur poursuit son travail en analysant le texte à partir de "$ Sinon".

Les textes écrits après "$ Alors et "$ Sinon" sont quelconques ; en particulier ils peuvent contenir eux-mêmes une méta-instruction conditionnelle.

Exemple :

```
$ A : = 'A = A + 1 ; ' ;
$ B : = 'B = B + 1 ; ' ;
$ Si $ C >=3  $ Alors $ A $ B ; Allera $ E $ Sinon $ B $ A
$ E : C = 3 ;
```

Lorsque $ C est supérieur ou égal à 3 (la méta-expression $ C >=3 vaut alors 1), le texte généré est le suivant :

```
A = A + 1 ; B = B + 1 ; C = 3 ;
```

tandis que dans le cas contraire, on obtient :

```
B = B + 1 ; A = A + 1 ; A = A + 1 ; C = 3
```

9.45 Edition d'un message d'erreur et commentaires.

La méta-instruction $ erreur (< méta-expression >, < méta-expression de chaîne >) permet l'édition, pendant le métatraitement, d'un message à l'utilisateur. La valeur de la méta-expression est utilisée comme niveau de sévérité de l'erreur (cf. chap. 16). Le métatraitement n'est pas interrompu.

Des commentaires peuvent être introduits dans un texte source. Si ceux-ci ne doivent pas figurer dans une édition du texte Civa "généré", ils seront écrits entre $ CO et $ OC.

Hormis les définitions de métamodules et métafonctions vues dans paragraphe suivant, tout ce qui n'a pas été défini jusqu'à présent est censé appartenir au langage et est recopié tel quel dans le texte généré.

9.5 METAMODULES ET METAFONCTIONS.

9.51 Introduction.

Avec les métavariables, méta-expressions, méta-instructions définies précédemment, il est possible d'écrire des programmes et des sous-programmes du métalangage de Civa. Comme d'habitude, un sous-programme est une séquence d'instructions (donc ici de méta-instructions possédant un nom, par lequel on peut en demander l'exécution. Nous distinguerons les sous-programmes délivrant une valeur - une métafonction est un

sous-programme fermé du métalangage délivrant une valeur pendant le métatraitement - et les autres sous-programmes ou métamodules, qui sont des sous-programmes ouverts ne délivrant pas de valeur.

Une métafonction ne conduit à aucune génération de texte, alors qu'un métamodule peut le faire.

Un métamodule constitue donc une "macro-instruction"du langage de Civa. Nous conserverons cependant le préfixe "méta" pour bien indiquer que son traitement n'est pas limité à des substitutions, il comporte également des évaluations.

Des métafonctions prédéfinies ont été jointes au métaprocesseur comme nous l'avons vu en 9.34. Toutes les autres métafonctions (et les métamodules) doivent avoir été définies par l'utilisateur.

Une métafonction, ou un métamodule, est déclarée dans une classe ou dans un module. Si elle l'est dans une classe c, elle peut être référencée dans tout module ou classe qui utilise c. C'est que la directive "utilise" est considérée également comme un élément du métalangage, avec le même sens que pour le langage.

On peut donc résumer le rôle de la directive "utilise" en disant que le métaprocesseur traite "utilise C;" en lui substituant le texte de la classe C ; de plus, un identificateur déclaré dans C désigne le même objet dans tout module qui "utilise" C et qui ne redéclare pas le même identificateur.

Une métafonction ou un métamodule peut également être déclarée dans une autre métafonction ou métamodule : elle lui est alors locale.

9.52 Métafonctions.

Une référence à une méta-fonction est constituée du nom de la métafonction (il commence par $) suivi éventuellement d'une liste, entre parenthèses, de paramètres effectifs séparés par des virgules. Un paramètre effectif est un identificateur du langage, une chaîne de caractères, une constante numérique, une méta-expression ou identificateur du métalangage. Dans le cas d'un identificateur du langage (chaîne de caractères), il pourra ne pas être écrit entre ' '.
Une référence à une métafonction peut apparaître n'importe où dans un

texte si sa valeur est une chaîne de caractères (il faudra bien sûr que la chaîne générée soit compatible avec le contexte). Si sa valeur est numérique elle ne peut apparaître que dans une méta-expression ou dans une expression arithmétique (sa valeur est alors générée sous forme de chaîne de caractères).

Lorsque le métaprocesseur rencontre une référence à une métafonction, celle-ci est évaluée en l'appliquant aux paramètres effectifs de la référence. La valeur obtenue remplace la référence.

La définition d'une métafonction est un texte qui commence par le symbole $ FONC, suivi du nom de la fonction (c'est un identificateur du métalangage), suivi éventuellement d'une liste,entre parenthèses, de paramètres formels séparés par des virgules. Un point-virgule sépare cette partie du corps de la métafonction. Les paramètres formels sont des identificateurs du métalangage.

Le corps de la métafonction est une suite composée uniquement de méta-instructions séparées par des points virgules se terminant par $ FIN FONC. Au moins l'une d'entre elles est une métainstruction d'affectation ayant en partie gauche l'identificateur de la métafonction.

Au cours de l'évaluation de la fonction, l'une au moins, de ces méta-instructions d'affectation doit avoir été exécutée avant la rencontre de $ FIN FONC.

La rencontre de $ FIN FONC entraîne le remplacement de la référence à la métafonction par la valeur qui vient d'être calculée. $ FIN FONC peut être étiqueté .

Les paramètres effectifs peuvent être désignés dans ce corps par l'intermédiaire de paramètres formels : toute occurrence d'un paramètre formel est remplacée par le nom du paramètre effectif qui a le même rang, dans la liste des paramètres effectifs, que lui dans la liste des paramètres formels. Ils peuvent également être désignés par $AF (Y), où Y est une méta-expression et $AF est une métafonction prédéfinie qui a été vue en 9.34. Elle permet l'accès aux éléments de la liste des paramètres effectifs de la référence en cours d'évaluation. $AF peut également être considéré comme la liste des paramètres effectifs.

9.53 Métamodules.

Une référence à un métamodule est une méta-instruction . Elle est donc suivie de ; ou d'un symbole de fin méta-instruction. Elle peut comporter des paramètres : les règles sont les mêmes que pour les méta-fonctions. Une référence à un méta-module s'écrit "$ nom de méta-module" donc " $ $ A" pour le métamodule $ A.(1)

A la rencontre d'une référence à un métamodule, le métaprocesseur la remplace par le corps du métamodule concerné. Il poursuit son analyse au début de ce corps, après avoir éventuellement substitué des paramètres formels aux paramètres effectifs.

La définition d'un métamodule commence par le symbole $ MOD, suivi de son nom, des paramètres formels éventuels, du corps de métamodule, et du symbole $ FINMOD. Le corps d'un métamodule est formé d'instructions et d méta-instructions séparées par des points-virgules. Une référence à un métamodule peut donc donner lieu à la production d'un texte.

9.54 Exemples de métamodules.

Des exemples de métamodules ont été présentés au paragraphe 2.43 et nous en présentons d'autres aux chapitres 11 et 13.

(1) Cette règle, un peu désagréable, est imposée par la nécessité de reconnaître, lors de l'interprétation du langage de commande, pendant l'entrée d'un texte, la présence des références à des métamodules. En effet, un méta-module pouvant contenir une directive "utilise" ou un appel de module, cette référence peut modifier le graphe (E, U) ou le graphe (E, R).

9.541 Métamodule de déclaration

Le métamodule $declar permet de générer un ensemble fixe de déclarations.

D'une référence à l'autre, les identificateurs peuvent changer.

```
$ mod  $ déclar ($ A, $ B, $ C, $ D, $ E, $ F) ;
    $ B entier ;
    $ si  $ A ≠ 0  $ alors  $ C file max (50) car ; $ E file max
    (75) car ;
                $ sinon  $ C, $ E file (50) car ; $ fsi ;
$ D, $ F réel ; $ fin mod
```

Exemple de référence :

```
.....; $ AR : = 3 ;.....; $ declar ($ AR, I, FM1, R1,
FM2, R2)
```

Texte équivalent :

I entier ;	entier étant un élément du langage, la ligne $ B entier ; appartient au langage. La métavariable $ B est remplacée par la valeur du paramètre effectif qui est la chaîne de caractères 'I'.
FM1 file max (50) car ;	$ A est remplacé par le paramètre effectif $ AR. La relation $ AR ≠ 0 est évaluée : elle est vraie et vaut donc 1. L'analyse se poursuit donc après $ Alors. $ C est remplacé par FM1.
FM2 file max (75) car ;	$ E est remplacé par la valeur du paramètre effectif.
R1, R2 réel ;	L'analyse précédente s'est arrêtée à $ Sinon pour reprendre après $ Finsi.

9.542 Métamodule d'initialisation.

```
$ mod  $ init ;
$ LOCAL  $ I, $ PE ;
$ PE : = $ AF (1) ;
$ Pour Chaque  $ X $ de $ PE  $ faire
$ X  =  $ AF (2) ; $ fpc  $ fin mod ;
```

$ init n'a pas de paramètres formels, mais deux paramètres effectifs. Le premier est un nom de métaliste dont les éléments sont des identificateurs à initialiser. Le second désigne la valeur initiale à leur donner. La méta-instruction $ PE : = $ AF (1) est nécessaire, pour pouvoir accéder aux éléments de la liste d'identificateurs. En effet, la liste de paramètres effectifs est une liste linéaire à deux éléments. L'écriture de $ AF (1) à droite de : = demande l'évaluation de cet élément. $ PE a donc pour valeur la liste d'identificateurs à initialiser.

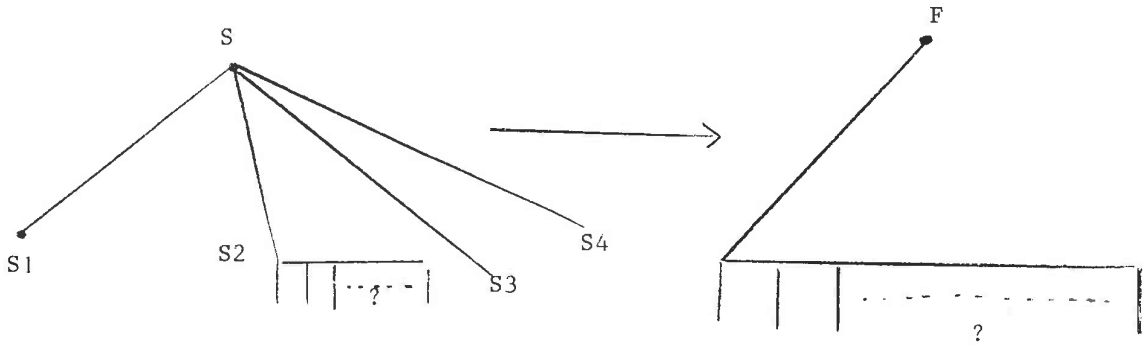
$ PE est une métaliste de travail propre à $ init : elle est donc déclarée locale, comme la métavariable $ I.

La séquence
```
$ liste : = A, B, C, D, E ;
$ init (liste, 5) ;
```
conduit donc à la génération des instructions :

```
A = 5 ; B = 5 ; C = 5 ; D = 5 ; E = 5 ;
```

9.543 Un métamodule de transfert.

Il s'agit du transfert d'une structure comportant une file de taille variable vers une file réceptive de taille variable également.



On suppose que les éléments S1, S3, sont des éléments de type simple. S4 est quelconque.

```
$ mod  $ transfert ($ S, $ F) ;
    $ local G ;
    F (1) = $ nom feuil ($ S, 1) ;
    $ G = $ nom feuil ($ S, 2) ;
    F (2, .) = $ G ;
    F (2 + taille actuelle ($ G)) = $ nom feuil ($ S, 3) ;
    F (2 + taille actuelle ($ G), .) = $ nom feuil ($ S, 4) ;
$ fin mod ;
F ( ) entier ; $ struct (A entier, G ( ) entier, H entier,
L ( ) entier) ;

    $ transfert (S, F) ;
conduit au texte :
    F (1) = A ;
    F (2,.) = G ;
    F (2 + taille actuelle (G)) = H ;
    F (3 + taille actuelle (G), .) = L ;
```

9.544 Transfert par nom.

Transfert des feuilles d'une structure A vers celles d'une structure B qui ont même nom et dont les prédécesseurs ont même nom (sauf la racine elle-même).

Ce métamodule est l'équivalent du Move Corresponding de Cobol ou Move by Name de PL1.

```
$ mod  $ move ($ A, $ B)
```

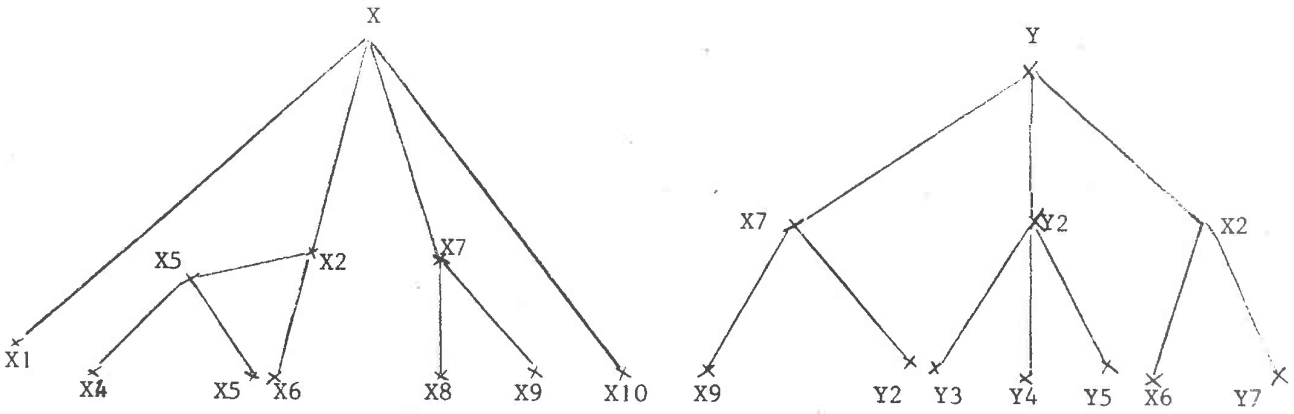
$ Local $ X, $ Y, $ X 1 ; $ co. Dans l'instruction qui suit, $ A est le paramètre de $ mot feuille et non l'identificateur du rang de l'élément courant $ oc

```
$ pour chaque $ E $ de $ mot feuille ($ A) $ faire
$ pour chaque $ F $ de $ mot feuille ($ B) $ faire
  $ X : = $ E      ;     $ X 1 : = $ X ;
  $ Y : = $ F ;
```

```
$ si  $X == $Y $et $ nbrepred ($A, $X) == $ nbrepred ($B, $Y)
      $alors $pour $K := 1, $ nbrepred ($A, $X) -1 $faire
                  $X := $ nompred ($A, $X) ; $X1 := $X1 . 'de' . $X1 ;
                  $Y := $ nompred ($B, $Y) ;
                  $ si $X ≠ $Y $alors  $allera . $fini  $fsi
            $ finpour ;
```

```
$ fsi ;
$ fini : $ fpc $ fpc $ fin mod
```

soient X et Y deux structures représentées par



On examine successivement les feuilles de X. Pour chacune d'elles, on cherche si elle a son homonyme dans les feuilles de Y. La première à réaliser cette condition est X6. $ X 1 prend la valeur 'X6'. On examine son prédecesseur dans chacune des deux structures : c'est X2. $ X 1 prend la valeur 'X6 de X2'. X6 n'avait que deux prédécesseurs : $ nbrepred ($ B, $ X) - 1 = 1. L'instruction "$ pour" est terminée. Le texte généré est X6 de X2 de X = X6 de X2 de Y ; les 'de' successifs peuvent être souvent superflus (si le même nom de champ ne figure pas ailleurs dans la structure) mais il serait long et inutile de chercher à les supprimer.

Notons que cette rédaction de $ move, entraîne l'évaluation répétée des mêmes fonctions. Ceci peut être évité en n'utilisant pas ces fonctions et en suivant l'exploration des structures paramètres grâce aux fonctions lv (lien vertical) et lh (lien horizontal).

9.545 Itération logique de 7.74.

```
$ déf MOD  $ Itération logique ($ N, $ lis, $ INIT, $ FIN, $ trait) ;
```

CO $ N indique le nombre d'indicatifs utilisés, $ LIS est une métaliste dont les éléments sont les noms des indicatifs, $ INIT est une liste dont chaque élément est une chaîne de caractères représentant une suite d'instruction : la suite "Init Ij", $ FIN est la liste des "fin Ij" et $ trait représente le traitement de la boucle la plus interne. oc ;

```
FF In entier ;
$ pour $ I = 1, $ N $ faire
        ZR . $ LIS ($ I) comme  $ LIS ($ I) ; co déclaration de
        ZRIj oc ;
        $ f pour ;

procédure proc 0 ; FFIn = 0 ;
        lire file support ; si  $ de file alors FFIn = 1 ;
```

décision	$FFI_n \neq 0$	F	V

```
$ def mode  $ table  ($ J) ; co  construction de la fin de la table
oc ;
```

action			
	$ init ($ J)	1	
	proc. $ LIS ($J)	2	
	retable	3	
	$ fin ($ J)		1

ft

```
$ fin mod ;
```

$ si $N 1 $ alors $ table (1) $ sinon $ allera $ Cas 1 $ fsi ;
co cet appel permet de construire exactement la procédure proc 0 de 7.74, lorsque $N 1 ; sinon elle sera complétée par la définition de la procédure d'ordre n -1 oc ;

$ pour $ I = 1, $N -1 $ faire

procédure proc. $ LIS ($I) ;

Décision	$ LIS ($I) ≠ ZR. $ LIS ($I)	F	-
	F F I n ≠ 0	F	V

$ table ($I) ; fin proc ;

$ fin pour ; co ce $ pour génére les déclarations des N - 2 procédures oc ; $ Allera $ ETI ;

$ cas 1 :

Action	proc. $ LIS ($ N)	1	
	retable	2	

ft ;

$ ETI : procédure proc. $ LIS ($N) ; $ trait ;
lire file support ;
si $ def file alors FI_n$ = 1 fin proc ;
proc o ;

$ fin mod ;

Un appel de métamodule tel que :

$ trait = 'traitement de In' ;
$ LIS = 'I1', 'I2', 'I3',.....'In' ;
$ Init = 'init I1,....., 'init In - 1' ;

$ fin = 'fin I1',....., 'fin In -1' ;

produit un texte identique à celui de 7.74.

On remarquera en particulier l'utilisation des méta-expressions de chaîne pour construire des identificateurs comme proc. $ LIS ($ I).

9.646 Exemple de métafonction

La métafonction $ mini calcule le minimum des éléments d'une liste.

$ fonc $ MINI ; $co cette fonction a 1 paramètre effectif : le nom de la liste $oc ;

$ local $ I, $ PAR ;

$ PAR : = $AF ; $co la liste est égale à la liste nommée dans AF $oc' ;
$ MINI = 0 ;
$ Pour chaque $ X $ de $ PAR $ faire

$ si $ X > $ MINI $ alors $ Alors $ $ MINI : = $ X
$fsi $fpc $ fin fonc ;

- Métafonction $ Max ($ A, $ B).

$ fonc $ MAX ($ A, $ B) ;
$ si $ A > $ B $ alors $ MAY : = $ A $ sinon $ MAX : =
$ B $ fsi $ fin fonc

REFERENCES AU CHAPITRE 9

1. BENAMGHAR L. — Instruction d'affectation et définition d'un métalangage dans le Projet Civa. Thèse. Université Nancy I (Juin 73).

2. STRATCHEY C. — A general purpose macro generator Computer Journal (Oct. 1965).

3. LEAVENWORK B. — Syntax macros and extended translation Comm. ACM 9.11 (Nov. 66). 790.

4. LEROY H. A. — A proposal for macro facilities in Algol. Algol Bulletin n° 22.

5. WAITE W. — A language independant macro processor. Comm. ACM 10.7. (July 67). 433 - 440

6. C. I. I. — Metasymbol sous SIRIS 7/SIRIS 8 - CII 10070 Réf. 3753 E/Fr

7. Mc [illegible] M. D. — Macro instruction extensions of compiler language Comm. ACM 3 - 4 (April 60). 214.

CHAPITRE 10

LES MODULES DE COMMANDE

10.1 COMMANDES DE CALCUL.

10.2 MODULE DE COMMANDE ET CLASSE D'APPLICATION.

10.21 Adjonction de déclaration dans une classe d'application.

1 Adjonction de déclarations d'objets élémentaires dans la classe d'application.

2 Adjonction de déclarations de modules et de classes.

10.22 Suppression de déclarations d'une classe d'application.

10.23 Modification des textes de modules et de classes d'une application.

1 Indication de l'unité à modifier.

2 Localisation de la modification dans l'unité.

3 Procédures de modification de texte.

10.24 Autres procédures de la classe de modification.

10.25 Module d'initialisation d'une classe d'application.

10.3 MODULE DE COMMANDE ET UNITES DE TRAITEMENT.

10.4 MODULE DE COMMANDE ET EXPLOITATION DU SYSTEME.

10.41 Déclarations de la classe de modification.

10.42 Variables contrôlables déclarées dans la classe système.

1 Anomalies.

2 Erreurs.

3 Autres évènements décrits dans la classe système.

4 Localisation d'un évènement.

CHAPITRE 10

LES MODULES DE COMMANDE

Le travail à exécuter est soumis au système (au module M_0) par l'intermédiaire d'un module de commande (cf. 1.8). Celui-ci a essentiellement pour rôle de décrire l'enchaînement de l'exécution des unités de traitement déjà décrites dans l'application et de permettre toutes les modifications souhaitables de la classe d'application. Alors qu'une unité de traitement décrit des travaux essentiellement répétitifs, un module de commande, au contraire, permet de tenir compte des conditions particulières à chaque exécution. Il constitue l'interface entre un utilisateur et la classe de son application. Il est interprété, puisqu'il n'est pas répétitif, par le module M_0, ou interprète des modules de commande, qui constitue l'exécutif du système. Toute action à réaliser par l'exécuteur est demandée par l'intermédiaire de commandes ou d'appels de modules.

10.1 COMMANDES DE CALCUL.

Un module de commande "utilise" la classe système C_s.

La première ligne d'un module de commande indique le nom de son application : le module de commande "utilise" la classe de cette application. Cette ligne s'écrit :

Application < nom de l'application > ;

Elle a pour effet de donner une valeur à la variable (file de caractères) de la classe "système" "nom d'application en cours".

Le module de commande utilise également la classe de modification. Celle-ci contient des procédures permettant de modifier les déclarations de l'application en cours, et uniquement celles-ci. Il peut également utiliser une autre classe, dite classe de commande (cf. 1.8). Celle-ci peut-être décrite modulairement et utiliser d'autres classes, elles sont assimilées à la classe de commande :

- il existe au plus une classe C, différente de la classe d'application, telle que $M_c \cup C$;
- toute classe D telle que $M_c \hat{\cup} D$ est une classe de commande.

Une directive "utilise" doit être écrite explicitement pour désigner la classe de commande utilisée.

Par l'intermédiaire de la classe d'application et des classes de commande, un module de commande a donc accès à des informations Civa. Elles peuvent être utilisées dans des calculs, par exemple pour déterminer s'il convient ou non d'appeler un module. Ces calculs sont décrits par des commandes de calcul. Les commandes de calcul sont les instructions du langage écrites dans le module de commande.

On peut considérer qu'à la rencontre d'une commande de calcul, le module M_0 crée un module dans l'application en cours, contenant uniquement cette instruction. Ce module utilise donc la classe d'application et, éventuellement les classes de commandes. M_0 le rend "intérieur au système" (cf. 1.3), l'exécute (appel de ce module), et le détruit.

Les commandes ainsi définies correspondent à l'instruction d'affectation, les tables de décision et de sélection, ainsi que leur forme simplifiée en instruction conditionnelle et l'instruction d'itération simple. Les instructions de traitement global des files ne sont pas admises à ce niveau, car les calculs décrits dans un module de commande portent le plus souvent sur des objets de type simple et rarement sur des files.

La définition de ces commandes est identique à celle donnée dans les chapitres précédents pour les instructions correspondantes.

10.2 MODULE DE COMMANDE ET CLASSE D'APPLICATION.

Un module de commande d'une application utilise la classe de l'application et c'est le seul type de module à pouvoir utiliser la classe de modification. Un module de commande peut donc gérer la classe de l'application par l'intermédiaire de cette classe de modification. Rappelons qu'il ne peut modifier que la classe de l'application en cours.

La classe de modification contient des procédures permettant d'ajouter, de retirer ou de changer des déclarations de la classe d'application. Pour des raisons de commodité, l'appel de ces procédures obéira parfois à des règles particulières d'écriture.

Dans tout ce chapitre, nous appellerons <u>déclaration d'objets élémentaires</u> toute déclaration d'objets du langage autres que les modules et les classes.

10.2 Adjonction de déclaration dans une classe d'application.

L'adjonction de déclarations dans une classe d'application, donc en particulier dans la classe système lorsque l'application "système" est l'application en cours, peut être demandée par l'intermédiaire de la procédure de modification. CREER.

10.2.1 Adjonction de déclarations d'objets élémentaires dans la classe d'application.

L'appel de la procédure CREER s'écrit alors :

CREER (< suite de déclarations >)

dans lequel < suite de déclarations > désigne une suite de déclarations du langage séparées par un point-virgule. Si les déclarations sont syntaxiquement correctes, elles sont alors ajoutées à la classe d'application. Si l'une des déclarations est syntaxiquement incorrecte, aucune adjonction n'est effectuée, il y a édition d'un message d'erreur et fin de la procédure CREER. De même, aucune adjonction n'est effectuée si l'un des identificateu déclarés dans la suite de déclarations est déjà déclaré dans la classe d'application . Il y a également édition d'un message d'erreur dans ce cas (double définition). En effet, l'adjonction d'une déclaration dans une classe d'application ne peut pas toucher les unités de traitement déjà créees dans cette application et déjà compilées ; si l'on acceptait, les doubles définitions, la nouvelle remplaçant l'ancienne, il faudrait dans ce cas recompiler toutes les unités utilisant cette classe. Nous n'acceptons donc le changement dans les déclarations de la classe d'application en cours, que lorsqu'ils sont demandés explicitement par une procédure CHANGER.

Avec les règles données ci-dessus, si la suite de déclarations est erronée, un nouvel appel de CREER, avec une suite corrigée de déclarations, ne constitue pas un changement de la classe d'application, mais bien une adjonction de déclarations. Il ne nécessite pas de recompilations des modules et des classes de l'application (cf. 10.3).

10.212 Adjonction de déclarations de modules et de classes

L'appel de la procédure CREER s'écrit alors

CREER ou CREER (COMMUNS)

ou CREER (liste de noms d'applications).

Cet appel permet de demander l'adjonction du texte de classes ou de modules à la classe de l'application en cours, et, s'il s'agit de modules, de les rendre intérieurs au système.

Si une unité de même nom existe déjà dans l'application, l'adjonction n'est pas réalisée et un message d'erreur est édité : pour remplacer un texte il faut le faire explicitement en supprimant le texte existant et en créant le nouveau, ou en utilisant la procédure "changer".

Si CREER n'est pas suivi de paramètre, tous les modules et toutes les classes dont le texte suit CREER sont ajoutés à la classe de l'application et ils ne pourront pas être employés par des modules ou classes d'une autre application (par "appelle" ou "utilise").

La suite d'unités à créer s'arrête à la première ligne du module de commande n'appartenant pas à une déclaration de classe ou de module : commande ou appel de module après fin classe ou fin module.

Si par contre, CREER est suivi de (COMMUNS), ces unités pourront être employées par tout module ou toute classe du système, et si CREER est suivi d'une liste de noms d'applications, seuls les modules et classes de ces applications pourront employer les unités créées (par l'intermédiaire d'une directive "utilise" ou "appelle" suivie d'un nom qualifié ou par l'intermédiaire de la procédure "ajouter").

La procédure de modification AJOUTER, permet d'adjoindre aux déclarations de la classe d'application celle d'une classe ou d'un module d'une autre application.

Un appel de cette procédure peut alors s'écrire :

AJOUTER (<nom d'unité de nom d'application>) SEUL.

Si l'unité correspondante, v, classe ou module, est efffectivement déclarée dans l'application désignée et si son emploi par l'application en cours a été autorisé, une copie en est tirée et introduite dans la classe de l'application en cours.

En particulier, l'unité v pourra être utilisée par l'intermédiaire de l'identificateur v et non plus par v de « nom d'application » . Comme pour la procédure "créer", l'adjonction ne sera réalisée que s'il n'existe pas déjà d'unité de nom v dans l'application en cours.

L'unité v "utilise" ou "appelle" certainement d'autres unités de l'application de v et une adjonction de v nécessite, en général, l'adjonction d'autres unités de l'application. L'appel de AJOUTER suivi du mot "ENVIRONNEMENT" permet de demander l'adjonction de l'unité v et des unités de $\hat{A}$ (v) et de $\hat{U}$ (v), à condition qu'il n'existe pas déjà d'unité de même nom dans l'application courante. S'il existe n unités ω_i de même nom que des unités existant dans l'application courante, seule seront ajoutées à celles-ci les unités de

$$\left(\bigcup_{i=1}^{n} \hat{A}(\omega_i) \cup \hat{U}(\omega_i) \right) \cup \hat{A}(v) \cup \hat{U}(v)$$

la liste des unités n'ayant pas pu être ajoutées est alors éditée. Elles pourront être ajoutées ultérieurement, une par une, en changeant leur nom.

Pour ajouter une unité v alors qu'il figure déjà une unité de même identificateur v dans l'application en cours, deux solutions sont possibles :

- supprimer l'unité figurant déjà dans l'application en cours et ajouter la nouvelle en conservant le même identificateur.
- ajouter le texte de la nouvelle unité en changeant d'identificateur, sans toucher à l'ancienne unité.

L'appel suivant de la procédure AJOUTER permet de demander l'adjonction dans l'application en cours de l'unité v de B, qui sera désignée par p dans celle-ci :

AJOUTER (v de b appelé p) { SEUL / ENVIRONNEMENT }

Dans ce cas, l'appel de AJOUTER peut être également suivi de SEUL ou de ENVIRONNEMENT. Cependant, nous n'avons pas défini de procédure d'adjonction permetttant d'ajouter une unité et son environnement en

modifiant les identificateurs des unités de cet environnement : en effet, supposons que cet environnement contienne une unité ω que l'on veut appeler x dans l'application en cours, car celle-ci contient déjà une unité ω, il ne suffit pas de changer le nom de l'unité, il faudrait également remplacer, dans les unités introduites, tous les appels de ω par des appels de x.

La procédure "ajouter" a donc un effet différent de la directive "appelle" et de la directive "utilise" appliquées à une unité d'une autre application, qui ne réalisent qu'une adjonction "locale". Ces directives ne peuvent s'appliquer qu'à des unités qui sont déjà compilées dans leur application. Tout leur environnement est ainsi accessible.

10.22 Suppression de déclarations d'une classe d'application.

La procédure de la classe de modification "supprimer" permet de demander la suppression de déclarations de la classe de l'application en cours et de cette classe uniquement.
L'appel de procédure s'écrit :

SUPPRIMER (<liste d'identificateurs>);

La liste d'identificateurs, séparés par un virgule, indique quelles sont les déclarations à supprimer. Ces identificateurs désignent des objets quelconques ; objets élémentaires, ou modules, ou classes. Pour les objets élémentaires, un appel de cette procédure n'entraîne pas de déplacement des autres objets de la classe d'application et les classes et modules de l'application ne doivent donc pas être recompilés (cf. 10.3)
Pour les identificateurs de module et de classe, seule l'unité correspondante v est supprimée (en particulier aucune unité de $\hat{U}^{-1}$ (v) ou de $\hat{A}^{-1}$ (v) n'est supprimée).
Cependant, la procédure "supprimer" peut également être appelée sous la forme

SUPPRIMER (<liste d'identificateurs>) référence,

ou encore, chaque identificateur de la liste peut être suivi du symbole référence. Pour chaque unité dont l'identificateur est suivi de référence, ou pour toutes les unités si référence suit la liste d'identificateurs,

la procédure "supprimer" édite la liste des identificateurs des unités appartenant à $\vec{U}^{-1}(v)$, à $\hat{A}^{-1}(v)$, $\hat{U}(v)$ et $\hat{A}(v)$.

10.23 Modification des textes de modules et de classes d'une application.

10.231 Indication de l'unité à modifier.

Nous avons vu jusqu'à présent que des unités peuvent être ajoutées dans une application en donnant explicitement leur texte ou en indiquant dans quelle autre application les prendre. On peut également, et c'est fréquemment le cas au cours de la mise au point d'une unité de traitement, vouloir construire une nouvelle unité à partir du texte d'une unité déjà intérieure au système. Des procédures de modification sont prévues à cet effet. Ces procédures de modification de texte (Payafar M., 71) permettent de construire une nouvelle unité à partir d'un texte existant en conservant le même identificateur. Elles portent toutes sur l'unité désignée par la valeur d'une file de caractères de la classe de modification : "unité en cours". La procédure CORRIGER a pour but de donner une valeur à cet élément. Elle s'écrit :

CORRIGER (< identificateur >).

"unité en cours" prend alors pour valeur cet identificateur. Les procédures de modification qui suivront porteront donc sur cette unité.

10.232 Localisation de la modification dans l'unité.

Pour indiquer en quel point du texte l'appliquer, l'appel d'une procédure de modification peut comporter trois types de localisation :

- un rang, ou position absolue,
- une position relative,
- un texte.

Une position absolue peut être repérée par une notation d'entier (calculable à l'évaluation du module de commande), dont la valeur est le rang depuis le début du texte source, de la ligne repérée.

Une position relative s'écrit $\smile * \smile$ ou $\leftrightarrow * + \alpha \leftrightarrow$ où α est une notation d'entier. $*$ repère la ligne qui suit la dernière modification. Après l'appel CORRIGER, $*$ repère la première ligne du texte source.

Un texte permet de repérer une position dans le texte source : l'endroit du texte source où est écrit le même texte (les blancs ne sont pas significatifs).

10.233 Procédures de modification de texte.

Elles sont au nombre de trois : "insérer", "oter", "modifier".

INSERER (<localisation>, n) ;

permet d'insérer les n lignes qui suivent cet appel de procédure, dans le texte de "l'unité en cours". L'insertion est faite après la ligne repérée par la localisation.

OTER (<localisation>, n) ;

permet de demander la suppression de n lignes du texte source, à partir de la ligne repérée par la localisation (y compris cette ligne).

MODIFIER (<localisation>, n) ;

permet de demander le remplacement de n lignes à partir de celle repérée par la localisation (comprise) par les n lignes de texte qui suivent l'appel de cette procédure.

On pourra remarquer que ces procédures permettent la mise à jour de textes quelconques.

10.24 Autres procédures de la classe de modification.

CHANGER (<identificateur>) ;

permet de demander le remplacement de l'unité désignée par l'identificateur, par le texte qui suit cet appel de procédure.

REMPLACER (<identificateur> par <identificateur>) ;

permet de remplacer le nom d'une unité : après l'appel, le second identificateur désigne l'unité qui était désignée par le premier avant l'appel. Celui-ci ne désigne plus rien.

LISTER (<liste d'identificateur>)

permet de demander une écriture du texte source des unités désignées par la liste d'identificateurs.

10.25 Module d'initialisation d'une classe d'application.

Lorsqu'un module de commande lui est soumis, et lorsqu'il a rencontré la directive "application" précisant la valeur "d'application

courante" (cf. 10.1), le module M_0 appelle implicitement un module d'initialisation de la classe d'application. Ce module est décrit par l'utilisateur. Il n'y a qu'un seul module d'initialisation par application. Ce module est désigné par INITIALISATION, dans toutes les applications. Il a pour rôle de fixer des valeurs, propres à l'application, aux paramètres d'exploitation et de définir des contrôles pour les variables contrôlables de la classe système.
"Initialisation" peut en fait être considéré comme un métamodule ayant une définition différente dans chaque application. Lorsque le module de commande l'appelle, le texte de "Initialisation" est "inclus" dans le module de commande.

10.3 MODULE DE COMMANDE ET UNITES DE TRAITEMENT.

Un module de commande peut également contenir des appels de modules de l'application courante. Un appel de module constitue une demande d'exécution de ce module. Nous avons vu en 1.9 , que les modules d'une application ne sont pas interprétés mais traduits avant de pouvoir être exécutés. Un module appelé dans un module de commande est qualifié de "module directeur" (cf. 1.8). L'ensemble des modules de $\hat{A}$ (m) des classes de $\hat{U}$(m), où m est un module directeur constitue une unité de traitement. Le module M_0 entretient pour chaque application un fichier des textes compilés des modules et des classes de l'application et un fichier des textes édités des unités de traitement.

Pour pouvoir exécuter un module directeur, l'unité de traitement correspondante doit être "éditée" - c'est-à-dire se présenter sous forme d'un module de chargement, produit par l'éditeur de liens. Pour pouvoir éditer cette unité de traitement, on doit disposer des traductions de toutes les unités qu'il contient.

Le module M_0 gère l'ensemble des unités d'une application et assure automatiquement tous les changements d'état nécessaires, comme nous le préciserons au chapitre 21.

Il peut cependant être souhaitable, pour en permettre l'appel par une autre application par exemple, de demander explicitement la compilation ou l'édition de modules.

L'appel de procédure :

COMPILER (<iter >, <mode>, <liste d'identificateur >)

permet de demander la compilation des unités désignées par la liste d'identificateurs, <mode> indique en quel mode faire cette compilation : exploitation ou mise au point.

<iter> indique s'il faut compiler chaque unité de la liste seule (seul) ou s'il faut compiler également son environnement (environnement).

RELIER (<identificateur>) ;

permet de demander la "reliure" , c'est-à-dire la pré-édition de liens puis édition de liens (cf. chap. 19), du module désigné.

10.4 MODULE DE COMMANDE ET EXPLOITATION DU SYSTEME.

10.41 Déclarations de la classe de modification.

La classe système contient des déclarations d'objets qui sont donc utilisables par toutes les classes d'application et qui sont donc communs à toutes les unités de traitement. Certains objets, tels que les paramètres d'exploitation ou les éléments de traducteur ne doivent pas être accessibles à ces unités de traitement. Ces objets doivent donc être déclarés dans une application particulière, dont la classe soit utilisée par tout module de commande, par le module initial et par eux exclusivement : c'est donc la "classe de modification" déjà rencontrée en 10.2.

Outre les procédures de modification, la classe de modification contient les déclarations :

- des modules du système de compilation : codifieur, générateur, pré-éditeur, éditeur et les classes décrivant leurs zones de communication
- des paramètres d'exploitation
- des fichiers descriptifs des classes et des modules
- un module d'initialisation du système : INIT.

La compilation pouvant être réalisée en deux modes : "exploitation" et " mise au point", les modules du système de compilation peuvent figurer dans la classe de modification en deux versions différentes.

Les paramètres d'exploitation sont déclarés dans la classe de modification et les valeurs "par défaut" de ces paramètres leur sont attribuées par le module M_0, qui, lorqu'il reçoit un module de commande commence par appeler un module d'initialisation du "système" : le module INIT de la classe de modification.

La classe de modification étant "utilisée" par un module de commande, lorsque celui-ci "appelle" le module d'initialisation de l'application, c'est-à-dire, lorsque le texte de ce module d'initialisation est inséré au module de commande, les paramètres d'exploitation peuvent à nouveau être modifiés pour recevoir les valeurs implicites spécifiques à l'application en cours. Enfin, ces paramètres pourront encore être modifiés par des instructions écrites explicitement dans le module de commande.

La liste exhaustive et le rôle de chacun des paramètres d'exploitation sera décrite dans (Viault D., 74).

Le module d'initialisation de la classe de modification, INIT, contient également la déclaration d'un certain nombre de contrôles comme nous le verrons au paragraphe suivant.

10.42 Variables contrôlables déclarées dans la classe système.

10.421 Anomalies.

Un certain nombre de conditions pouvant être considérées comme anormales sont détectées par le matériel. Le plus souvent, ces évènements ne constituent pas un danger pour l'exécution d'une unité de traitement, qui peut en général être poursuivie. Il suffit alors d'éditer un message d'édition.

Il convient donc de prévoir, au niveau de la classe système, une action par défaut à entreprendre systématiquement dès qu'un tel évènement se réalise tout en laissant la possibilité de décrire d'autres séquences de récupération si on le désire.

Pour l'ensemble de ces conditions (anomalies), la classe système contient la déclaration d'une variable contrôlable "anomalie", qui prend la valeur "vrai" lorsqu'une anomalie est détectée (cf. 6.6). Pour chaque anomalie, il est également déclaré une variable contrôlable appelée "anom i" :

anomalie, anom 1, anom 2,...., anom n booléen contrôlable ;

La détection de l'anomalie "anom i" se traduit par l'affectation à

"anom i" de la valeur "vrai". Ces variables contrôlables peuvent être suspendues ou libérées.

Le module INIT d'initialisation de la classe de modification contient la déclaration des contrôles qui sont définis de façon standard pour chacune de ces variables :

```
quand anom 1 faire impressions standard ("anomalie anom i") ;
                   anomalie = vrai fqd ;
quand anom 2.....
```

"impression standard" est un module de la classe système permettant l'impression du message donné en paramètre et des renseignements permettant de localiser l'instruction ayant causé l'évènement. Il est décrit dans (Ducloy J. 73) en 6.41.

Si dans une application, on désire définir des contrôles différents, il suffira de les décrire dans le module "initialisation" de la classe de cette application.

Un métamodule déclaré dans la classe système, permet de les écrire facilement. Il peut être appelé par

$ $ init anomalies (< liste des anomalies à contrôler de façon standard >)

Sont considérées comme des anomalies le débordement entier dbent, le débordement flottant dbflt, l'erreur arithmétique décimale erad, etc... et le débordement de file pour une file en position d'émetteur ihllect (indice hors limite en "lecture"), lorsqu'on est en mode "mise au point".

10.422 Erreurs

Lorsqu'un évènement anormal doit entraîner l'arrêt de l'unité de traitement, on parle d'erreur et non plus d'anomalie. Ces évènements sont toujours détectés par le matériel ou le système : il leur correspond des variables contrôlables qui ne peuvent pas être vraiment suspendues ou libérées : une libération ou une suspension fait reprendre le contrôle standard. Comme pour les anomalies, une variable booléenne contrôlable est forcé à "vrai" dès que l'une des erreur est arrivée. La séquence de récupération d'un contrôle associé à cette variable ne peut contenir d'appel de module ou de procédure. Hormis deux restrictions, elles sont décrites comme les anomalies. Sont considérés comme des erreurs de ce type les débordements de file en position d'émetteur lorsqu'on est en mode "mise au point".

Cependant certaines erreurs ne permettent pas de reprendre le contrôle du calculateur : erreurs dites "catastrophiques". Une seule variable contrôlable est prévue pour elles : erreur. La séquence de récupération ne peut contenir que des impressions. Sont de ce type la rencontre d'une erreur simple dans la séquence de récupération d'une erreur simple ou le dépassement du temps de calcul accordé à une unité de traitement.

10.423 Autres évènements décrits dans la classe système.

Il s'agit de l'intervention de l'opérateur itopérateur et l'interprétation liée aux compteurs de temps itcompteur qui sont des évènements reconnus par le matériel et d'évènements dont l'arrivée peuvent être prévus par le compilateur.

Pour ces trois éventualités, il est déclaré trois booléens dans la classe système, qui peuvent être déclarés contrôlables ultérieurement dans une application. Cette disposition est prise parce que le contrôle de ces évènements entraîne un ralentissement de l'exécution et il n'est pas généré lorsque ces variables ne sont pas déclarées contrôlables.

Trois autres objets sont déclarés dans la classe système pour avoir accès au temps d'exécution :

- la variable réelle contrôlable temps restant : elle contient le nombre de secondes restant disponibles pour la chaîne de traitement.
- une procédure initcompteur (t) où t est une expression entière ; elle initialise le compteur de temps à la valeur : t secondes.
- une variable réelle non contrôlable temps compteur : elle contient le nombre de secondes restantes avant le passage à zéro du compteur de temps.

La variable booléenne itcompteur prend la valeur vrai lorsque le compteur de temps, préalablement initalisé par "initcompteur" passe à la valeur zéro.

Pour les évènements dont l'arrivée peut être prévue par le compilateur, il s'agit de :

entrée module
entrée procédure
sortie procédure
sortie module
fin instruction ,

qui sont cinq booléens déclarés dans la classe système mais ne peuvent être contrôlés qu'en mode "mise au point". Les quatre premiers y sont déclarés contrôlables - fin instruction doit être déclaré contrôlable par l'utilisateur. Ils sont prévus pour permettre l'écriture en Civa de service de trace, tout comme les dispositifs de localisation d'un évènement.

10.424 Localisation d'un évènement .

La classe d'application système contient la déclaration d'un entier adrevt utilisé par le module M_0 pour y ranger, lorsque survient un évènement, l'adresse de l'instruction ayant provoqué cet évènement. Cet entier n'est pas contrôlable.

Y sont déclarés également :

- nom module file (max = 12) car ;
- nom procédure file (max = 12) car ;
- numéro entier.

Pendant l'exécution d'une unité de traitement compilée en mode "mise au point", et uniquement dans ce mode, le module M_0 range dans "nom module" le nom du module en cours, dans "nom procédure" le nom de la procédure en cours : il est vide (blanc) si l'on est pas dans un appel de procédure, dans "numéro" le numéro de la ligne contenant l'instruction qui a provoqué l'évènement.

Ces objets peuvent être employés par l'utilisateur, par exemple pour prévoir, dans une séquence de récupération, l'édition de la localisation de l'instruction ayant provoqué l'évènement.

Ces informations pourront être utilisées dans des modules d'aide à la mise au point de programmes, comme par exemple le module "impression standard" suivant :

```
module impression standard (message) ;
        message file car ;
        si mise au point alors
        $ imprimer ( '****', message, 'dans le module', nom module,
                     'dans la procédure', nom procédure, 'à la ligne',
                     numéro)
        sinon $imprimer ('****', message, 'à l'adresse', adrevt)
        fsi fin mod.
```

REFERENCES DU CHAPITRE 10

DUCLOY J., 73 — Compilation dans le projet Civa
Thèse Docteur-Ingénieur Nancy Mars 73

PAYAFAR M., 71 — Modification des textes dans le projet Civa
Thèse d'Université Nancy 71.

VIAULT D., 74 — Définition et interprétation des modules de commande dans le projet Civa. Thèse de 3ème Cycle
A paraître.

CHAPITRE II

ACQUISITION DES DONNEES ET EDITION DES RESULTATS

Si on observe la suite des opérations que subissent les données d'un traitement entre le moment de leur création et leur traitement proprement dit on peut constater qu'elle commence par deux étapes très importantes, mais sur lesquelles nous ne nous étendrons pas ici, l'élaboration des données et leur saisie. La saisie aboutit à la présentation des informations sur un support physique dont la nature est plus propre à la saisie qu'aux traitements : rubans, cartes et quelquefois bandes magnétiques. La présentation elle-même des informations sur leur support est déterminée par la commodité de la saisie et d'une éventuelle lecture sur le support (cartes), en particulier, si même il y a un codage des informations à leur saisie il est faible et en tous cas peu souhaitable, car c'est une source importante d'erreur. Enfin, les informations ainsi saisies sont présentées consécutivement sur le support.

L'étape suivante est en général un changement de support (cartes à bande par exemple) dont le but est de ranger les informations à traiter sur un support plus commode pour les traitements et éventuellement l'archivage. Ce changement de support est, en général, une simple recopie. Si, toutefois, il est nécessaire de modifier la disposition relative des différents éléments des données, il faut alors procéder à une première étape de traitement qui effectue ces transformations (changements de structures dus à une organisation des données propre à la saisie et différente de celle des traitements).

Ce changement de support est suivi d'une étape de contrôle de la validite des données. Elle est plus ou moins importante selon la présence de contrôles au cours de l'élaboration et de la saisie, puis au cours du traitement proprement dit et selon l'influence plus ou moins grande de la présence d'erreurs dans les données entrées. On est alors amené à écrire pour chaque application des programmes importants effectuant ces contrôles, en utilisant la plupart du temps le même langage que pour les traitements. Cette étape est

donc conçue en général comme un traitement. Elle ne modifie pas la présentation des données, mais élimine simplement des données erronées. Elle doit donc être suivie en général d'une étape de mise à jour de l'ensemble des données contrôlées et prêtes à être traitées.

Le traitement peut alors commencer. Remarquons cependant que même après les contrôles les informations sont toujours présentées sur les supports magnétiques selon des formats que nous qualifierons "d'externes", en ce sens que ce sont les formats imposés par la saisie : les nombres réels ou entiers sont écrits sous forme d'une suite de chiffres décimaux, les nombres décimaux sont écrits en format "étendu"... Pour pouvoir effectuer des opérations sur ces valeurs il sera donc encore nécessaire de les convertir en un format dit "interne" car c'est celui nécessité par les traitements : entiers et réels en binaire, décimaux condensés. Si on travaille en Cobol par exemple, ces opérations de conversions seront nécessaires à chaque transfert d'information depuis, ou vers, un fichier.

Les résultats des traitement sont constitués en général d'un ensemble important d'informations qui sont produites sur un support intermédiaire sous un format "externe", mais sous une présentation qui est en général impropre à leur diffusion.

La dernière étape est alors une étape d'édition qui réalise un changement de support des résultats obtenus et leur présentation sous une forme diffusable (adjonction d'informations complémentaires, cadrages, etc...) avec, éventuellement, un certain contrôle des valeurs obtenues.

Il nous semble souhaitable, à la fois pour la simplicité de conception et pour l'efficacité des traitements, de rassembler en une seule, les étapes intermédiaires entre la saisie et le traitement : c'est-à-dire le changement de support, les contrôles, les conversions. Nous appellerons <u>acquisition</u> la réunion de ces trois étapes et nous proposons de définir des outils spécifiques de l'acquisition.

En effet, plutôt que de devoir décrire de façon impérative les opérations à effectuer, il vaudrait beaucoup mieux n'avoir qu'à préciser les caractéristiques d'une opération d'acquisition. Il faut pour cela trouver des

solutions suffisamment générales pour pouvoir être appliquées dans chaque cas, les données à acquérir étant caractérisées par un certain nombre de paramètres.

Ces paramètres sont essentiellement les formats externes des données, les formats internes, l'organisation des données, les conditions qu'elles doivent vérifier.

Les formats externes d'une donnée sont simples : elle est constituée d'une suite de caractères numériques ou alphanumériques et il suffit alors de dire combien et dans le cas de caractères numériques, on peut vouloir préciser la forme des nombres décrits (emplacement de la virgule par exemple).

Les formats internes sont caractérisés par des types et en CIVA, ils ont donc déjà été décrits dans des classes (chapitre 4).

L'organisation peut être double. Une organisation spécifique des traitements, et une organisation spécifique des supports externes. En général les données à acquérir sont des groupes répétitifs et l'objet interne correspondant est une file (de taille variable) d'objets structurés. La structure d'un élément peut être décrite dans une classe : nous appellerons description interne la description de la file et de ses éléments. La description interne est un paramètre de l'acquisition, elle indique l'organisation interne des données et leur format interne. Mais nous avons vu ci-dessus que l'organisation des données après la saisie peut être différente de celle nécessaire aux traitements : par exemple l'objet interne est une file de taille bornée d'entiers, l'objet externe est une suite d'entiers terminée par un zéro, ou encore l'objet interne est une structure, l'objet externe est une suite de valeurs de champs de cette structure, tous les champs n'étant pas nécessairement représentés et les champs absents se voyant attribuer une valeur par défaut à l'acquisition. Nous voyons donc apparaître un second paramètre de l'acquisition la "description externe" des données. Elle permet d'indiquer les formats externes de chaque élément, par l'introduction de "types externes". Nous définissons alors la notion de "groupe d'entrée" comme étant un groupement de données qui sont toujours juxtaposées sur le support (c'est donc une "structure externe").

La description externe peut également indiquer l'organisation des groupes d'entrées sur le support ; elle décrit un article du fichier externe ; un fichier externe est constitué d'une suite d'articles. Description externe et description interne peuvent décrire deux organisations différentes. La correspondance entre les groupes d'entrées de la description externe et les champs de la structure interne est alors un paramètre de l'acquisition.

Enfin le dernier paramètre d'une acquisition est constitué par l'ensemble des conditions que doivent vérifier les données à acquérir. Elles permettent de décrire les contrôles à opérer. Ces contrôles sont de divers types :

- contrôles de total par lot,

  ce contrôle suppose que le support porte également le total des valeurs d'une suite de données à contrôler ; ce total est recalculé à l'acquisition et comparé à la valeur lue.

- contrôles d'assortiment,

  ce contrôle consiste à vérifier qu'un article est complet, c'est-à-dire que tous les groupes d'entrées dont la présence a été déclarée obligatoire sont effectivement représentés dans cet article, ou encore, si un préordre ou un ordre a été défini sur l'ensemble des articles, il peut consister à vérifier qu'il est bien respecté (contrôle de séquence).

- contrôle de compatibilité,

  il s'agit de vérifier que les valeurs lues pour un groupe d'entrées sont compatibles avec celles des autres groupes d'entrées du même article, ou avec celles du même groupe d'entrée dans les articles précédents.

- contrôles de validité du format.

Services d'acquisition :

Il devient alors naturel de songer à définir un générateur de programmes d'acquisition à partir de ces quatre types de paramètres : description externe, description interne, correspondance externe-interne, contrôles.

C'est la solution que nous avons adoptée. C'est d'ailleurs la solution souvent adoptée pour les problèmes d'édition (1).

En fait, nous n'avons pas écrit un générateur de programmes tout à fait général et qui devait donc pouvoir être utilisé dans tous les cas. En effet, un générateur est un software "fermé" en ce sens que l'ensemble des valeurs possibles des paramètres est figé et qu'il n'est pas possible de lui adjoindre de nouvelles possibilités ; il se doit donc d'être le plus général possible.

Nous avons préféré définir le métalangage de CIVA (chapitre 9) qui nous permet d'écrire facilement des générateurs et, éventuellement, si les générateurs proposés sont insuffisants, qui devrait permettre à un utilisateur de définir lui-même de nouveaux générateurs de programmes d'acquisition. Ce qui permet alors de générer plus facilement des textes de programmes bien adaptés à chaque cas, ce qui est particulièrement important dans la mesure où le texte généré va être exécuté successivement pour chacun des articles du support externe.

Le travail de Jean-Jacques CHABRIER (CHABRIER J.J., 73) décrit l'état actuel de nos travaux sur ces problèmes. Il décrit les différents types de contrôles que l'on peut avoir à effectuer, il précise la notion de groupe d'entrée et les problèmes de correspondance entre donnée externe et structure

(1) Il existe de nombreux générateurs de programmes d'édition, ce qui prouve que la fonction d'édition est généralement bien considérée comme une fonction bien individualisée, alors que ce n'est pas encore le cas de l'acquisition.

interne. Enfin il propose plusieurs métamodules d'acquisition.

Il fait le point à un instant donné de l'élaboration du projet et il est bien sûr perfectible. En particulier, il devrait être possible de définir un langage de description des paramètres encore plus simple d'utilisation. Il conviendrait sans doute aussi d'insister particulièrement sur les métamodules d'acquisition dans le cas simple où les structures internes reflètent exactement l'organisation des informations sur le support externe. Enfin ce travail doit être poursuivi pour mettre au point une méthode d'analyse des données conduisant facilement à la mise en évidence des paramètres de l'acquisition.

Les problèmes d'édition sont assez semblables à ceux d'acquisition et peuvent être justiciables des mêmes procédés. Tous les résultats sont produits en "format interne" et une étape d'édition fait passer d'une file d'objets en format interne à leur présentation en format externe sur un support lisible. Des métamodules d'éditions permettent de générer pour chaque file un programme spécifique d'édition. Cependant, même si l'édition comporte aussi des contrôles de validité des résultats, les problèmes de contrôle sont moins importants que pour l'acquisition. Par contre on trouve alors des problèmes importants de "cinématique du fichier à éditer" et de programmation des ruptures. Les métamodules obtenus sont donc assez différents.

Dans la version actuelle, J.J. CHABRIER a utilisé essentiellement des tables de décision pour décrire cette cinématique, mais il semblerait particulièrement intéressant d'utiliser les instructions d'itération logique définies au chapitre 8.

Les services d'édition ainsi obtenus possèdent alors cet avantage important de pouvoir être décrits dans le même langage que celui utilisé pour décrire les traitements et ils peuvent réutiliser les descriptions déjà faites (description interne).

REFERENCES DU CHAPITRE II

CHABRIER J. J., 73 Acquisition et édition des fichiers, analyse des données dans le projet CIVA.
Thèse 3ème cycle, NANCY 1, décembre 73.

CHAPITRES 12 - 13 - 14

12. ACQUISITION.

13. EDITION.

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

Ces chapitres ne sont pas présents ici ; les problèmes d'acquisition et d'édition ont été décrits dans (CHABRIER J.J., 73) et un premier ensemble de solutions y est décrit ; quant aux métamodules décrivant des opérations de service, ils ont été décrits dans (BENAMGHAR L., 73).

BENAMGHAR L. 73 Instruction d'affectation et définition d'un métalangage dans le projet CIVA. Thèse Doctorat d'Ingénieur. NANCY 1. Juin 73.

CHABRIER J.J. 73 Acquisition et édition des fichiers, analyse des données dans le projet CIVA. Thèse 3ème cycle. NANCY 1. Décembre 73.

CHAPITRE 15

EXEMPLES D'APPLICATION

15.1 CALCUL DE PRIMES D'ASSURANCE.

15.11 Principe du calcul.

15.12 Paramètres.

15.13 Calcul de la prime.

15.2 EXERCICE .

CHAPITRE 15

EXEMPLES D'APPLICATION

Le mode d'expression défini dans ce projet a déjà été employé dans diverses analyses : problèmes de gestion de l'Université de Nancy I par Madame Rolland, problèmes de gestion de l'Université de Nancy II par Monsieur Barthélémy. Il est également utilisé dans l'enseignement de l'I. U. T. d'informatique.
L'exemple de nous développerons d'abord ici est extrait d'un problème de gestion des assurés automobiles dans une compagnie. Nous ne décrivons ici qu'un seul poste de travail : le calcul de la prime d'assurances. Nous ne reprenons pas son analyse, mais simplement la description de la solution.
Nous présentons ensuite l'exemple d'un énoncé d'exercice destiné à des étudiants débutants en informatique, pour montrer essentiellement l'intérêt de la programmation modulaire même pour de petites applications, en tant qu'aide à l'analyse d'un problème. Il ne s'agit donc pas d'une application de Civa mais d'une application de la modularité.

15.1 CALCUL DE PRIMES D'ASSURANCE.

15.11 Principe du calcul.

Pour chaque assuré, en fonction de paramètres à définir, on détermine un nombre de points attribués à cet assuré.
Selon le nombre d'accidents survenus durant les deux dernières années, ce nombre de points sera majoré ou minoré.
On détermine ensuite le coefficient à appliquer à cet assuré en fonction du nombre de points, du véhicule et de l'assurance choisie.
Le montant de la prime peut alors être calculé.

15.12 Paramètres.

Le paramètre "conducteur" sera caractérisé par différentes valeurs

telles que l'âge de l'assuré, son sexe, la date d'obtention du permis....
Une table de décision permettra de décrire le calcul du nombre de points à partir de ce paramètre.
Le paramètre type d'usage caractérise les différents types d'usages pouvant être demandés par assuré. Sur un imprimé de police d'assurances, ces types sont codés. Les codes d'usage et leur signification sont les suivants :

Usages généraux

-10- tous déplacements
-11- affaires commerces
-12- promenades

Salariés

-20- promenade trajet
-21- promenade non cadre

Fonctionnaires-enseignants

-30- fonctionnaires et assimilés ecclésiastiques
-31- officiers ministériels et leurs salariés

Artisans

-40- artisans et leurs salariés

Profession de l'agriculture

-50- agriculteurs assimilés
-51- professions annexes de l'agriculture

Paramètre "véhicule"

Une compagnie d'assurance différencie les véhicules par leur puissance fiscale, leur utilisation,... ainsi nous aurons douze catégories de véhicules différentes que nous repérerons par un code "code véhicule" sous forme d'un entier allant de 01 à 12.

Où par exemple :

un véhicule de 7 CV fiscaux berline sera codé 09.

Paramètre "zone circulation".

De la même manière, une compagnie d'assurance divise la France en dix zones que nous appellerons "code circulation" et qui seront des entiers :

01, 11, 02, 21, 03, 31, 04, 41, 05, 06

où par exemple :

Le code 04 équivaut à grande ville

Le code 06 équivaut à Paris, Marseille, Lyon.

Paramètre "antécédent accident".

Ce paramètre nous indique si l'assuré a eu des accidents dans les deux dernières années.

- S'il n'y a pas eu d'accident, l'assuré profitera d'un bonus c'est-à-dire une réduction de prime.
- S'il y a eu un seul accident durant les deux dernières années, l'assuré aura une légère majoration de prime.
- S'il y a eu plus d'un accident pendant les deux dernières années il y a malus c'est-à-dire une majoration de prime suivant le nombre d'accidents survenus.

Ce paramètre sera un entier appelé "nbacc", et sera un indice d'une file "accident" qui nous indique le taux de la majoration.

Paramètre "type assurance".

Un assuré peut choisir différents types d'assurances. Nous coderons ce paramètre "type assurance" sous la forme d'un entier de la manière suivante :

1 tiers collision

2 tiers illimité

3 tous risques

Il peut également demander à être couvert pour un certain nombre de suppléments. Ces suppléments sont également codés de la manière suivante :

-01- vol

-02- incendie

-03- bris de glace

-05- passager (place assurée à 10.000 Frs)

-05- passager (place assurée à 15.000 Frs)

-06- passager (place assurée à 20.000 Frs)

-07- protection juridique

de -08- à -48- toute combinaison.

L'élement de la file, nous indiquera le coût des suppléments que l'assuré a demandé à la compagnie.

Exemple : l'assuré demande les clauses suivantes : bris de glace , vol, incendie, passager (place à 15.000 Frs), protection juridique ; le code des combinaisons de ces clauses est 25.

A chaque code est associé un coût du (des) supplément (s) correspondant (s)

Par exemple :

01 vol	40 Frs
02 incendie	50 Frs
03 bris glace	20 Frs
05 passager	80 Frs
07 protection juridique	30 Frs
' '	
' '	
' '	
25	220 Frs (somme des précédents)

Ces coûts pourront être représentés par une file des compléments (indicée par le code).

15.13 Calcul de la prime.^

```
Application assurance ;
module calcul prime ;
       utilise structure, travail ;
Pour chaque  assuré de fich assurés par  numéro police croissant,
             résultat de fichrésul faire
             initialisation ;
             calcul des points ;
             prime fpc
fin module ;
Classe structure ; utilise type ;
       fichassuré file assuré ;
```

type struct (n° police, conducteur, véhicule, code usage, code circulation, code assurance entier, nbaccidents, autres) ;

fich résul file struct (n° police, coût) ;

fin classe

module initialisation ;

utilise structure, type, procédures ;

sélection	Code circulation =	0i	11	02	21	03	31	04	41	05	06
action		pro 1	pro 11	pro 2	pro 21	pro 3	pro 31	pro 4	pro 41	pro 5	pro 6

ft ;

co en fonction de code circulation cette table aiguille le calcul du nombre de points vers des tables de sélection différentes oc ;

init 1

fin module ;

classe procédures ; utilise structure, travail ;

procédure pro 1 ;

nbp = sélection 2

Code véhicule =	1	2	3	4	5	6	7	8	9	10	11	12
10	13	14	17	20	23	25	27	29	31	33	35	38
11	10	11	14	17	20	22	24	26	28	30	32	35
12	7	8	11	14	17	19	21	23	25	27	29	32
20	2	3	6	9	12	14	16	18	20	22	24	27
21	0	1	4	7	10	12	14	16	18	20	22	25
30	7	8	11	14	17	19	21	23	25	27	29	32
31	3	4	7	10	13	15	17	19	23	25	25	28
40	0	0	0	3	6	8	10	14	16	18	21	24
50	0	0	3	6	9	11	13	17	19	21	23	25
51	0	0	0	3	6	8	10	14	16	18	21	24
Code usage =												

ft ;

sélection

action

Code véhicule==	1	2	3	4	5	6	7	8	9	10	11	12
Plafond	10	10	11	13	16	18	20	24	26	28	31	34

ft ;

fin proc ;

-------- co pour chacun des dix codes de circulation, on définit ainsi une table de sélection de dimension 2 représentant les valeurs du nombre de points dans chaque cas et une table de sélection de dimension 1 pour déterminer la valeur du plafond correspondant oc ;

```
procédure init 1 ;
      âge = diff (date, date de naissance) ; ce âge est arrondi en années ce
      permis = diff (date - date permis) fin proc ;
module calcul des points ;
      utilise structure, travail ;
```

décision / action

célibataire	V	V	F	V	F	-	F
sexe = 'm'	V	F	-	F	-	-	-
âge < 25	V	V	V	V	V	F	F
permis < 2	-	V	V	F	F	V	F
nbp = nbp + 12	1	1	1				
nbp = nbp + 6				1	1	1	
test plafond	2	2	2	2	2	2	
antécédent	3	3	3	3	3	3	1

fi ;

```
procédure test plafond ;
      si nbp > plafond alors nbp = plafond fsi ;
      fin proc ;
      fin module ;
module antécédent ;
      utilise structure, travail ; nacc entier ;
      nacc = nb accidents (1) + nb accidents (2)
```

sélection / action

nacc	= 0	= 1	≥ 2
nbp = nbp+2	1		
nbp = nbp-2		1	
recherche			1

fi

```
procédure recherche ; nbpacc entier ;
      nbpacc = accident (code circulation, code véhicule, nacc) ;
      si nbp   nbpacc alors nbp = nbpacc fsi ;
```

```
      co accident est une file de dimension 3 à éléments entiers : en
         fonction du code de circulation, du code du véhicule et du nombre
         d'accidents dans les deux années précédentes elle indique le
         nombre de points à attribuer à l'assuré ; elle est déclarée et
         initialisée dans la classe de l'application "assurance" oc
fin mod ;
module prime ; co calcul final de la prime et préparation du résultat oc
      utilise structure; nbpc entier ;
         nbpc = coefficient (code véhicule, nbp, code assurance) ;
coût de résultat = nbpc * prix + complèment (supplèment) ;
numéro de police de résultat = numéro police de assuré ;
      co coefficient est une file de dimension 3 dont les éléments sont des
      entiers qui indique dans chaque cas le coefficient à appliquer à
      l'assuré. Elle est déclarée et initialisée dans la classe d'application
      "assurance" ; de même complèment est une file de dimension 1 dont
      l'indice d'entrée sera la variable supplèment ; elle est également
      déclarée et initialisée dans la classe d'application oc ;
fin module ;
classe type ; co dans cette classe on définit les types des champs utilisés
              dans la classe structure oc ;
      type n° police déc (12) ;
      type conducteur struct (nom assuré (15) carac,
                              prénom assuré (10) carac,
                              date naissance date,
                              sexe carac,
                              profession (10) carac,
                              adresse (rue (30) carac,
                                       ville (15) carac,
                                       code post entier),
                              date permis date) ;
      type date struct (jour entier,
                        mois entier,
                        année entier),
      type véhicule struct (marque - genre véhicule (10) carac,
```

```
                immatriculation (10) carac,
                valeur état neuf déc 9 (5) V99,
                puis. fiscale entier,
                code véhicule entier) ;
    type code usage entier ; code circulation type entier ;
        type antécédent file (2) entier ;
        type autres struct (supplément entier,
                            mode paiement carac,
                            date échéance date) ;
        type coût déc 9 (4) V 99 ;
    fin classe ;

    classe travail
        âge, permis, nbp, plafond entier      fin classe ;
```

Dans cet exemple nous avons déclaré dans la classe "structure" la structure d'un assuré en indiquant uniquement la suite des identificateurs de sous-structures qui la composent. Ce sont en fait des identificateurs de type et on a vu (chap. 4) qu'ils peuvent être utilisés comme sélecteur. Il aurait été facile d'utiliser à ce niveau des références à des métamodules dont la déclaration se trouverait dans la la classe "type". Cette solution ne ferait pas déclarer de types intermédiaires.

15.2 EXERCICE.

Cet exercice s'adresse à des étudiants débutants en informatique, connaissant uniquement l'écriture d'algorithmes rudimentaires en langage d'assemblage CII 10070 (1).

Cet exemple a été choisi essentiellement parce qu'il est simple et court (il n'est pas nécessaire d'avoir de gros problèmes à résoudre

1) En fait, nous utilisons une machine plus simple, simulée sur 10070, comme il est décrit dans (Dendien M. C. 72 et 71).

pour que la modularité soit utile), et parce que les étudiants savent déjà décrire les actions élémentaires qui interviennent dans sa solution.

Enoncé :

Sur chaque carte d'un paquet, on a perforé une phrase dont on veut compter les mots. Les mots sont séparés par un blanc. La phrase se termine par un point. La dernière carte du paquet commence par une étoile.

Description du travail :

1. Init 1 : initialisations éventuelles.
2. Lect : lire une carte, si c'est la dernière s'arrêter
3. Traitement : traiter la carte lue - écrire le résultat.
4. Recommencer en 2.

Ce que l'on décrirait mieux ainsi :

Pour chaque phrase faire Traitement.

module traitement ;

1. Init 2 initialisations
2. tests
3. Ecrire le résultat

module Ecrire ;

C'est une opération qui a déjà été rencontrée, que les étudiants connaissent et qui a déjà été décrite.

module Tests ;

PHRASE (C) =	␣	.	autre
MOT = MOT+1	1	1	
Progresser	2		1
Recommencer	3		2

module progresser ;

C = C + 1 ;

si C >80 alors ERREUR fin module ;

module Init 2 ; initialiser MOT à zéro et C au premier caractère de la phrase.

On peut donc passer à la programmation, chaque opération décrite ci-dessus étant connue. C sera le registre 3, MOT le registre 2.

module Lect ; la lecture a déjà été étudiée : elle est décrite par le sous-programme Lecarte, registre de liaison 15, rangement à l'adresse CARTE. D'où le texte pour LECT :

```
LECT        BAL, 15     LECARTE
            LB,  1      carte     chargement du 1er
                                  octet de carte
            CI,  1      C '*'     comparaison à *
            BNE         $ + 2     tant qu'il n'y a
                                  pas égalité,
            ARRET                 on saute l'arrêt.
```

module progresser :

```
            AI, 3    1      C = C + 1
            CI, 3    80
            BG       ERREUR  fin de carte ?
```

module tests :

```
      TESTS LB, 4  CARTE, 3  chargement du caractère courant
            CI, 4  C '␣'     est-ce un blanc ?
            BE     TRAIBLAN
            CI, 4  C '.'
            BE     TRAIPOIN
      PROGRESSER AI, 3  1          } il s'agit du module décrit
                                   } ci-dessus.
                 CI, 3  80       }
                 BG     ERREUR   }
      RETOUR      B     TESTS
      TRAIBLAN   AI, 2   1        traitement du caractère '␣'
                  B     PROGRESSER
      TRAIPOIN   AI, 2   1
```

module INIT 2 :

```
            LI, 2  0
            LI, 3  0
```

La modularité a surtout été utile pour l'analyse, et pour isoler des séquences faciles à écrire. L'absence de langage adapté nous oblige à rassembler les différents modules "à la main" pour pouvoir en constituer un programme.

Ceci aurait pu s'écrire simplement sous une forme qui est en fait la spécification de la solution et qui serait sa description en Civa :

```
Pour chaque   CARTE de PAQUET   faire
    Pour chaque X de CARTE  tel que X = BLANC
                            tant que X ≠ POINT
                faire MOT = MOT + ! fpc ;
    écrire (MOT) fpc ;
```

REFERENCES DU CHAPITRE 15

DENDIEN M. C., 71

Simulation de machines pour l'enseignement de la programmation. Thèse 3e cycle. Univers. NANCY 1, 1972.

DENDIEN M. C. et DERNIAME J. C., 72

Génération de programmes de simulation de machines R.A.I.R.O., B.1, 1972.

CHAPITRE 16

ORGANISATION GENERALE DE LA REALISATION

CHAPITRE 16

ORGANISATION GENERALE DE LA REALISATION

16.1 INTRODUCTION.

La réalisation décrite n'a pas pour but de livrer un produit complet et utilisable industriellement de façon fiable mais simplement de montrer que les propositions avancées dans le projet ne sont pas qu'une vue de l'esprit et qu'il est possible de les réaliser en obtenant un programme performant, c'est-à-dire qu'il est possible de construire ce produit complet et fiable.

La responsabilité de l'organisation générale de la réalisation a été spécialement confiée à Jacques Ducloy. Son rôle a été essentiellement d'analyser la mise en oeuvre effective de cette réalisation et de proposer un système de programmation permettant de dégager les autres membres de l'équipe ayant à intervenir dans la réalisation, des problèmes liés au matériel et au système d'exploitation utilisé (CII 10070 sous Siris 7) ainsi que des problèmes d'interface des constituants, liés à l'organisation générale de la réalisation. Il a pour cela, proposé des normes d'écriture de l'application "réalisation" en utilisant au maximum les concepts mis en évidence dans Civa : modularité de la description des actions et des informations, programmation en mode déclaratif, facilité de maintenance, etc...

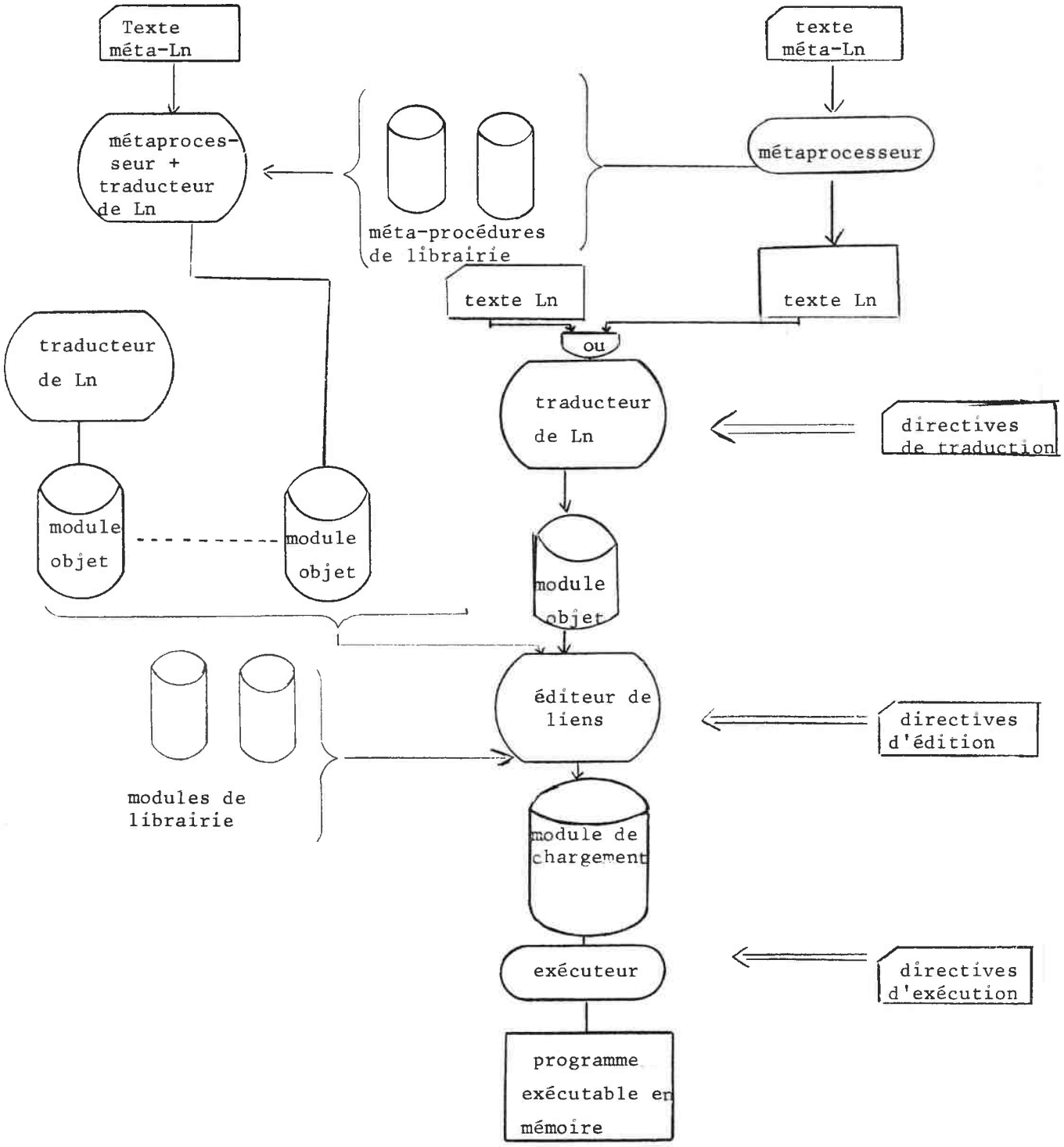
Bien que tous les points de cette organisation générale aient été discutés et parfois modifiés au cours des réunions de travail de l'équipe, il s'agit là essentiellement de son travail personnel, dont il a d'ailleurs rendu compte dans (Ducloy J. 73). Nous nous contenterons donc ici d'évoquer les grandes lignes de cette organisation en précisant les points importants pour la compréhension des chapitres suivants.

16.2 SYSTEME DE COMPILATION SOUS SIRIS 7.

Le système Siris 7 est un système classique dans lequel les identificateurs rencontrés dans les programmes sont remplacés par des adresses (ici ce seront des adresses virtuelles) avant de passer à l'exécution : il n'y a pas d'édition de liens dynamique.
Pour permettre le partage d'objets entre des unités compilées séparément, SIRIS 7 fait intervenir un éditeur de liens statique.
Tous les textes sources sont traduits dans le même langage intermédiaire, appelé langage-objet, pour constituer des modules-objets. Un module-objet contient les références non satisfaites : références externes et références en avant. Il est constitué de définitions externes et d'une suite de directives de chargement destinées à l'éditeur qui les interprète, pour constituer un module de chargement dans lequel toutes les adresses sont résolues par rapport à une origine unique et dans lequel il ne subsiste plus de références non satisfaites.

Le système de protection de CII 10070 (protection par clé et verrous associés aux pages de la mémoire) amène à créer dans chaque module-objet des sections différentes par type de protection. Le langage d'assemblage Symbol, le métalangage Métasymbol et bien sûr le langage des modules-objets permettent d'indiquer dans quelle section générer chaque partie du texte source. Un système de compilation sous SIRIS 7 est donc caractérisé par le schéma suivant :

Schéma général d'un système de compilation sous SIRIS 7.



16.3 SYSTEME DE COMPILATION CIVA.

16.31 Compilation Civa sous Siris 7.

La définition du projet Civa comme un ensemble complet d'outils permettant de créer et d'exécuter des chaînes de traitement, devrait conduire logiquement à la réalisation d'un système d'exploitation propre aux applications Civa. Nous avons préféré, étant donné l'objectif de cette réalisation, utiliser Siris 7, quitte à introduire localement des "astuces" pour contourner les difficultés dues aux limitations de Siris 7, plutôt que de définir un nouveau système d'exploitation acceptant néanmoins les textes Cobol, Fortran, Algol... etc, de Siris 7.

16.32 Le compilateur.

La compilation d'un module d'une application conduit à la production :

- d'un module-objet contenant des "constantes" : traduction des instructions et constantes du texte source. Il sera protégé contre l'écriture.
- une déclaration de référence externe $ < nom du module > repérant une zone destinée à contenir les éléments variables locaux à ce module.
- un enregistrement dans le fichier descriptif de l'application indiquant la liste des classes utilisées ; la liste des module appelés, la taille du module-objet construit et la taille de la zone variable "$ nom".

La compilation d'une classe conduit à la production d'un module objet si elle contient des constantes (constantes ou procédures), à une référence externe du type $ nom, si elle contient des variables, et, dans tous les cas, à une table des identificateurs déclarés dans la classe et un enregistrement dans le fichier descriptif de l'application, indiquant la taille du module-objet et celle de la zone "$ nom", ainsi que la liste des classes utilisées.

Les différentes unités, modules ou classes, sont compilées séparément. La compilation se déroule en deux phases :

- codification du texte source, conduisant à un texte en langage intermédiaire, la "chaîne codée", dans lequel les problèmes d'adressage ont été résolus.
- génération qui partant de la chaîne codée construit un module objet.

16.33 Relations entre modules et classes.

16.331 Relation "appelle".

Deux modules peuvent être reliés par la relation "appelle". Deux modules sont compilés séparément il doit donc exister un lien entre eux : le nom d'un module appelé sera une référence externe. Pour chaque module existera donc une définition externe : le nom du module, et autant de références externes qu'il y a de modules différents appelés.

Tous les appels de modules seront réalisés par une instruction de branchement à un sous-programme utilisant le même registre de liaison : BAL, 15.

16.332 Relation "utilise".

Le texte d'un module M peut contenir une occurrence de tout identificateur dans $\hat{U}$ (m) (cf. 1. 4). On pourrait songer à repérer chaque identificateur par une référence externe, mais outre le nombre très élevé de références externes que cela entraînerait, il y aurait trop de risque de confusion : deux classes pouvant déclarer le même identificateur. Les classes seront donc compilées.

La compilation d'une classe consiste essentiellement à attribuer une place à chacun des objets relativement à l'origine d'une zone :

- zone des constantes, repérée par une définition externe du nom de la classe : elle contient les constantes et les procédures.
- zone des variables, repérée par une définition externe : $ < nom de la classe > , elle conduira tous les objets dont la valeur sera fixée à l'exécution.

En fait la zone "$ nom" est donc vide jusqu'au moment de l'exécution, il ne s'agit que d'une réservation. La définition de $ nom peut donc être retardée.

La compilation d'un "utilise" dans un module ou une classe, conduit donc à la production dans le module-objet d'une référence externe au nom de la classe et d'une référence externe en "$ nom" de la classe. La table des symboles de la classe "utilisée"étant jointe à celle de l'unité en cours, comme nous le verrons en étudiant la codification, toute occurrence d'un identificateur de la classe "utilisée" donne lieu à une adresse relative à une référence externe (translation).

16.34 Ordre des compilations.

Pour pouvoir compiler une classe ou un module m, il faut donc que les classes de $\hat{U}$ (m) aient déjà été fournies, c'est-à-dire qu'on en possède le texte source. Pour éviter de répéter des compilations d'une même classe, il est même souhaitable qu'elles soient déjà compilées.

A la demande d'exécution d'une chaîne de traitement (apparition du nom de son premier module dans un module de commande) toutes les classes susceptibles d'être utilisées par cette chaîne doivent logiquement être présentes. Nous attendrons donc un tel moment pour lancer des compilations.

L'ordre des compilations est déterminé grâce au graphe de la relation "utilise" : $\hat{U}^{-1}$(m), ou m est le premier module de la chaîne, est un pré-ordre qui impose les contraintes à respecter : on en déduit l'ordre des compilations en ordonnant arbitrairement les éléments équivalents.

Lors de l'apparition dans le texte d'un module de commande, d'un texte de module ou de classe, l'enregistrement associé dans le fichier descriptif des classes et modules de l'application est crée, les "utilise" sont notés. $\hat{U}^{-1}$ (m) pourra donc être construit à partir du fichier descriptif.

16.35 Pré-édition de liens.

Les zones de constantes pourraient être implantées dans le module de chargement correspondant à une chaîne les unes à la suite des autres. De même pour les zones de variables "$ nom".

Les objets que contient une de ces zones ont en général la même durée de vie, la durée d'exécution du premier module qui les utilise (quelquefois un peu moins, s'ils sont crées pendant l'exécution de ce module cf. I . 5), on peut donc parler de durée de vie d'une zone.

Or les zones ont des durées de vie différentes et il est normal de chercher à en tirer partie pour déterminer les recouvrements possibles en mémoire et diminuer ainsi la taille du module de chargement. Pour les zones de constantes, de tels recouvrements nécessitent des chargements de "segments" en cours d'exécution, nous ne les envisagerons donc qu'en cas de nécessité (place disponible en mémoire insuffisante). Pour les zones de variables, les recouvrements peuvent être déterminés pendant l'exécution (pile dynamique d'Algol), mais cela allonge l'exécution de façon non négligeable. En rappelant l'hypothèse faite au chapitre I selon laquelle les chaînes de traitements sont suffisamment répétitives pour que l'on puisse passer du temps à préparer une exécution, à condition que celle-ci soit la plus rapide possible, il est naturel d'envisager de déterminer statiquement ces recouvrements (1).

(1) Ainsi en Algol, il est impossible de déterminer l'emplacement qui sera occupé par une variable déclarée dans une procédure, ou d'un tableau de taille variable (quel que soit l'endroit de sa déclaration). L'adresse d'un objet de ce type est alors un couple formé de l'adresse de l'origine du segment qui le contient et de l'adresse ralative de l'objet dans ce segment. Ce couple est transformé en une adresse pendant l'exécution, par indexation par exemple. L'accès à un objet est donc plus long que si on avait pu obtenir une adresse à la compilation. En Civa, l'absence de récursivité fait qu'il est possible d'étudier statiquement toutes les relations entre les modules et les classes et donc, pour les objets de taille fixe, d'obtenir des adresses dès le stade de la compilation.

C'est le rôle du pré-éditeur de liens, qui sera étudié au chapitre 25, de déterminer statiquement, c'est-à-dire avant l'exécution de la chaîne, les emplacements relatifs des zones intervenant dans une chaîne et d'imposer les emplacements à l'éditeur de liens.

16.36 Schéma du système de compilation Civa.

L'éditeur de liens est celui de Siris 7. Il dispose d'une librairie "Civa". Les textes sources et les textes objets des classes et des modules sont conservés, ainsi que les tables des identificateurs déclarés dans chaque classe, ce qui constitue une librairie au niveau de la compilation.

La pré-édition de liens produit un module-objet contenant toutes les déclarations de définition externe du type $ nom, ainsi que des consignes à l'éditeur.

L'exécuteur est celui de Siris 7.

Aucune directive n'est plus nécessaire.

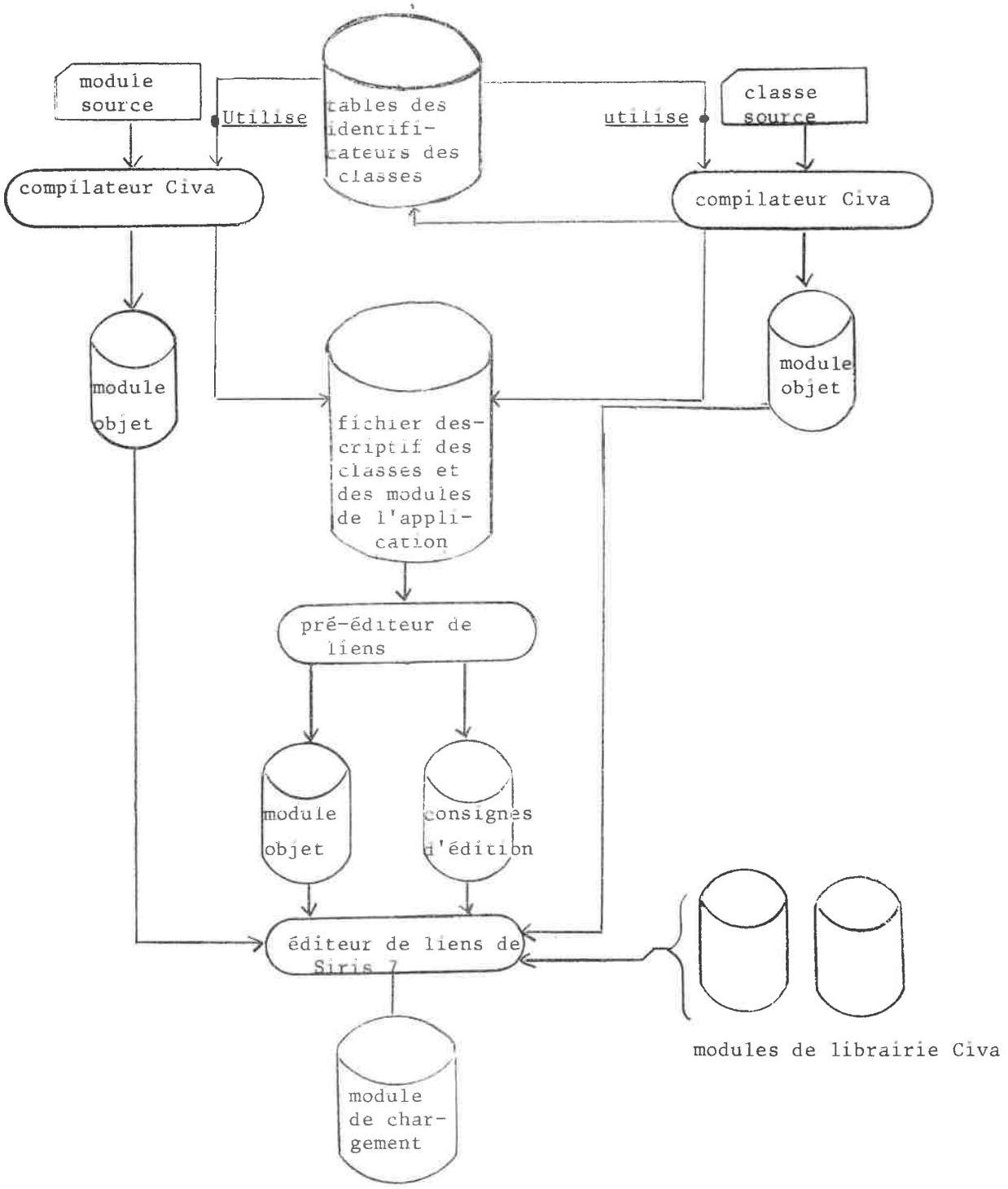


Schéma du système de compilation Civa.

16.37 Exemple récapitulatif (Ducloy J., 73)

Dans cet exemple, afin de simplifier les modules-objets, on n'a pas tenu compte systématiquement des sauvegardes de registre. Les modules-objets sont écrits en équivalent Métasymbol.

```
module   m1 ;
utilise  c1 ;
entier   x ;
i = 1 ;
m2 ;
x = i ;
fin module ;
```

1

```
module  m2 ;
utilise c2 ;
entier x ;
i de c1 = i de c2
k = n ; égal ;
carré ; cube ;
x = t ;
fin module ;
```

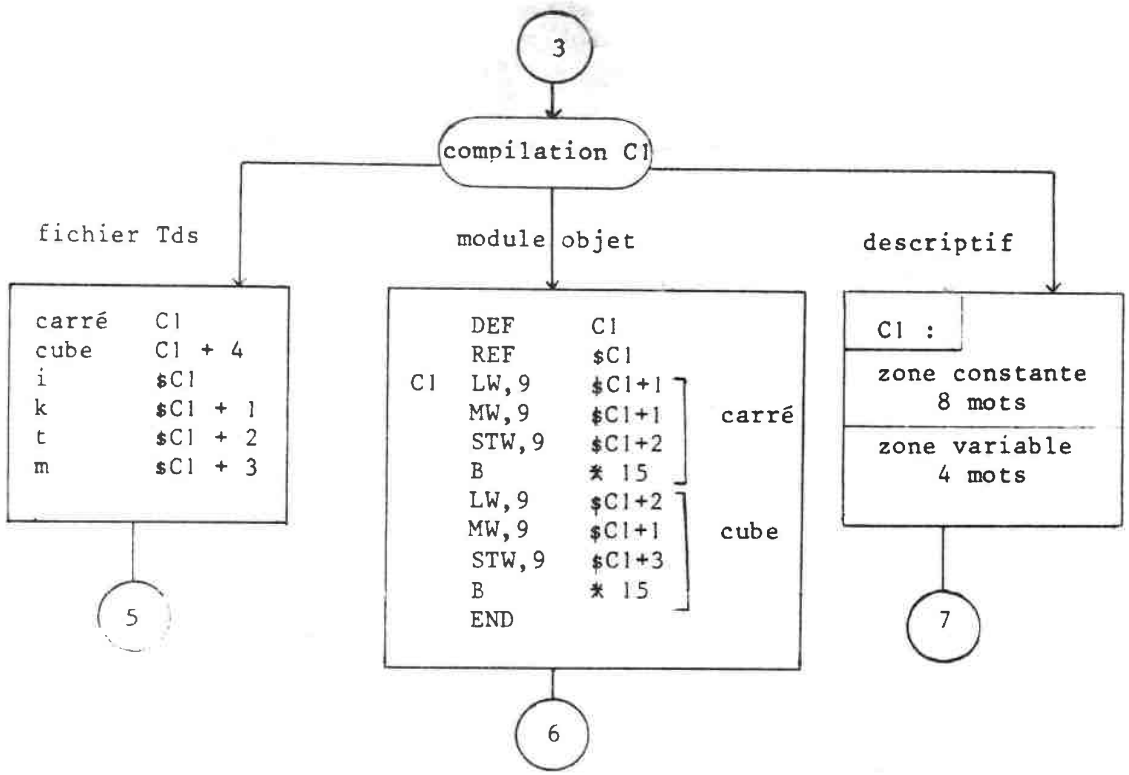
2

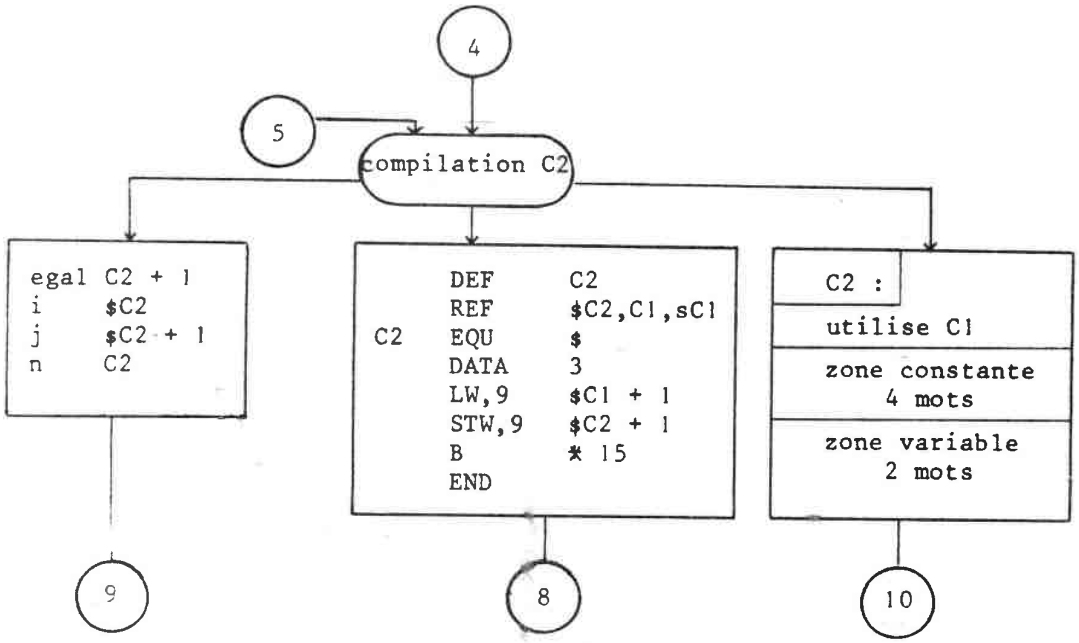
```
classe c1 ;
entier i, k, t, m ;
procédure carré ;
t = k*k ;

procédure cube ;
n = t*k ;
fin classe
```

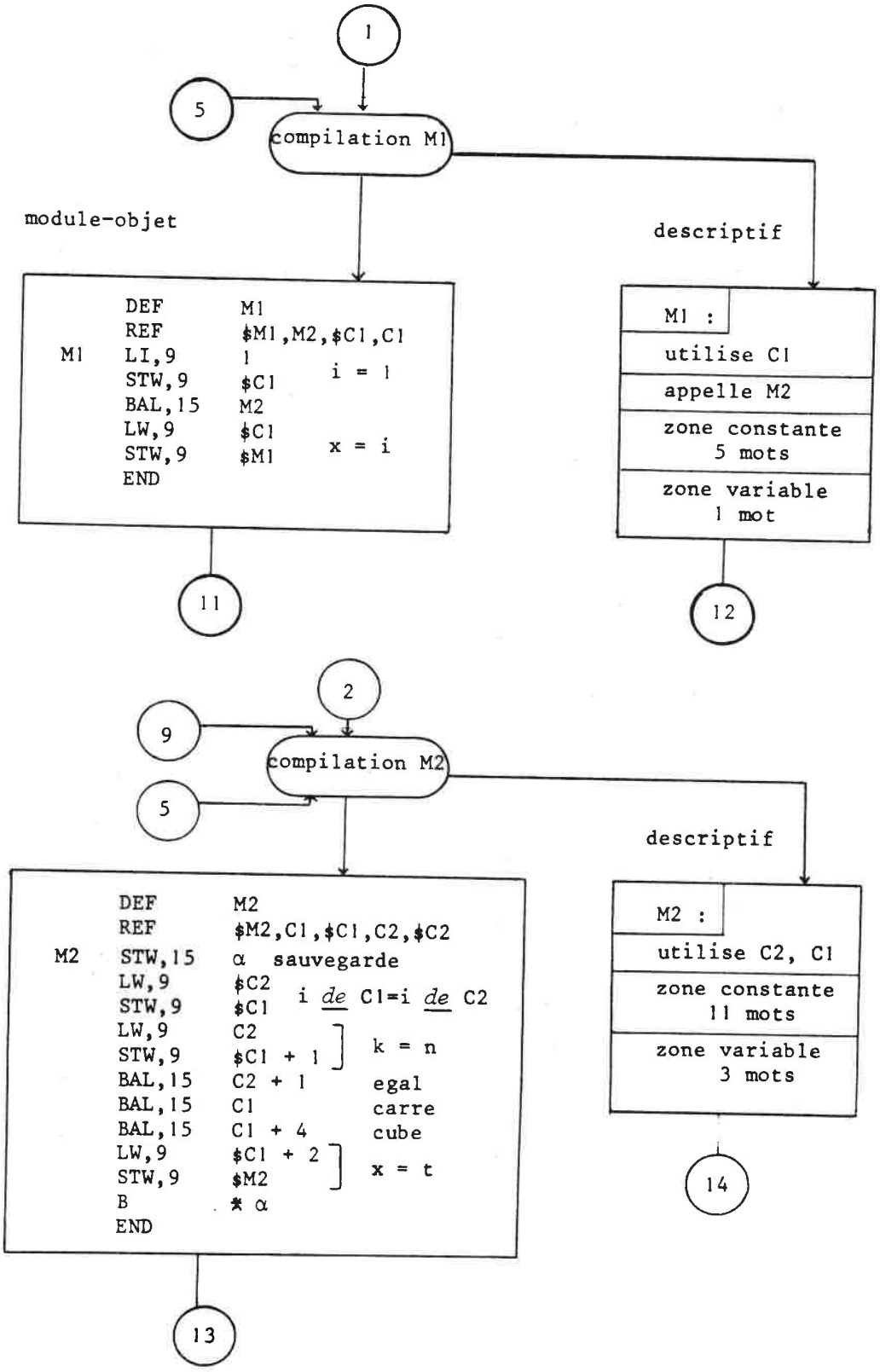
3

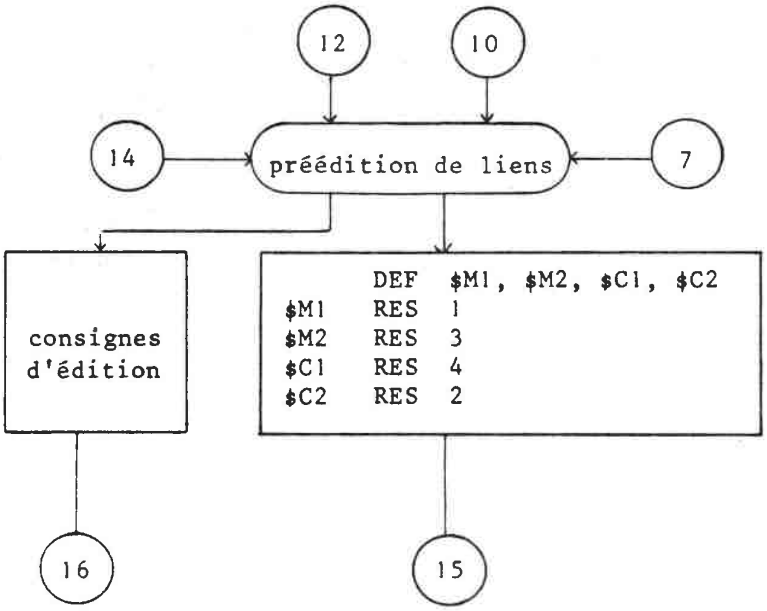
```
classe c2
entier j, i ;
entier n = 3 ;
utilise c1 ;

procédure égal ;
j = k
fin classe
```

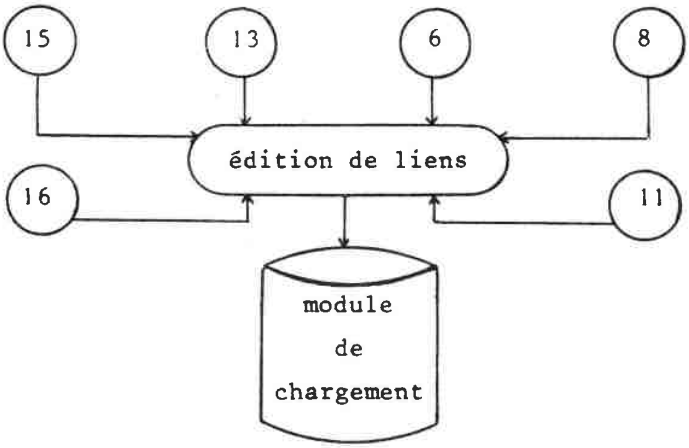
4


3
compilation C1
fichier Tds
module objet
descriptif
carré C1
cube C1 + 4
i $C1
k $C1 + 1
t $C1 + 2
m $C1 + 3
DEF C1
REF $C1
C1 LW,9 $C1+1
MW,9 $C1+1
STW,9 $C1+2
B * 15
carré
LW,9 $C1+2
MW,9 $C1+1
STW,9 $C1+3
B * 15
cube
END
C1 :
zone constante
8 mots
zone variable
4 mots
5
6
7


4
5
compilation C2
egal C2 + 1
i $C2
j $C2 + 1
n C2
DEF C2
REF $C2,C1,sC1
C2 EQU $
DATA 3
LW,9 $C1 + 1
STW,9 $C2 + 1
B * 15
END
C2 :
utilise C1
zone constante
4 mots
zone variable
2 mots
9
8
10


1
5
compilation M1
module-objet
descriptif
DEF M1
REF $M1,M2,$C1,C1
M1 LI,9 1
STW,9 $C1
i = 1
BAL,15 M2
LW,9 $C1
STW,9 $M1
x = i
END
M1 :
utilise C1
appelle M2
zone constante
5 mots
zone variable
1 mot
11
12
2
9
5
compilation M2
descriptif
DEF M2
REF $M2,C1,$C1,C2,$C2
M2 STW,15 α sauvegarde
LW,9 $C2
STW,9 $C1
i de C1=i de C2
LW,9 C2
STW,9 $C1 + 1
k = n
BAL,15 C2 + 1
egal
BAL,15 C1
carre
BAL,15 C1 + 4
cube
LW,9 $C1 + 2
STW,9 $M2
x = t
B * α
END
M2 :
utilise C2, C1
zone constante
11 mots
zone variable
3 mots
13
14


12
10
14
préédition de liens
7
consignes
d'édition
DEF $M1, $M2, $C1, $C2
$M1 RES 1
$M2 RES 3
$C1 RES 4
$C2 RES 2
16
15


15
13
6
8
édition de liens
16
11
module
de
chargement

16.4 ECRITURE DU SYSTEME DE COMPILATION.

16.41 Choix de Métasymbol.

Pour CII 10070, nous avions la possibilité d'écrire le système de compilation :

- en langage "évolué" Algol, Fortran, Cobol
- dans le langage machine d'une machine hypothétique pour lequel il suffirait d'écrire un interprète ou unassembleur pour 10070, ou pour une autre machine puisque cette solution offre l'avantage d'être disément "portable" ,
- en langage d'assemblage Symbol, ce qui est long et ne permet pas facilement de bien structurer la réalisation, mais conduit à des textes réputés efficaces.
- en LP 70, intermédiaire entre un langage d'assemblage et un langage évolué : les performances sont analogues à celles d'un texte écrit en langage d'assemblage, mais l'écriture est beaucoup plus facile et plus lisible. L'existence de procédures permet une structuration stricte de la réalisation.
- en Métasymbol, langage de méta-assemblage sur 10070 ; la modularité de la réalisation apparaît moins évidente, par contre celle de son écriture est grandement favorisée par l'existence de procédures (analogues aux métamodules du chapitre 9) et de métafonctions. Toute action ou information étant décrite par une procédure, les modifications dans le texte de la réalisation sont très faciles. D'autre part, et surtout, il permet de dégager tous les participants à l'écriture de la réalisation de tous les détails technologiques gênants, en les reportant sur une seule personne qui écrit les procédures correspondantes. N'ayant à écrire ces séquences de code qu'une seule fois, cette personne pourra y passer plus de temps et optimiser le texte obtenu.

Enfin, Métasymbol se prête assez bien à la définition de concepts analogues à ceux rencontrés dans Civa : les modules et les classes et les métamodules. Notre choix s'est donc porté sur Métasymbol.

16.42 X modules, X classes, X métamodules.

Un certain nombre d'identificateurs devront être réservés à l'usage des procédures du système d'écriture : ils commencent par X ou Z, les autres identificateurs commençant par une autre lettre. Les X métamodules décrivent la construction de textes de la réalisation : ce sont des procédures de Métasymbol.
Un X module est une unité de description d'action du compilateur. Un X module peut en appeler un autre. Les opérations à effectuer lors d'un appel sont toujours les mêmes (sauvegarde de registres, transmission de paramètres) ; de même pour les opérations de début et de fin de module : trois X métamodules sont donc prévus pour produire les textes correspondants :

X module qui définit un début de X module,
X fin mod qui définit la fin d'un X module,
X call qui réalise un appel de X module.

L'emploi d'une directive "utilise C" peut être considérée, en Civa, comme une demande de copie du texte de la classe (à cet endroit, en précisant que les objets ainsi déclarés sont les mêmes que ceux déclarés ailleurs par une directive identique. Remarquons que cette restriction ne porte que sur les objets occupant une place à l'exécution : pour les métamodules et métafonctions déclarés dans la classe, elle est inutile.

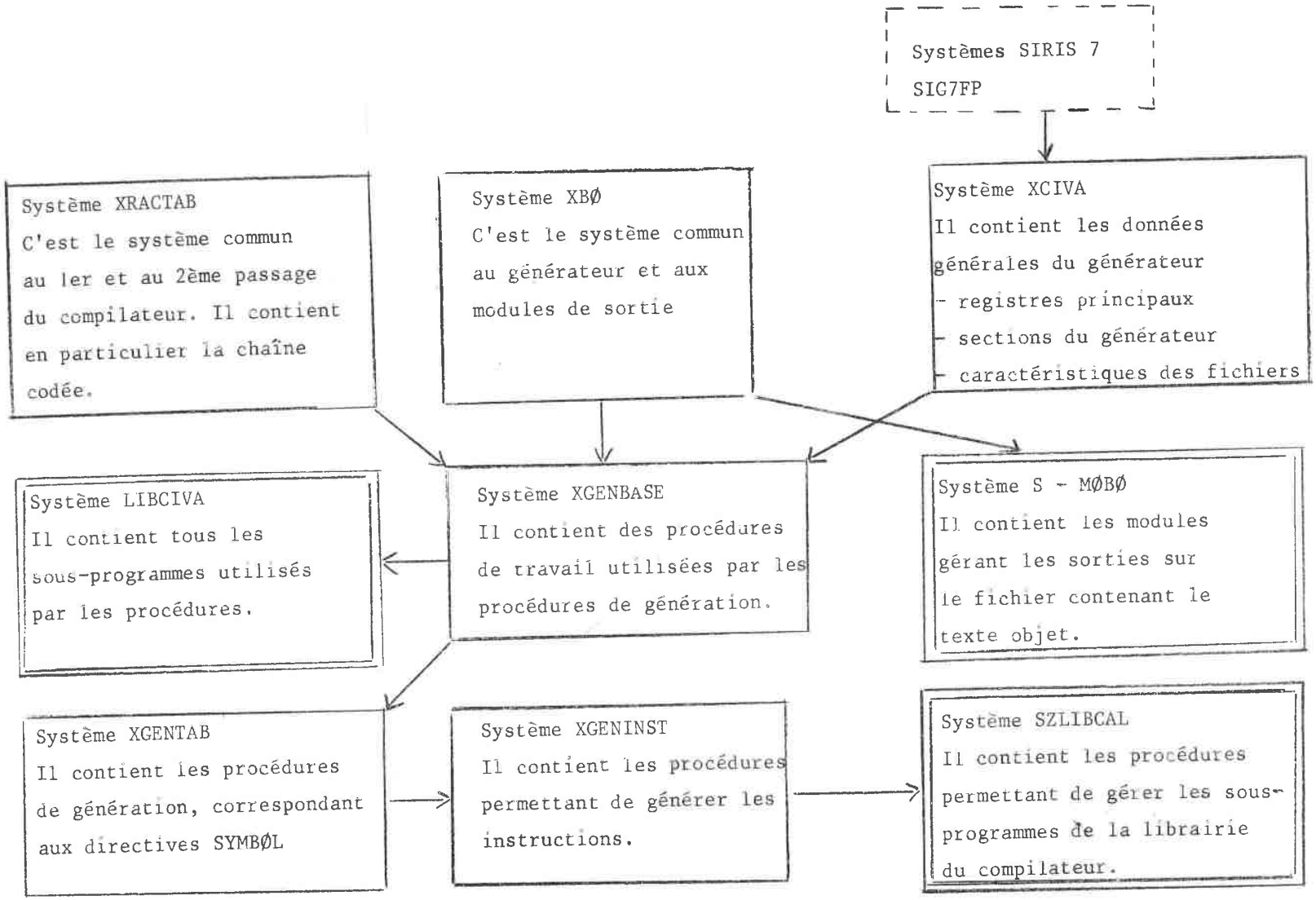
La directive SYSTEM de Métasymbol (CII.1) permet d'insérer dans le texte d'un programme, une suite de lignes Métasymbol préalablement définie dans une partition d'un fichier. Une section fictive de Métasymbol est telle que si plusieurs modules-objets contiennent chacun une section fictive de même nom, celle-ci ne figure qu'une seule fois dans le module de chargement produit.
Pour définir l'équivalent d'une classe, une X classe, il suffit de constituer une suite de lignes de Métasymbol et la ranger dans une partition ayant le nom de la classe, les déclarations d'emplacement figurant dans une section fictive de même nom.

La X classe "système" comporte toutes les déclarations des X métamodules tels que X module, X call...., les définitions de registres etc... Elle doit être utilisée par tout X module. Elle a pour nom X Civa.

16.43 Organisation générale.

Pour permettre une écriture facile de l'ensemble de la réalisation nous avons donc été amenés à définir un certain nombre de X classes, des "systèmes" au sens de métasymbol. Ils sont décrits en partie dans (Ducloy J., 73) et dans (Viault D., 73). Nous nous contenterons ici d'en évoquer le schéma général, en insistant sur le fait qu'il s'agit là d'un travail de préparation très important, mais qu'il se montre très efficace pendant l'écriture de la réalisation, et de donner deux exemples de procédures (leur spécification en fait).

16.431 Schéma général.



X → Y signifie que le système X est utilisé par le système Y.

X Le sytème X est utilisable sous forme de langage source. Il doit être appelé avant la compilation du programme.

X Le système X est directement utilisable sous forme objet. Il doit être joint un programme pour l'édition de liens.

16.432 Exemples de procédures.

L'ensemble des procédures est décrit dans (Viault D., 73)

a) procédure XSØRTBØ

Il s'agit d'une procédure de la X classe XGENBASE
Elle a pour but de gérer le tampon de sortie du module objet.
Elle insère dans ce tampon (décrit dans la X classe XBØ)
une information reprérée par son type et sa localisation.
Le type, octet, chaîne d'octets, chaîne d'octets précédée
d'un octet de comptage ou mot, est précisé par un code dans
la zone de commande (BYTE, CHAIN, TEXTC, WORD)

XSORTBØ, BYTE (VAL, 5)

Pour chaque type, des codes ont été définis pour décrire
la localisation. Nous nous contenterons, ici, de donner des
exemples.

XSORTBØ, BYTE (ADWI, POINTEUR, (DB,7), (AI, 3))

permet de générer dans le module objet un octet dont la
valeur est

- celle de l'octet d'adresse "contenu de POINTEUR" + 7
- + 3

XSØRTBØ, WØRD ✳ P, 5

permet de générer le mot dont l'adresse est le contenu de P
augmenté du contenu du registre 5.

XSORTBO, TEXTC (ADWI, POINT, (IBR, 5)) , (VAL, 40)

demande la génération de la chaîne, précédée de son octet de
longueur, qui commence à l'adresse d'octet obtenue en ajou-
tant au contenu de POINT, celui du registre 5 et dont la
longueur est de 40 octets.

Chacun de ces appels de procédure de Métasymbol, écrits dans
le compilateur Civa, est transformé, à l'assemblage du
compilateur en la suite d'instructions la mieux adaptée au
cas particulier, permettant de générer dans un module-objet
ce qui est demandé. Il s'agit souvent d'appels de sous-program-
mes décrits dans LIBCIVA et, lorsque cette solution s'avère
plus courte, d'une suite d'instructions réalisant toute la
génération.

b) Procédures de génération d'instruction

Il s'agit de procédures de XGENINST qui utilisent donc la procédure précédente. Elles permettent de générer une "instruction 10070", c'est-à-dire, en fait, une rubrique de chargement contenant une instruction.
Les paramètres doivent préciser le code de l'instruction, le registre utilisé, éventuellement le registre d'index utilisé et la marque d'adressage indirect, enfin la partie adresse qui est caractérisée par un numéro de déclaration et une translation par rapport à cette déclaration et, éventuellement, le type de résolution de cette adresse. La procédure XGINABS est utilisée dans le cas où l'adresse est absolue :

XGINABS, LI, 1 (VAL, 3)

permet de générer "LI, 3"

XGENABS, (ADB, A, (IBR, 7), (ADB, B) (REG, 6)

La procédure XGINSFR permet de produire une séquence de génération d'instruction dans le cas où l'adresse est celle d'une référence en avant. La procédure XGINSDEC est utilisée quand l'adresse est définie par rapport à une déclaration.

XGINSDEC, (ADB, A (IBR, 7) , (ADB, B) ((VAL,0)),(REG,Rx1)

correspond au cas où le code se trouve dans une table de codes commençant à l'adresse d'octet A. Il s'y trouve au rang contenu dans le registre 7 ; le numéro du registre utilisé est à l'adresse d'octet B, l'adresse est celle d'une déclaration (définition externe par exemple) dont le numéro est dans le registre RX1 (pas de translation par rapport à cette référence). Cet appel de procédure pourrait donc être utilisé pour construire le texte d'un sous-programme du générateur B, RX1 et le Registre 7 étant supposés garnis à l'appel de ce sous-programme.

16.44 Maintenance de la réalisation.

16.441 Sauvegardes.

Les modules-objets associés au compilateur sont regroupés dans un fichier partitionné, chaque partition ayant comme nom celui du X module. La même organisation est retenue pour les textes sources sous forme compressée (CII . 1)

Deux versions du fichier des modules-objets existent : une version de travail et une version témoin. Les essais et mises à jour sont effectuées avec le fichier de travail uniquement. Une modification de la version témoin n'est effectuée qu'à partir de la version de travail et uniquement lorsque celle-ci est complète.

Il est également entretenu un fichier "archives" contenant tous les textes qui ont figuré dans la version témoin.

Essais et mise à jour sont réalisés une seule et même personne, le "secrétaire de la réalisation" comme nous le verrons en 16.5.

16.442 Modification des textes.

Une modification devant être effectuée sur un texte peut être locale à un X module et peut même être locale à une version d'un essai : les aides à la mise au point de Siris 7 permettent de faire cette modification.

Si cette modification touche une Xclasse, elle peut avoir des répercussions sur les textes objets des X modules qui utilisent cette classe. Il est alors nécessaire de retrouver tous ces X modules. Des dispositifs, analogues à ceux vus au chapitre 20 permettent de gérer automatiquement toutes les mises à jour nécessaires. Ils ont été décrits dans un document technique (Cridlig A. 72).

16.5 L'EQUIPE DE TRAVAIL CIVA.

L'équipe travaillant au projet Civa était à l'origine essentiellement composée d'enseignants universitaires et de deux ingénieurs du service d'exploitation de l'Institut Universitaire de Calcul Automatique de Nancy. Le projet ne constituait pour aucun d'entre nous son travail principal. Les difficultés de communication entre des personnes travaillant à mi-temps, voire à tiers-temps, dans des endroits différents (on a pu compter 6 localisations différentes en 1973) sont évidentes. A ce sujet voir aussi (Dijkstra,

D'autre part, universitaires, nous sommes plus enclins à réflèchir à des problèmes de conception - toute l'équipe l'a fait, ce qui a conduit à de nombreuses et longues discussions - et à fouiller des cas "intéressants" plutôt qu'à produire efficacement des lignes de programmes, ce qui est assez naturel et expliqué également par des nécessités de carrière. Il est alors difficile d'éviter le risque d'hétérogénéité et les divergences entre les diverses publications et de faire en sorte qu'au contraire, toutes les énergies soient employées à la définition et la réalisationd'un d'un projet commun.

Cette situation nous a amenés à définir précisément les principes d'organisation de la réalisation que nous venons de résumer. Elle nous a également conduit à demander l'aide du Comité de Recherche en Informatique, qui nous l'a accordée sous forme d'un contrat (n° 73 004), ce qui a permis alors d'employer deux personnes à temps plein et quelques vacations. Elle nous a conduit enfin à structurer l'équipe de réalisation elle-même, de façon, plus précise. Nous avons alors rejoint progressivement les idées de Baker et Mills (Baker F. T., 72) et (Mills H. D., 72).

Les universitaires continuent leur activité de recherche consistant à étudier complétement un aspect du projet et à faire des propositions - (sont actuellement en cours l'analyse des données et leur contrôle à l'entrée en ordinateur, la production de la documentation des chaînes de traitement, la définition d'une "méthode d'analyse" utilisant les concepts de Civa, les problèmes liés à l'existence d'objets de taille variable). Ils participent à l'analyse de la partie de la réalisation en rapport avec leurs études.

Parallèlement, une cellule de programmation est chargée de la réalisation. Elle est composée d'étudiants vacataires payés pour écrire des programmes et de deux personnes à temps plein. Elle est organisée autour du "responsable de la réalisation".

Le travail principal de ce responsable est de concevoir mais aussi de coder des programmes. Tous les autres membres de la cellule s'en réfèrent à lui. Il code les X modules centraux du système d'écriture de la réalisation et définit avec les autres membres de l'équipe les X modules qu'ils ont à réaliser. Les

textes écrits par d'autres sont revus et incorporés dans la version en cours de développement sous sa responsabilité immédiate.

Le "secrétaire" de la cellule de programmation entretient la version témoin du système de compilation, c'est-à-dire l'ensemble des X classes, X modules, X métamodules, de leur documentation et des jeux d'essai. C'est lui qui est chargé de la constitution des fichiers partitionnés des textes sources et des modules objets et de celle du fichier descriptif de la réalisation (analogue au fichier descriptif d'une application). Il entretient le fichier "archives".

Il est responsable de tous les essais d'exécution qui lui sont demandés par les autres membres de l'équipe (sauf bien sûr lorsqu'il s'agit d'un texte isolé).

Il est donc pratiquement le seul à devoir se préoccuper de l'interface avec Siris 7.

Responsable de la réalisation et secrétaire devraient permettre de coordonner les efforts des membres de l'équipe en évitant toute redondance et atteindre ainsi une plus grande efficacité.

Dans le cadre du contrat CRI cité ci-dessus, les deux personnes employées à temps plein ont joué ces rôles : l'une de responsable de la réalisation, l'autre de secrétaire.

REFERENCES DU CHAPITRE 16

BAKER F. T, 72 — Chief programmer Team Management of Production programming
IBM Systems Journal, vol. 11, n° 1, 1972.

C. I. I. 1 — Métasymbol sous Siris 7/Siris 8
C. I. I. Réf. 3851 E/Fr.

CRIDLIG A., 72 — Conséquences d'une modification d'un texte source dans une application Civa ou Métasymbol. Document interne Nancy 1972.

DIJKSTRA — The structure of THE multiprogramming system.
CACM, 11.5 (1968).

DUCLOY J., 73 — Compilation dans le projet Civa.
Doctorat d'Ingénieur - Université de Nancy - Mars 1973.

MILLS H. D., 72 — The impact of structured programming on software engineering and production.
Chapitre français de l'A. C. M. 1972.

VIAULT D., 73

CHAPITRE 17

LA CODIFICATION

17.1. INTRODUCTION.

17.2. LE CODIFIEUR.

17.21. Différents segments d'une unité.

17.22. Constitution des tables.

1. ZTAB (table des identificateurs).
2. ZCO (table des valeurs des constantes).
3. ZTCLA (table de chaînage des appels de classe).
4. ZMOCLA (renseignements complémentaires).

17.23. Contenu des tables.

17.24. Constitution de la chaîne codée.

17.3. LE METAPROCESSEUR.

CHAPITRE 17

LA CODIFICATION

17.1. INTRODUCTION.

Nous avons vu au chapitre 16, que la compilation d'un module ou d'une classe est effectuée en deux étapes : codification et génération.

Le codifieur a deux rôles essentiels :

- assurer le "métatraitement", c'est-à-dire l'évaluation des métainstructions et des appels de métamodules et de métafonctions pour obtenir un texte ne comportant plus que des objets CIVA ;
- analyser chaque instruction CIVA pour produire une suite de commandes en langage intermédiaire ; dans cette chaîne tous les identificateurs ont été remplacés par un couple (type, emplacement), où "emplacement" est une "adresse segmentée", c'est-à-dire un couple n° de segment et adresse relative dans le segment.

Ces deux rôles sont remplis par deux parties distinctes de la codification qui sont appelées à tour de rôle (coroutines) : le métaprocesseur et le codifieur proprement dit.

L'étude et la réalisation de la codification a été confiée à Mme MANSUY qui présente son travail dans (MANSUY D., 74).

17.2. LE CODIFIEUR.

C'est lui qui est chargé d'analyser les instructions et de remplacer les identificateurs par des "adresses segmentées". Son travail s'effectue donc en deux étapes : constitution de tables des identificateurs et production d'une chaîne codée. Au fur et à mesure de la constitution des tables d'identificateur, il doit également déterminer quels sont les segments dans lesquels il placera les objets.

17.21. Différents segments d'une unité.

Rappelons que les classes et les modules sont compilés séparément et que le résultat d'une compilation est constitué essentiellement d'un module objet et d'un descriptif (cf. chapitre 16).
Trois types d'objets peuvent être considérés :

- constantes : constantes de langage ou corps de procédures et modules.
  Les objets constants d'une même unité sont regroupés dans un même segment : la zone des constantes. C'est elle qui constituera le module objet proprement dit.

- variables de taille fixe, c'est-à-dire ceux dont l'encombrement peut être connu avant l'exécution ; c'est le codifieur qui le déterminera.
  Ces objets n'ayant pas de valeur initiale, ce sont des emplacements "vides" : ils ne figurent pas dans le module objet.
  Ils sont simplement repérés dans la table des identificateurs : il leur est associé une "adresse segmentée" les localisant dans cette zone.

- objets de taille variable : un pointeur situé dans la zone des objets de taille fixe permet de les situer dans une zone reservée à l'ensemble des objets de taille variable d'une unité de traitement.
  La valeur de ce pointeur est déterminée pendant l'exécution lors de la première attribution de valeur à cet objet. Le codifieur n'a donc pas à se préoccuper de la localisation des objets de taille variable : il repérera simplement l'emplacement du pointeur dans la zone des variables de taille fixe par une adresse segmentée.

La traduction d'une unité peut donc conduire à deux segments différents. Nous verrons, au chapitre 25, comment sont regroupés les différents segments au cours de la phase de reliure.

17.22. Constitution des tables.

Une table d'identificateurs est constituée pour chaque unité, module ou classe. Nous avons vu en 1.2. que, si une unité "utilise" une classe, tous les identificateurs de cette classe sont valides dans cette unité. C'est dire que les tables des classes "utilisées" par une unité devront être jointes (elles seront chaînées entre elles) à celle de l'unité.
Pour chaque classe, le codifieur construit les quatre tables suivantes :

1. ZTAB (table des identificateurs)
2. ZCO (table des constantes)
3. ZTCLA (table des chaînages des appelés de classe)
4. ZMOCLA (table des renseignements sur les identificateurs non simples).

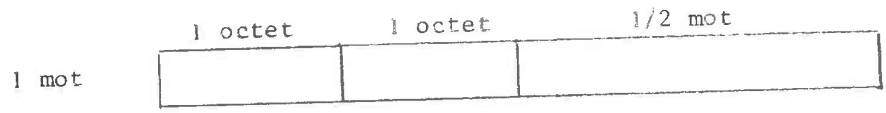
Remarque : le nom de la classe est rangé dans la liste des classes existant pour l'application.

17.221. ZTAB (table des identificateurs).

Cette table est organisée pour utiliser une méthode de recherche par "Hash-coding". On y rencontre par identificateur une zone de 4 mots :

pointeur de table	descripteur	nom d'identificateur	

Le pointeur de table contient l'adresse relative du prochain identificateur de la table appartenant à la même classe d'équivalence que lui, ou il vaut zéro s'il est le dernier identificateur de la table de cette classe d'équivalence.
Chaque identificateur est caractérisé par un descripteur qui peut figurer dans la table des constantes, dans le tableau des identificateurs, et dans la chaîne codée. Dans cette chaîne, c'est lui qui permet de définir le type et la localisation de l'information manipulée directement ou bien indirectement.

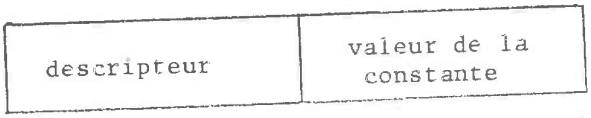


Type s'il est simple non conditionné et structuré (→9 type -1), sinon 0	2 n+1 si constante 2 n+2 si variable n:n° de la classe dans la table des appels de classe, ou 0 si module.	- si 0 dans le 1er octet c'est l'adresse des pointeurs dans la table des pointeurs - sinon localisation par rapport à la zone de la classe.

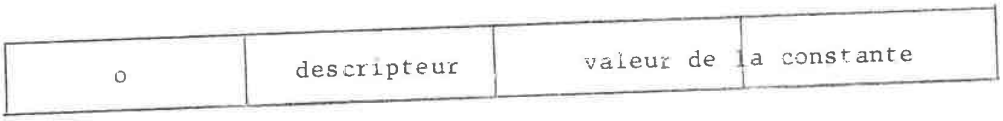
17.222. ZCO (table des valeurs des constantes).

Elle contient deux mots mémoire par constante :

soit



si la constante est un réel ou un entier (si elle tient sur un mot) ; soit

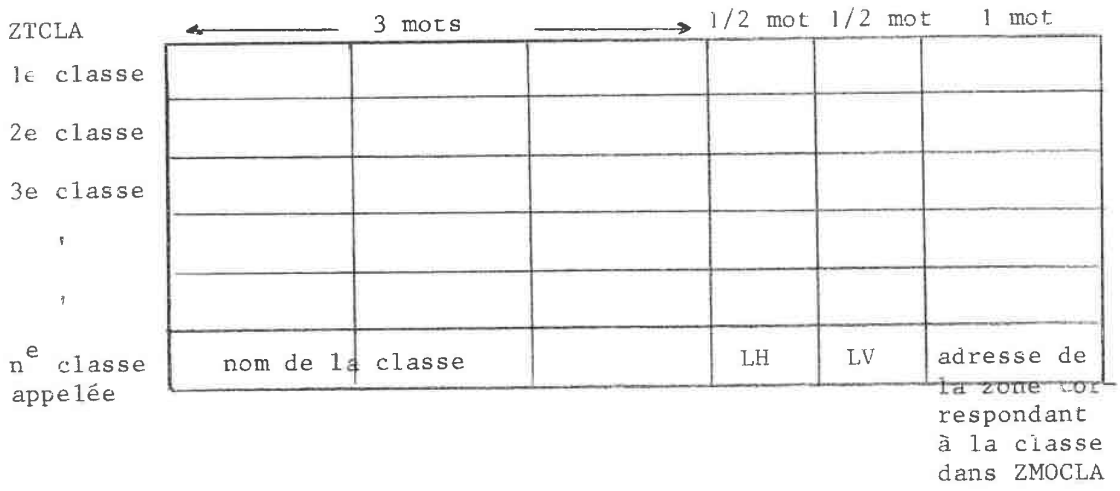


si la constante est en double précision (si elle tient sur 2 mots) le descripteur est le même que l'on retrouve dans la table des identificateurs, si la constante a un nom, il suit le schéma général des descripteurs d'identificateurs.

La table des valeurs des constantes sera jointe par le générateur au module objet : elle ne comporte que les valeurs des constantes rangées bout à bout dans ZCO.

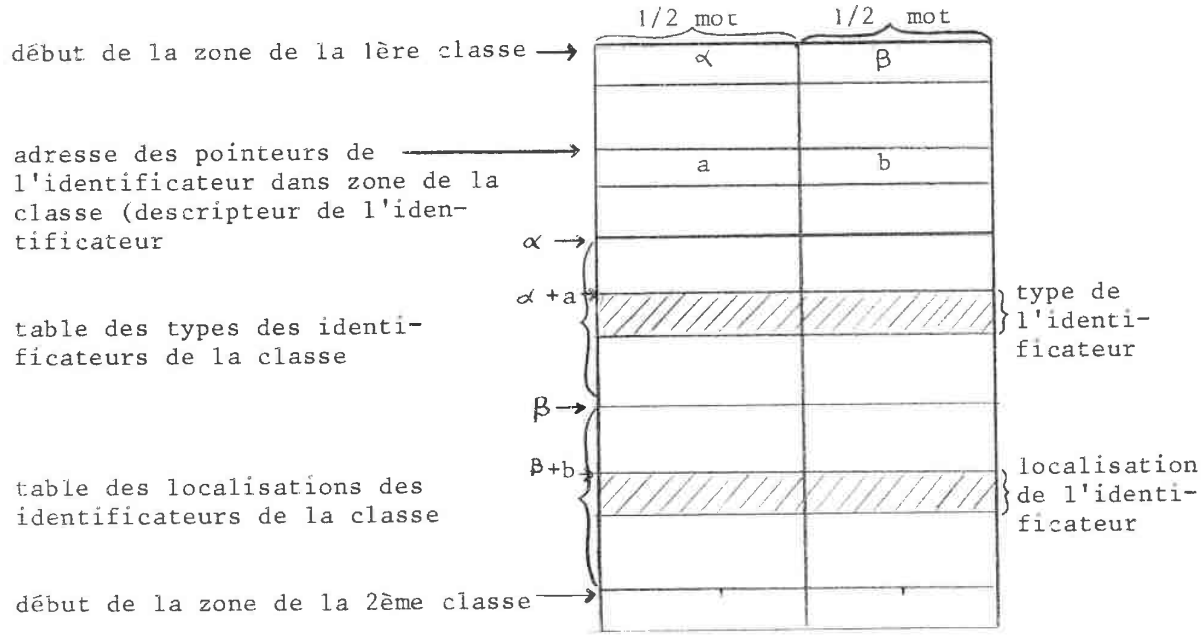
17.223. ZTCLA (table de chaînage des appels de classes).

Cette table sert à donner un numéro à chaque classe utilisée par le module ou la classe, on intègre à cette table la table de chaînage des appels de chaque classe utilisée, en vérifiant que les classes qu'elle contient ne sont pas déjà dans la première classe.



17.224. ZMOCLA (ensemble des renseignements sur les identificateurs non simples des classes appelées).

Elle est constituée de zones (une zone par classe) mises bout à bout



17.23. Contenus des tables.

La table ZMOCLA contient par identificateur un mot contenant deux pointeurs : l'un vers la description de son type, l'autre vers la description de sa localisation.
Si l'identificateur concerné est celui d'un type, il n'a pas de localisation.
Si un identificateur A est déclaré du type déclaré B : A type B (cf. 4.3.), les identificateurs A et B pointeront vers la même description de type, A pointera également vers une localisation et B n'aura pas de localisation.
Si B est un identificateur d'objet et A est déclaré par analogie à B, alors A et B auront même pointeur de type mais pointeront vers des localisations différentes.
On trouvera dans (MANSUY D., 74), la définition précise du contenu de ces tables.

17.24. Constitution de la chaîne codée.

Au cours de l'analyse du texte d'une unité, les identificateurs et les mots-clés du langage sont remplacés par un descripteur. La chaîne codée est donc constituée d'une suite de descripteurs. Pour les mots-clés, un code leur est attribué une fois pour toutes. On pourrait écrire par exemple (cf. 4.27.) :

```
code chaîne codée code (fin chaîne codée,
                        début procédure,
                        appel module ou procédure,
                        contrôle utilisateur,
                        si, alors, sinon, fsi,
                        quand, fin quand,
                        décision, solution, action, fin
                                                table,
                        suspendre, reprendre, libérer,
                                                sortir,
                        pour chaque, de, faire, fin
                                                pour chaque,
                        .....) ;
```

Pour les identificateurs, ils sont tous remplacés par un descripteur (cf. 17.221.)

Si l'objet déclaré est de type simple le descripteur contient tous les renseignements nécessaires au générateur : type et localisation sous forme d'adresse segmentée, sinon le descripteur contient un pointeur vers la table ZMOCLA.
Le résultat de la codification est constitué d'une chaîne codée et des tables ZMOCLA et ZCO qui sont transmis au générateur.
On trouvera dans (MANSUY D., 74), la description détaillée du codifieur.

17.3. Le métaprocesseur.

C'est un métaprocesseur classique (MAC ILROY M.D., 60), (LEAVENWORK B.M., 66), (STRATCHEY C., 65).

Ses grandes lignes en ont été décrites dans (BENAMGHAR L., 73) pages 83 à 111. Il sera précisé et réalisé dans (HARMANT G., 74).

REFERENCES DU CHAPITRE 17

BENAMGHAR L., 73 Instruction d'affectation et définition d'un métalangage dans le projet CIVA. Doctorat d'Ingénieur. NANCY 1. Juin 73.

HARMANT G., 74 Métatraitement dans le projet CIVA. Thèse Université NANCY 1. A paraître.

MAC ILROY M. D., 60 Macro instruction extensions of compiler language. Comm. ACM 3 - 4 (April 60). 214.

LEAVENWORK B.M., 66 Syntax macros and extended translation. Comm ACM 9 - 11 (Nov. 66). 790.

MANSUY D., 74 Implantation des identificateurs et codification dans CIVA. NANCY 1. A paraître.

STRATCHEY C., 65 A general purpose macro generator. Computer Journal (oct. 65).

CHAPITRE 18

TRADUCTION DES TABLES DE DECISION

CHAPITRE 18

TRADUCTION DES TABLES DE DECISION

18.1 INTRODUCTION.

Une table de décision est une description, en mode déclaratif, des conditions régissant des actions à entreprendre. Les tables admises en Civa ont été décrites au chapitre 5. La traduction de ces tables pose un certain nombre de problèmes que nous évoquerons rapidement, et, principalement, un problème de stratégie. Etant un "énoncé", une table de décision laisse beaucoup de liberté au traducteur qui doit déterminer dans quel ordre évaluer les conditions et éventuellement dans quel ordre exécuter les actions. Les choix devant être opérés dépendent de l'objectif assigné au traducteur : nous avons retenu deux objectifs possibles, l'objectif "idéal" étant une fusion des deux :

- minimiser la place occupée par la traduction obtenue, ce que nous chercherons à réaliser en minimisant le nombre total des tests écrits dans la traduction
- minimiser le temps d'évaluation d'une table, ce que nous chercherons de deux manières : en minimisant le nombre de tests nécessaires pour reconnaître chaque règle ou en minimisant le temps moyen d'évaluation de la table en tenant compte des probabilités de réalisation des diverses conditions.

Plusieurs algorithmes ont déjà été décrits pour traduire les tables de décision. On trouvera un certain nombre de références à la fin de ce chapitre. Ils n'acceptent en général que des tables non ambiguës et à entrées limitées. Ils peuvent être regroupés en deux catégories, selon qu'ils relèvent de la méthode dite des "masques" qui analyse la table règle par règle ou d'une méthode d'analyse globale de la table.

Ce chapitre reprend le travail réalisé par Aubry B. dans le cadre de ce projet (Aubry B., 73). Un certain nombre de modifications ont été apportées, c'est cette version qui est réalisée.

18.2 METHODE DES "MASQUES". (KING P. J. H., 66), (DATHE G., 72).

Toutes les expressions booléennes décrivant les conditions sont d'abord évaluées. Les valeurs obtenues, rangées dans l'ordre de présence des conditions dans la souche constituent un vecteur booléen appelé "masque". Ce vecteur est alors comparé successivement à chaque vecteur constitué par les entrées de condition d'une règle jusqu'à ce qu'il y ait égalité - on exécute alors l'ensemble d'actionsde la règle trouvée et on poursuit les comparaisons si l'on accepte les ambiguïtés et, sinon, on s'arrête - ou alors qu'on ait épuisé toutes les règles.
Cette méthode a l'avantage essentiel d'être simple à mettre en oeuvre. La place occupée par le texte produit est minimale, en ce sens que le nombre de tests écrits est minimal : un par ligne de condition de la table. Le temps d'évaluation par contre est prohibitif dès que la table contient des entrées de condition indifférentes : en effet, quelles que soient leurs réalisations, les conditions sont toujours toutes évaluées, alors que certaines d'entre elles sont inutiles.
C'est néanmoins cette méthode qui est adoptée dans le projet, dans le cas de la traduction des tables de décision d'un module de commande. Ces tables sont en fait interprétées. Le temps d'exécution de ces tables n'est pas minimal , mais leur traduction est plus rapide que par une méthode qui fournirait une traduction meilleure. Avantage et inconvénient s'équilibrent quant au temps d'évaluation lorsque la table n'est exécutée qu'une fois. C'est la méthode la plus facile à mettre en oeuvre, c'est cette dernière raison qui l'emporte.
Notons cependant que, dans le cas d'une génération, il peut être rentable d'abaisser la durée moyenne d'exécution de la table en modifiant l'ordre dans lequel les règles sont comparées au masque et en commençant par comparer les règles les plus fréquemment réalisées. Cela suppose qu'à la traduction de la table les probabilités de réalisation de chacune des règles soient connues. On peut, par exemple, effectuer une première traduction par une méthode des masques simple et exécuter un certain nombre de fois la chaîne de traitement obtenue, en mesurant les fréquences de réalisation des règles. Ces données sont alors utilisées dans une modification ultérieure du texte objet pour réordonner les règles.

18.3 METHODES D'ANALYSE GLOBALE.

Il s'agit d'une famille de méthodes souvent décrites mais qui ne diffèrent que très peu les unes des autres. Nous décrirons donc un algorithme général en indiquant les points pour lesquels différents choix sont possibles.

18.31 Algorithme général.

Il consiste à choisir une condition dans la table. Il est alors possible de diviser la table en deux sous-tables dans lesquelles cette condition ne figure plus. L'une de ces tables est obtenue, à partir de la table d'origine, par le regroupement des règles dont l'entrée de la condition choisie est F ou '-', l'autre par le regroupement des règles dont l'entrée de cette condition est V ou '-'.

Ce procédé est répété pour toutes les sous-tables successivement obtenues et tant qu'il existe des condition à tester.

Ce qu'on peut décrire ainsi :

```
        stock = table à traduire ;
tant que stock ≠ ∅ faire
        choisir 1 (table) ; co  choix d'une table dans le stock oc ;
        si toutes conditions testées alors placer dans le texte
        (actions) sinon
        choisir 2 (condition) ; placer dans le texte (condition) ;
        diviser et ranger (table) ; co la table est divisée en
        deux sous-tables qui sont rangées dans stock, dans un
        ordre à définir oc ;
        fsi    fp ;
```

Les stratégies peuvent donc varier par :

- le critère de choix de la table à extraire du stock : "premier arrivé premier servi" ou "dernier arrivé premier servi" ce qui impose le type d'organisation du stock : en file d'attente ou en pile ; en fait, ce choix n'influe pas sur le texte obtenu mais seulement sur la programmation du traducteur. Nous choisirons une pile

- par le choix de la condition à tester ;
- par l'ordre de rangement des deux sous-tables obtenues à chaque itération.

Ce choix n'a pratiquement pas d'importance. Nous avons choisi de stocker d'abord la table correspondant aux entrées V et '-' et ensuite la table correspondant aux entrées F et '-'.

18.32 Construction du texte objet.

Le texte objet est construit progressivement par la procédure "placer dans le texte (actions ou conditions)" appelée à chaque itération. Pour illustrer cette construction, représentons le texte objet par un organigramme et montrons comment celui -ci est élaboré.

Dans cet exemple, le critère de choix de la table à extraire du stock est "dernier arrivé, premier servi" et le stock est organisé en pile ; le critère de choix de la condition à tester est "la première rencontrée".

Soit à traduire la table 1 :

Décision				
C1	V	V	F	F
C2	V	F	V	F
Action	A1	A2	A3	A4

Les tables de décision ou de sélection peuvent toujours se ramener à cette présentation en désignant par Ai l'ensemble d'actions de la règle Ri.

Phases de la traduction

stock ← table 1

1ère itération :

choisir 1 (table) : c'est nécessairement table 1 ;

choisir 2 (cond) : on choisit C1

placer dans le texte (cond) ;

diviser et ranger (table 1) ;

on obtient la table 2 pour les entrées V et '-' et la table 3 pour les entrées

Table 2 :

	R1	R2
C2	V	F
	A1	A2

Table 3 :

	R3	R4
C2	V	F
	A3	A4

Table 2 est d'abord rangée dans stock puis table 3.

2ème itération :

Choisir 1 (table) : table 3 est choisie ;

choisir 2 (cond) : C2 ;

placer dans le texte (cond) :

diviser et ranger (table 3) ;

on produit ainsi la table 4 (V) et la table 5 (F).

Création du texte

(organigramme)

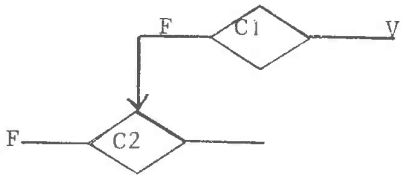
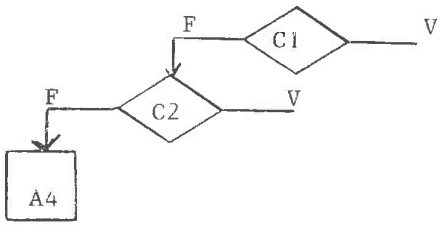


Table 4 : A3

table 5 : A4

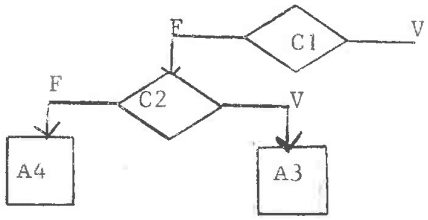
3ème itération :

choisir 1 (table) : table 5 ;
placer dans le texte (A4) ;



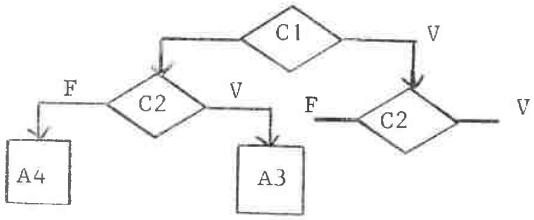
4ème itération :

choisir (table) : table 4 ;
placer dans le texte (A3)



5ème itération :

choisir 1 (table) : table 2 ;
choisir 2 (cond) : C2 ;
placer dans le texte (C2)
diviser et ranger (table 2) ;



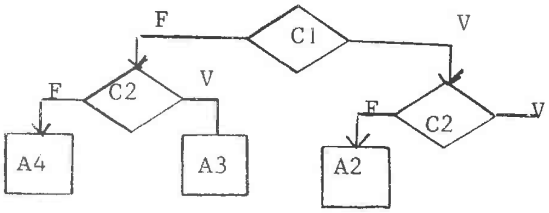
on obtient la table 6 pour V et la table 7 pour F.

Table 6 : A1

Table 7 : A2

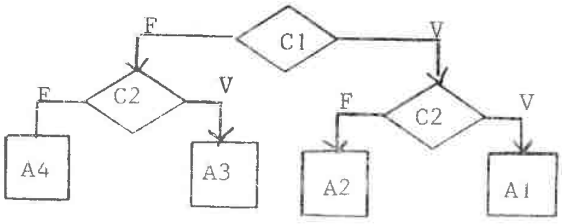
6ème itération :

choisir (table) : table 7 ;
placer dans le texte (A2)



7ème itération :

choisir (table) : table 6 ;
placer dans le texte (A1) ;
fin ;



18.33 Algorithme de cette famille minimisant le nombre moyen de tests.

Cet algorithme ne propose pas le texte le plus petit possible pour évaluer une table : c'est celui fourni par la méthode des masques. Parmi ceux pouvant être proposés dans cette famille d'algorithmes il ne propose aucun test inutile ; partant, le nombre de conditions rencontrées sur chaque branche de l'organigramme est minimum, la place occupée en mémoire par le texte objet est donc minimum elle aussi pour les algorithmes de cette famille.
On applique la règle suivante :

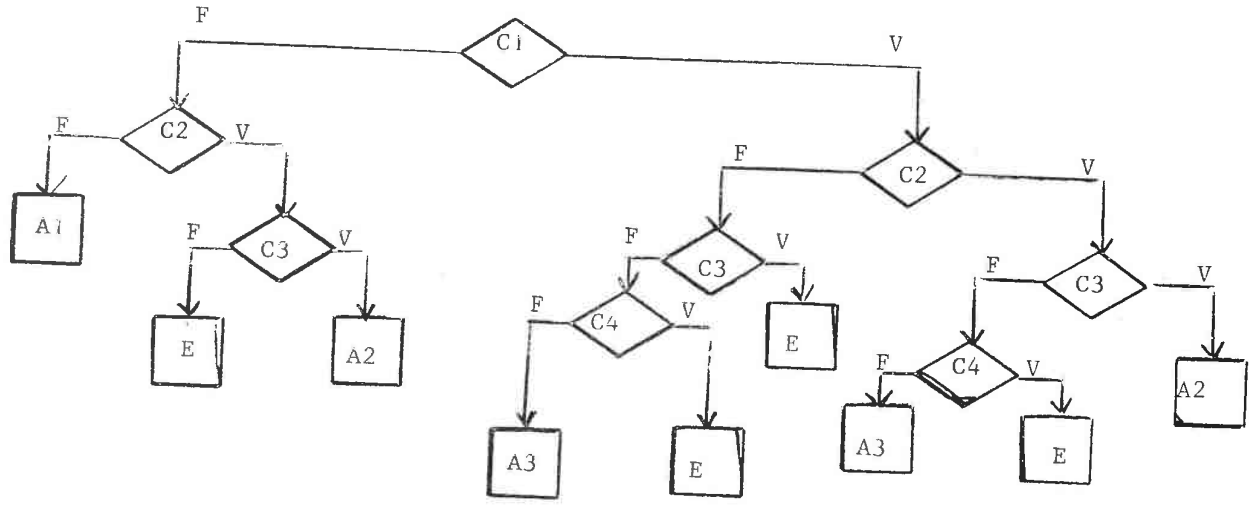
REGLE : choisir la condition qui a le moins d'entrées valant '-' ou '+ F' ou '+ V', pour décomposer une table en deux sous-tables.

En effet, si la table est complète et si toutes les entrées de condition sont significatives (c'est-à-dire si elles valent V ou F) pour sélectionner toute règle de la table, il faut autant de test qu'il y a de conditions. Les conditions indifférentes permettent de diminuer ce nombre de tests.

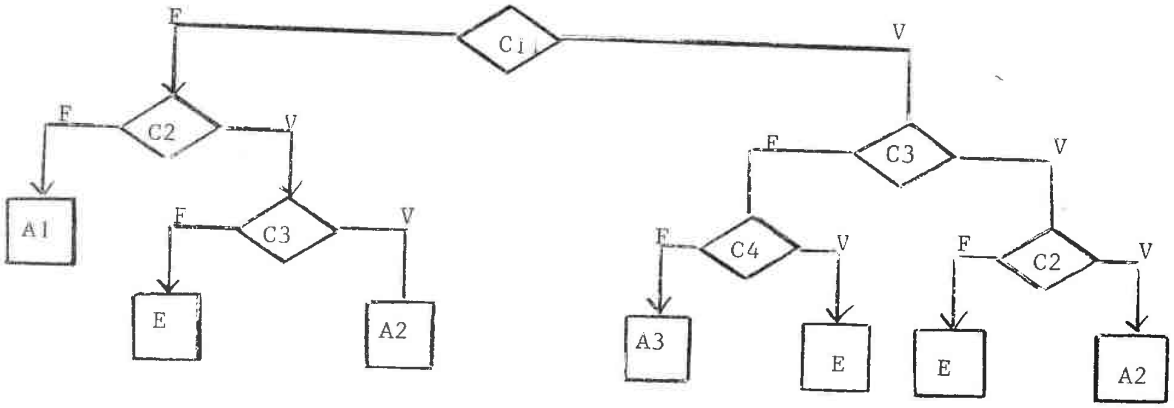
Exemple :
Soit la table 8

C1	F	-	V
C2	F	V	-
C3	-	V	F
C4	-	-	F
	A1	A2	A3

Un algorithme tel que celui utilisé en 18.32 (choisir la première condition) conduirait à l'organigramme suivant :



Alors que la règle ci-dessus conduit à la traduction :



18.34 Tables de décision ambiguës et tables de sélection.

18.341 Tables ambiguës.

Une table est ambiguë si deux règles ont même ensemble d'entrées de condition mais des ensembles d'entrées d'action distincts (cf. 5. 222)
La méthode des masques devient lourde à appliquer dans ce cas : pour s'assurer que toutes les règles ambiguës ont été vues, il faut poursuivre la comparaison du masque à toutes les règles dans tous les cas, alors que, si on n'acceptait pas de règles ambiguës, il suffirait de cesser les comparaisons dès que l'on a trouvé une concordance.

Dans une méthode d'analyse globale de la table, les règles ambiguës ne créent pas de difficultés supplémentaires. En effet, lors de la division d'une table en sous-tables,on regroupe toutes les règles ayant l'entrée de la condition choisie valant 'V' dans l'une d'elles, et celles valant 'F' dans l'autre : les règles ambiguës sont donc dans la même sous-table. Deux règles ambiguës correspondant aux actions A1 et A2 donnent donc naissance à la table restreinte à la suite A1; A2.

18.342 Tables de sélection.

La différence de valeur entre une table de décision et une table de sélection n'existe qu'en présence de règles ambiguës : dans une table de décision tous les ensembles d'actions correspondants sont exécutés alors que dans une table de sélection, seul l'ensemble d'action de la première règle (dans l'ordre d'écriture de la table) dont la valeur est vraie, est exécuté. Il est donc nécessaire de pouvoir déterminer quelle est cette première règle dans un ensemble de règles ambiguës.
Il suffit pour cela de construire les sous-tables d'une table en respectant l'ordre de présence des règles dans la table d'origine.
Dans les actions d'une table de sélection on peut rencontrer une instruction retable (I), qui demande une nouvelle évaluation

de la table à partir de la Ième règle. Ceci nous amènera, chaque fois qu'une branche de l'arborescence conduit à plusieurs actions, c'est-à-dire à chaque cas d'ambiguïté, à conserver le numéro des règles conduisant à cette ambiguïté : l'action entreprise est alors la première dont le numéro de règle est supérieur ou égal à I.

18.343 Entrées imposées.

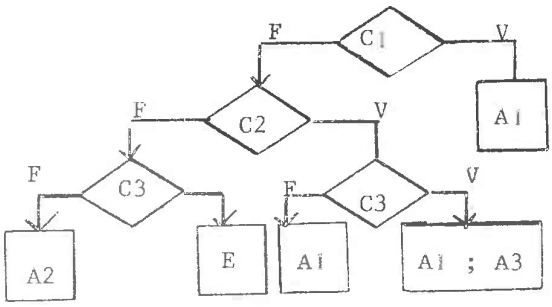
Une entrée imposée '+ F' ou '+ V' indique que, lorsque les entrées significatives de la règle sont vérifiées, celle-ci est inutile. Une entrée de ce type est donc considérée comme une entrée indifférente au moment du choix d'une condition pour diviser la table en deux sous-tables.
Par contre, lors de la division effective, si la condition choisie a une entrée + V ou + F, il sera inutile de faire figurer celle-ci dans les sous-tables obtenues.
Ainsi les tables 9 et 10 conduisent aux traductions suivantes :

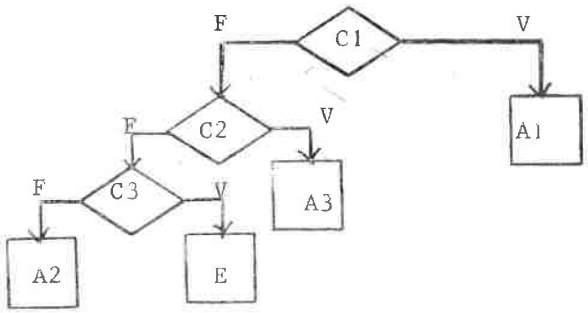
Décision	C1	V	F	F
	C2	-	-	V
	C3	-	F	-
Action		A1	A2	A3

Table 9

Décision	C1	V	F	F
	C2	-	+F	V
	C3	-	F	+ V
Action		A1	A2	A3

Table 10





18.35 Efficacité d'une traduction selon le nombre moyen de tests.

Soit une table de décision T à n conditions et q règles, et NT_p le nombre moyen de tests nécessaires à l'analyse complète de la table pour une traduction représentée par le programme P. (il s'agit de la moyenne du nombre de tests nécessaires pour déterminer quelle action entreprendre, rapporté au nombre d'actions indiquées : q + 1, sauf si la table est complète, cas où ce nombre vaut n).

On a donc immédiatement $\forall P \ (NT_p \leq n)$, l'égalité étant réalisée dans le cas d'une table complète dont toutes les entrées sont explicites (ou dans tous les cas par la méthode des masques).

Soit S_{ij} = 1 si l'entrée de la condition ECij vaut V ou F (cf. 5.41),

= 0 sinon.

Le nombre de tests pour reconnaître une réalisation comme étant celle de la règle Rj est borné inférieurement par :

$$NTM_j = \begin{cases} \sum_{i=1}^{n} S_{ij}, \text{ si la table n'a pas de règle E} \\ \sum_{i=1} S_{ij} + 1 \text{ si la table possède une règle E.} \end{cases}$$

$$\text{donc} \quad NTM = \frac{1}{q+k}\left(\sum_{\substack{i=1,\ n \\ j=1,\ q}} S_{ij} + kn \right)$$

où k vaut 1, s'il y a une règle E et 0 sinon.

Nous appellerons efficacité selon le nombre moyen de tests d'une traduction P le rapport $EFT = \frac{NTM}{NT_p}$.

Ainsi pour la table 8, l'efficacité selon le nombre moyen de tests de la traduction obtenue par l'algorithme de 18.32 peut être calculée par :

$$NTM = \frac{1}{4}(7 + 4) = 2,75$$

$$NT_p = \frac{1}{9}(2 + 3 + 3 + 4 + 4 + 3 + 4 + 4 + 3) = \frac{30}{9} = 3,33$$

Alors que pour l'algorithme de 18.33 elle est de

$$NT_p = \frac{1}{7}(2 + 3 + 3 + 3 + 3 + 3 + 3) = 2,857$$

$$\text{efficacité de 18.33} = \frac{2,75}{2,857} = 0,962$$

Cette définition de l'efficacité d'une traduction, tout comme l'algorithme de 18.33, favorise le gain du nombre de tests, mais en en fait ce n'est pas toujours le gain souhaitable. En effet, il vaut quelquefois mieux augmenter le nombre de tests sur une branche aboutissant à une règle qui est rarement réalisée, si cela permet de diminuer le nombre de tests rencontrés pour une règle fréquemment réalisée surtout si les tests sont longs à évaluer. D'où la nécessité d'introduire la fréquence de réalisation des règles, ainsi que le coût d'évaluation des conditions qui n'est pas toujours identique pour toutes les conditions.

18.36 Efficacité d'une traduction selon la durée moyenne d'évaluation.

18.361 Table sans règle E.

Soit t_i la durée d'estimation de la condition C_i, D_p la durée moyenne d'analyse d'une table donnée par un programme P, et f_j la fréquence relative de réalisation de la règle R_j.

La durée d'estimation d'une règle est minorée par :

$$DM_j = \sum_{i=1}^{n} S_{ij} \cdot t_i .$$

D'où une durée moyenne minimum pour l'évaluation de la table :

$$DM = \sum_{j=1}^{q} DM_j = \sum_{j=1}^{q} \sum_{i=1}^{n} S_{ij} . t_i \cdot fj$$

$$= \sum_{i=1}^{n} ti \cdot \sum_{j=1}^{q} S_{ij} . fj$$

Soit $P_i = \sum_{j=1}^{q} S_{ij} . fj$; P_i représente la probabilité minimale de test de la condition i.

$$DM = \sum_{i=1}^{n} P_i \; t_i$$

D'autre part, pour tout p ($DM \leq D_p \leq \sum ti$)

Nous définissons l'efficacité selon la durée d'évaluation, de la traduction P d'une table par

$$E_p = \frac{DM}{D_p}$$

Nous verrons au paragraphe suivant un exemple de calcul de cette efficacité.

18.362 Table avec règle E.

Si une table possède la règle E dont la probabilité de réalisation n'est pas négligeable, les évaluations précédentes sont insuffisantes.

La forme $DM = \sum_{i=1}^{n} Pi \; ti$ peut être conservée en considérant que chaque condition ayant des entrées indifférentes a une entrée significative dans E.

$$Pi = \sum_{j=1}^{q} Sij . fj + k'_i f_E \text{ avec } k' = 1 \text{ si Ci a des entrées indifférentes V ou F}$$

$$= 0 \text{ sinon}$$

On obtient alors une valeur approchée de DM.

18.37 Algorithme d'analyse globale minimisant la durée moyenne d'évaluation d'une table.

18.373 Algorithme.

Il différera de celui de 18.32 uniquement par le choix de la condition utilisée pour diviser la table en sous-tables. La première condition choisie sera la première du programme résultant : elle sera toujours testée, quelle que soit la réalisation : il faut donc choisir celle qui a la plus grande probabilité d'être testée mais qui coûte le moins cher. D'où la première règle :

REGLE 1 : Choisir la condition pour laquelle le quotient P_i / t_i est maximum. (C'est-à-dire celle dont le test est le plus probable si tous les coûts t_i sont égaux.)

Cette règle peut être insuffisante : deux conditions pouvant faire aboutir au même quotient.

REGLE 2 : En cas d'ambiguïté après application de la règle 1, choisir la condition dont le test est le plus efficace, c'est-à-dire celui pour lequel la probabilité P (Ci = vrai) est la plus proche de 1/2, parmi les conditions retenues par la règle 1.

18.372 Mise en oeuvre.

La mise en oeuvre de cet algorithme nécessite la connaissance des fréquences de réalisation des différentes règles de la table. Cet algorithme ne peut donc être envisagé que pour le cas de tables importantes et utilisées suffisamment longtemps. On peut alors procéder à une première traduction de la table par l'algorithme de 18.32 et faire exécuter le programme obtenu en mesurant les fréquences de réalisation des règles : il faut inclure des incrémentations de compteurs dans la traduction. Il convient de répéter cette exécution un nombre de fois suffisant pour que les fréquences obtenues puissent être considérées comme des probabilités de réalisation.

On refait alors une traduction de la table en suivant cet algorithme.

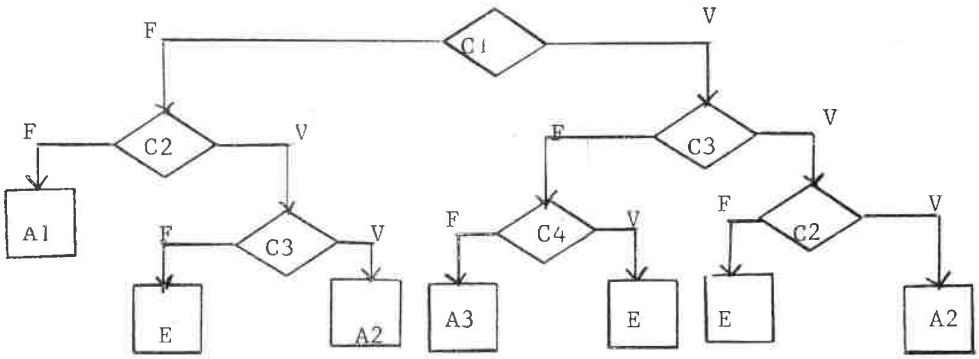
En fait les calculs de D_M et D_p devraient être faits en partant , non pas des probabilités de réalisation des règles, mais des probabilités de voir une condition vérifiée. Celles-ci sont plus délicates à obtenir au cours de la phase d'essai.

18.373 Efficacité de cet algorithme.

Soit la table 11

Condition				
	C1	F	-	V
	C2	F	V	-
	C3	-	V	F
	C4	-	-	F
		A1	A2	A3

L'application de l'algorithme du paragraphe 18.32 conduit à l'organigramme suivant



L'efficacité selon le nombre de tests de cette traduction est calculée par :

$$NTM = \frac{1}{4}(7 + 4) = \frac{11}{4}$$

$$NT_{P_1} = \frac{1}{7}(2 + 3 + 3 + 3 + 3 + 3 + 3) = \frac{20}{7}$$

$$EF\ T_1 = \frac{77}{80} = 0{,}9625$$

Supposons que les fréquences de réalisation des différentes règles soient les suivantes :

R1 : 1/8 ; R2 : 5/8 ; R3 : 1/8 ; E : 1/8 ;

et que les coûts d'évaluation des conditions soient :

C1 : 9 ; C2 : 1 ; C3 : 2 ; C4 : 1 ;

Calculons les $Pi = \sum_{j=1}^{3} S_{ij} \cdot fj + k'i \cdot f_E$

$P1 = \frac{3}{8}$; $P2 = \frac{7}{8}$; $P3 = \frac{7}{8}$; $P4 = \frac{2}{8}$;

$$DM = \sum_{i=1}^{n} Pi.ti = 3 \times 9 + 7 \times 1 + 7 \times 2 + 2 \times 1 = 50/8$$

$$DM = \frac{50}{8}$$

Pour calculer le temps moyen d'évaluation de la table par cette traduction, nous supposerons que la règle E peut être atteinte par l'une des trois branches possibles avec la même probabilité. De même, si une action figure plusieurs fois, nous supposerons que les probabilités sont uniformément réparties selon les différentes branches aboutissant à cette action. (La connaissance des probabilités de réalisation des conditions permettrait d'éviter cette hypothèse - mais elle n'est pas très importante).

$$D_{p1} = \frac{1}{8}\Big((1 \times (1+9) + \frac{1}{3}(2+1+9) + \frac{5}{2}(2+1+9) + 1(1+1+9)$$

$$+ \frac{1}{3}(1+1+9) + \frac{1}{3}(1+2+9) + \frac{5}{2}(1+2+9)\Big) = 92,66/8.$$

D'où une efficacité selon la durée d'évaluation pour cette traduction $\boxed{E_{p1} = \frac{50}{92,66} = \underline{0,539}}$

Appliquons maintenant l'algorithme de I8.37I. Il vient :

Ière itération :

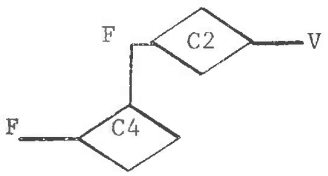
$\left[Pi/Ci\right] = \frac{3}{8x9}, \frac{7}{8xi}, \frac{7}{8x2}, \frac{2}{8xi}$; la condition choisie est C2.

La sous-table "entrée de C2 = F" est

Ci	F	V
C3	-	F
C4	-	F
	A1	A3

dans laquelle $\left[Pi\right] = \frac{1}{8}\left[2, 2, 2\right]$ et $\left[Pi/CI\right] = \frac{1}{8}\left[\frac{2}{9}, 1, 2\right]$

La condition C4 est donc choisie :

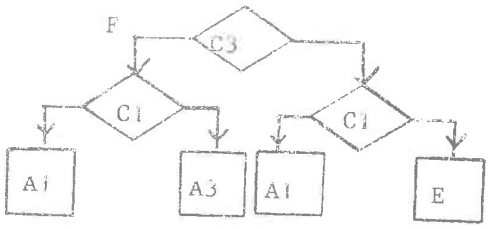


Prenons la sous-table "entrée de C4 = F" :

$$\left[Pi\right] = \frac{1}{8}\left[2,\ 2\right] \ ; \ \left[Pi/Ci\right] : \frac{1}{8}\begin{bmatrix} \frac{2}{9}, & 1 \end{bmatrix}$$

C1	F	V
C3	-	F
	A1	A3

C'est donc C3 qui est choisie. On obtient

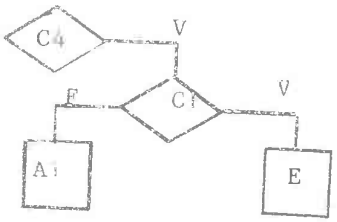


Reprenons la table "entrée de C4 = V" :

$$\left[Pi\right] = \frac{1}{8}\left[1,\ 0\right] \ ; \ \left[Pi/Ci\right] = \frac{1}{8}\begin{bmatrix} \frac{1}{9}, & 0 \end{bmatrix}$$

C1	F
C3	-
	A1

C1 est choisie ce qui conduit à



Prenons la table "entrée de C2 = V"

$$\left[Pi\right] = \frac{1}{8}\left[2,\ 6,\ 2\right] \ ; \ \left[Pi/Ci\right] = \frac{1}{8}\begin{bmatrix}\frac{2}{9}, & 3, & 2\end{bmatrix}$$

C1	-	V
C3	V	F
C4	-	F
	A2	A3

C3 est donc choisie

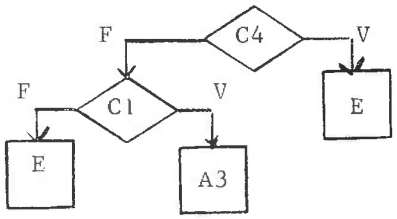
Table "entrée de C3 = F"

$$\left[Pi\right] = \frac{1}{8}\left[1,\ 1\right] \ ; \ \left[Pi/Ci\right] = \frac{1}{8}\begin{bmatrix}\frac{1}{9}, & 1\end{bmatrix}$$

C1	V
C4	F
	A3

C4 est choisie.

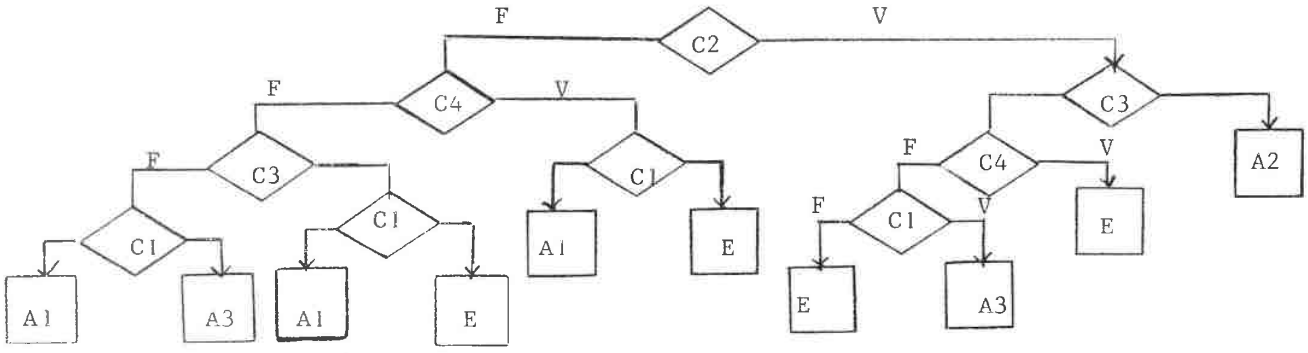
On obtient



La sous-table "entrée de C3 = V" se réduit à :

C1	-
C2	-
	A2

Traduction P2 obtenue :



En prenant les mêmes conventions que pour la traduction P1, on a :

$$D_{P_2} = \frac{1}{8}\left(\frac{1}{3}(9+2+1+1) + \frac{1}{2}(9+2+1+1) + \frac{1}{3}(9+2+1+1)\right.$$

$$+ \frac{1}{4}(9+2+1+1) + \frac{1}{3}(9+1+1) + \frac{1}{4}(9+1+1)$$

$$\left. + \frac{1}{4}(9+1+2+1) + \frac{1}{2}(9+1+2+1) + \frac{1}{4}(1+2+1) + 5\,(2+1)\right)$$

$$= 52{,}75 \,/\, 8$$

$$E_{p2} = \frac{50}{52{,}75} = 0{,}948 \text{ au lieu de } 0{,}539$$

L'efficacité selon la durée a donc considérablement augmentée, au détriment, bien sûr de l'efficacité selon le nombre de tests qui vaut maintenant :

$$NT_{p2} = \frac{1}{10}(4+4+4+4+3+3+4+4+3+2) = 3{,}5$$

$$EFT_2 = \frac{11}{14} = \underline{0{,}785} \text{ au lieu de } 0{,}9625$$

18.374 Conclusion.

Les tables de décision ou de sélection d'un module de commande sont traduits par la méthode des masques.
Les autres tables de décision ou de sélection sont traduites par l'algorithme de 18.2, algorithme d'analyse globale minimisant le nombre moyen de tests nécessaires à l'évaluation de la table.
Pour les tables dont l'utilisation est particulièrement courante, qui contiennent un nombre non négligeable d'entrées indifférentes et telles que les coûts d'évaluation des conditions soient distincts, on peut penser à utiliser l'algorithme de 18.371, minimisant la durée moyenne d'évaluation, après avoir mesuré les fréquences de réalisation des règles à l'aide d'une autre traduction de la table.

18.4 REALISATION.

Nous passerons rapidement sur les détails de la réalisation proprement dite. Elle est décrite dans un document technique (Fiegel C. 73) Disons simplement qu'une table peut être transformée par le codifieur. Celui-ci codifie chacun des constituants de la table, mais en outre il peut être amené à déduire de la table source, une autre table pour exprimer les simplifications d'écriture du paragraphe.
Le générateur traduit d'abord toutes les souches de conditions sous la forme de fonctions et toutes les souches d'action sous la forme de sous-programmes. Puis il génère un sous-programme (appelant les précédents) pour chaque ensemble d'actions.

Des études précises ont été faites pour chiffrer le coût d'autres solutions possibles, elles ont montré l'intérêt de celle-ci.
Une table de décision ou de sélection de dimension 2, est considérée comme le produit de deux tables.

REFERENCES DU CHAPITRE 18.

AUBRY B., 73 — Traduction des tables de décision. Thèse 3ème cycle - Université Nancy I - mars 1973

CHAPIN N., 67 — Parsing of decision tables. CACM. Vol. 10, n° 8 (1967) p. 680 - 684.

DATHE G., 72 — Conversion of decision tables by rule Mask methods without rule mask. CACM Vol 15, n° 10 (1972), p. 906-909.

FIEGEL C., 73 — Réalisation du traducteur de tables de décision dans le projet Civa..Note technique. Nancy (Oct. 1973).

KING P. J. H., 66 — Conversion of decision table to computer programms by rule mask technique CACM vol. 9, n° 11 (1966), p. 796-801.

MUTHUKRISHNAN C. R., 70 — Muthukrishnan C. R. and Rajamaran V. On the conversion of decision tables to computer programs. CACM. vol. 13, n° 6 (1970), p. 347-351.

POLLACK S. L., 65 — Conversion of limited entry decision tables. CACM. vol. 8, n° 11 (1965), p. 677 - 682.

PRESS L. I., 65 — Conversion of decision tables to computer programs. CACM. vol. 8 n° 6 (1965), p. 385-390.

SHWAYDER K., 71 — Conversion of limited entry decision tables to computer programs. A proposed modification to Pollack's algorithm. CACM. , vol. 14, n° 2 (1971), p. 69 - 73.

CHAPITRE 19

TRADUCTION DES EXPRESSIONS ARITHMETIQUES ET DES CONTROLES

19.1 TRADUCTION DES EXPRESSIONS ARITHMETIQUES.

19.2 IMPLANTATION DES CONTROLES.

19.21 Mecanismes de base.

19.22 Changements d'état d'une variable contrôlable.

1 Utilisation de BCV2 pour "reprendre".

2 Initialisation des BCV.

3 Instruction "quand" et libération explicite.

4 Instruction "suspendre" et "reprendre".

5 Libérations implicites.

6 Présence d'une instruction "sortir" dans une séquence de récupération.

19.23 CONCLUSION.

CHAPITRE 19

TRADUCTION DES EXPRESSIONS ARITHMETIQUES ET DES CONTROLES

Ce chapitre est particulièrement court : la traduction des expressions arithmétiques est un problème classique et les solutions en sont bien connues. Seule la traduction des contrôles, plus exactement l'ensemble des modifications devant être apportées à la traduction des instructions d'affectation pour tenir compte des contrôles, soulève quelques problèmes.

19.1 TRADUCTION DES EXPRESSIONS ARITHMETIQUES.

Le codifieur a transformé le texte source en une chaîne codée dans laquelle, chaque opérateur d'une expression arithmétique est codé et chaque opérande est remplacé par son adresse. Partant de cette chaîne, le module de traduction des expressions arithmétiques procède en deux étapes :

- analyse syntaxique de la chaîne codée et production d'un texte représentant la suite d'opérations élémentaires à effectuer, sous forme d'un code à 3 adresses, cette étape intermédiaire étant prévue pour permettre une optimisation du code généré. Le module correspondant utilise un algorithme classique d'analyse d'expression arithmétique parenthésée avec priorité des opérateurs, à l'aide de deux piles : pile opérateur et pile opérande. Dans le texte produit, les mémoires de travail nécessaires sont désignées par des numéros, aucune réservation n'étant faite à ce niveau.
- génération, qui fait passer du texte intermédiaire à la traduction de l'expression arithmétique en langage objet. C'est au cours de la génération que les noms de mémoires de travail sont associés à des emplacements.

Deux versions de la génération sont utilisées selon que la compilation se fait en mode mise au point (génération simple) ou en mode d'exploitation (génération avec optimisation de l'utilisation des registres).
Le module de génération simple gère l'espace de travail nécessaire aux calculs intermédiaires à l'aide d'une pile.
Le module de génération avec optimisation recherche l'existence de sous-expressions identiques dans une section de programme composée exclusivement d'instructions d'affectations. Il est donc amené à modifier le texte intermédiaire. La chaîne de codes à 3 adresses obtenue est ensuite divisée en sous-chaînes décrivant des opérations portant sur des opérandes d'un même type.
Pour chaque sous-chaîne, l'implantation des variables de travail est réalisée en utilisant au maximum les registres. On utilise pour cela l'algorithme d'allocation de registres de Sethi et Ullman (Louit G., 72).
Le code objet est alors généré.
Les modules d'analyse et de génération simple sont décrits, en utilisant le langage Civa, dans (Ducloy J., 73).

19.2 IMPLANTATION DES CONTROLES.

19.21 Mécanisme de base (Ducloy J., 73).

Une affectation à une variable contrôlable peut nécessiter l'exécution d'une action particulière : l'exécution de la séquence de récupération de cette affectation consitute un évènement de la classe définie par l'instruction "quand".

La détection de ces évènements est réalisée la plupart du temps par programme. Or une variable contrôlable n'est pas toujours contrôlée et, quand elle l'est, une instruction d'affectation à cette variable ne constitue pas toujours un évènement de la classe cherchée : il est donc extrêmement important que la solution adoptée pour l'implantation des contrôles n'augmente que très peu le temps d'exécution d'une instruction d'affectation, lorsque la séquence de récupération ne doit pas être entreprise, et en particulier, qu'elle ne pénalise pas une unité de traitement possédant des variables contrôlables pendant les périodes où celles-ci ne sont pas à l'état contrôlé.

Pour cela, à chaque variable contrôlable est associé un bloc de contrôle de variable (B C V) formé de 3 mots. Le premier mot de ce bloc contient une instruction. Cette instruction est exécutée après chaque affectation à la variable contrôlable i, par l'intermédiaire d'une instruction :

EXU BCVI (i)

Si la variable contrôlable i est à l'état "libéré" (cf. 6.66), BCVI (i) contient une instruction inopérante NØP (LCFI, o o car elle est la plus rapide).

La déclaration de i contrôlable augmente donc le temps d'exécution de l'affectation à i de 2, 5 μs ce qui est assez faible.

Si i est à l'état "contrôlé" BCVI (i) contient alors une instruction de branchement avec sauvegarde du compteur ordinal (appel de sous-programme). L'adresse sauvegardée est celle de l'instruction qui suit EXU.

BCVI (i) contient alors :

BAL, RLC SSP contrôle i

où RLC désigne le registre de liaison utilisé pour les contrôles et SSP contrôle i, le sous-programme de contrôle de la variable i.

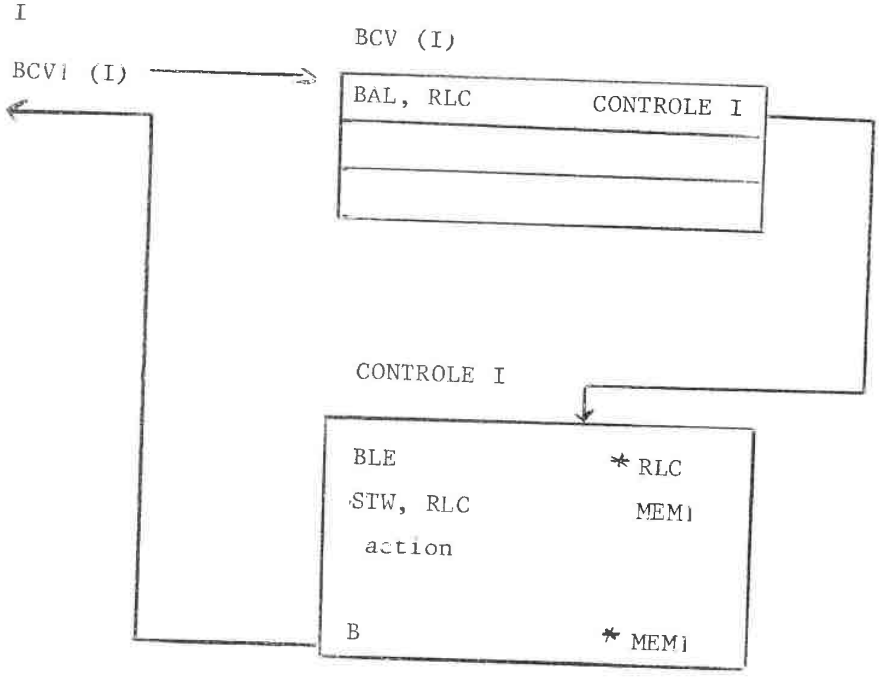
Remarquons que, si la classe d'évènements déclarée est de la forme i op 0 où op est un opérateur de comparaison, la première instruction du sous-programme "ssp contrôle i" est un branchement selon le code de condition (qui n'est pas modifié par Bal, RLC).

Exemple :

quand i > 0 faire action fqd ;

L'instruction "i = 3" se traduit par

```
LI, R    3
STW, R   I
EXU      BCVI (I)
```



tandis que pour le contrôle :

quand j > t faire actionj fqd;

Le sous-programme de contrôle commence par :

```
CONTROLE J    CW, R    T
              BLE      * RLC
              action j
```

19.22 Changements d'état d'une variable contrôlable.

19.221 Utilisation de BCV2 pour "reprendre".

Lorsqu'une variable contrôlable est à l'état "suspendu", l'action à entreprendre lors d'une affectation à cette variable est vide : BCVI doit donc contenir NØP.
Lors d'une reprise, BCVI doit retrouver la valeur qu'il contenait avant la suspension : BCV2 sera utilisée pour conserver le contenu antérieur de BCVI tant que la variable est à l'état "suspendu".
Une instruction "suspendre" et une instruction "reprendre" étant

exécutées implicitement au début et à la fin de la séquence de récupération, ces instructions doivent être rapides.
Pour que l'exécution de "reprendre" soit la plus rapide possible, il faut qu'elle soit indépendante de l'état dans lequel se trouve la variable contrôlée.
Pour cela, BCV2 doit contenir NØP et l'instruction à exécuter si l'état est contrôlé ou suspendu.

19.222 Initialisation des BCV.

A chaque appel d'un module déclarant une variable contrôlable, celle-ci se trouve à l'état "libéré" jusqu'à la rencontre d'un "quand". Le premier et le second mot du BCV doivent donc être initialisés à la valeur NØP (c'est-à-dire LCFI, 0 0).
Cette initialisation se fait au début de l'exécution d'un module pour les variables contrôlables d'un module et la première fois qu'on utilise une classe pour les variables contrôlables de celle-ci.

19.223 Instruction "quand" et libération explicite.

La traduction d'une instruction "quand" consiste alors en la construction de la séquence de récupération sous forme de sous-programme comme il a été vu en 19.21 et au rangement de l'instruction

BAL, RLC contrôle (I) dans BCV1 et dans BCV2.

On verra en 9.225, qu'une instruction "quand" a également pour effet de ranger le compteur NIV dans BCV3.
Pour l'instruction explicite "libérer" il suffit de ranger NOP dans BCV1 et dans BCV2, et 0 dans BCV3, comme nous le verrons en 19.225.

19.224 Instructions "suspendre" et "reprendre".

"Reprendre" peut alors être toujours réalisé par un simple transfert de BCV2 dabs BCV1.

"suspendre" consiste en la mise de NOP dans BCV1.

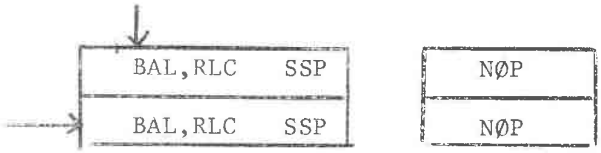
Exemple :

```
module m ;
    i, j entier contrôlable ;
```

BCV (i) BCV (j)

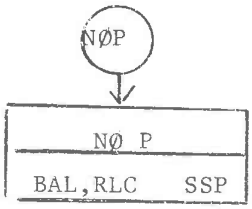
initialisation implicite

```
quand  i faire
   "suspendre"  j ;
j = 3 ; i = 1 ;
       reprendre j ;
       fqd
```



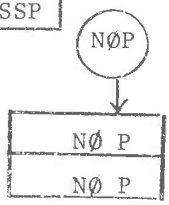
i passe à l'état "contrôlé".

i = 3 ; on exécute : suspendre i ; (implicite)

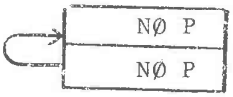


suspendre j

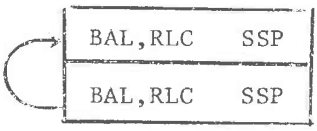
j = 3 ; i = 1 ;



reprendre j ;



reprendre i (implicite)



19.225 Libérations implicites.

Nous avons vu en 6.65, que la durée d'un contrôle est limitée à la durée de vie du module dans lequel est exécuté l'instruction "quand" définissant ce contrôle. A la fin d'exécution de ce module, la variable contrôlable, peut encore être valide (si elle est déclarée dans une classe) il convient alors de la libérer.

Cependant, une instruction "quand" peut être écrite dans une procédure, déclarée dans une classe utilisée par le module m et cette procédure peut être appelée conditionnellement ; il est alors impossible de retrouver à la compilation de m, toutes les instructions "quand" susceptibles d'être exécutées dans le module m.

Pour chaque module, le compilateur génère une table (TBCVM) des BCV pour lesquels il faudra peut être faire une libération implicite à la fin de l'exécution de ce module. Le troisième mot des BCV est utilisé pour déterminer quels sont les contrôles à libérer.

Considérons le graphe (E, A) des appels de modules d'une unité de traitement. L'exécution de cette unité, correspond à un cheminement dans ce graphe. Nous appellerons niveau d'un appel de module m, la longueur du chemin de m_0, module directeur de l'unité à m, qui a été utilisé pour aboutir à cet appel.

A l'exécution d'une unité de traitement, un compteur NIV est entretenu, qui contient le niveau du module en cours d'exécution : il suffit d'augmenter NIV de 1 à chaque appel de module et de le diminuer de 1 à chaque retour. NIV appartient à la classe système, et il est initialisé à 0, au début du module de commande.

Une instruction "quand" aura pour effet secondaire, de ranger NIV dans BCV3.

A la fin d'exécution du module m, avec le niveau n, toutes les variables contrôlées dont le BCV figure dans TBCVM (m) est pour lesquelles BCV3 vaut n sont libérées :

NØP est rangé dans BCV1 et dans BCV2,

0 dans BCV3.

Le premier mot d'une table TBCVM contient le niveau du module auquel est attaché cette table.

19.226 Présence d'une instruction "sortir" dans une séquence de récupération.

On a vu en 6.675 , que la présence d'une instruction "sortir" dans la séquence de récupération d'un contrôle demande la fin d'exécution du module dans lequel est défini ce module (exécution de "quand").
Il est alors nécessaire de libérer les contrôles définis dans ce module, mais aussi ceux définis dans les modules que celui-ci a appelés.
Pour cela, on entretiendra une pile à l'exécution contenant des adresses de tables TBCVM : tout module contenant une TBCVM, empile en début de traitement, l'adresse de celle-ci et la retire lors de sa fin d'exécution. Pour exécuter sortir on explore alors cette pile pour n'en retenir que les tables TBCVM ayant un niveau supérieur ou égal à celui du module exécutant "quand" (contenu de NIV). On libère alors les variables contrôlables dont les BCV sont dans ces tables et qui doivent l'être selon la règle vue au paragraphe précédent.

19.23 CONCLUSION.

L'implantation des contrôles peut paraître lourde pour ce qui concerne le problème des libérations implicites. Le temps d'exécution supplémentaire qui en découle est compensé par la facilité d'expression des contrôles ainsi définis, qui deviendraient assez vite difficile à manier si toutes les libérations devaient être explicites.
Remarquons enfin que l'introduction des contrôles dans le langage ne pénalise pratiquement pas une unité de traitement qui ne les utilise pas : chaque module doit comporter une instruction d'augmentation de niveau, au début de son exécution : MTW, 1 NIV
et une instruction de diminution de ce niveau en fin d'exécution : MTW, -1 NIV, ce qui est très facile (4,5 μ1).

REFERENCES DU CHAPITRE 19

DUCLOY J, 73 Compilation dans le projet Civa
Doctorat d'Ingénieur - Nancy 1 - Mars 1973.

LOUIT G., 72 Optimisation des expressions arithmétiques.
Extension des algorithmes de Sethi R. et Ullman J. D.
Congrès de l'A. F. C. E. T., Grenoble 1972.

CHAPITRES 20 - 21 - 22 - 23

---

20. INTERPRETATION DES MODULES DE COMMANDES.

21. IMPLANTATION DES FILES ET DES ENSEMBLES.

22. ENTREES SORTIES IMPLICITES ET EXPLICITES.

23. TRADUCTION DES INSTRUCTIONS D'ITERATION.

Ces quatre chapitres recouvrent des domaines de la réalisation qui sont encore en cours.

Le chapitre 20 indique les solutions utilisées pour résoudre les problèmes posés par les modules de commande et les possibilités qu'ils offrent (cf. chapitre 10). Ces problèmes sont essentiellement :

- gestion de l'état des différentes unités et de leurs transitions ;
- réalisation des interfaces entre les textes CIVA et le système d'exploitation utilisé (SIRIS 7), en particulier en ce qui concerne l'occupation de la mémoire et la gestion des fichiers ;
- gestion des différentes bibliothèques.

Ces problèmes seront décrits dans (VIAULT D., 74).

Les chapitres 21, 22, 23 concernent tous les trois les files d'ensemble et les opérations qu'on peut effectuer sur eux. FIEGEL C. s'est chargé de la réalisation de ces problèmes. On trouvera dans (FIEGEL C., 74) la description des solutions adoptées pour les résoudre.

REFERENCES DES CHAPITRES 20 - 21 - 22 - 23

FIEGEL C. 74 Traduction des instructions d'itération dans le projet CIVA et application à COBOL. Thèse 3ème cycle. NANCY 1. A paraître. 4ème trimestre 74.

VIAULT D. 74 Définition et interprétation des modules de commande dans le projet CIVA. Thèse 3ème cycle. NANCY 1. A paraître. 4ème trimsetre 74.

CHAPITRE 24

EVALUATION DES INSTRUCTION D'ITERATION LOGIQUE

CHAPITRE 24

EVALUATION DES INSTRUCTIONS D'ITERATION LOGIQUE

24.1. INTRODUCTION.

Nous avons vu au chapitre 8 la définition des instructions d'itération logique simple ou multiple. Elles ont été définies en donnant leur équivalent en CIVA.

Or, les notions introduites sont indépendantes du langage utilisé. Elles pourraient donc être employées avec profit pour construire des programmes dans d'autres langages : les instructions d'itération logique, les déclarations d'indicatif servent alors d'entrée à un générateur de programme qui produit les lignes correspondant à la cinématique automatique des fichiers. On verra la génération d'un tel programme en Cobol dans (HERTSCHUH N., 74).

Dans ce chapitre, nous donnons d'abord la description de l'exécution d'un ensemble d'instructions logiques simples et imbriquées. Puis nous traitons le cas général des instructions d'itération logique multiple imbriquées, lorsque tous les fichiers concernés n'ont pas le même indicatif mineur.

24.2. EVALUATION D'INSTRUCTIONS D'ITERATION LOGIQUE SIMPLE IMBRIQUEES.

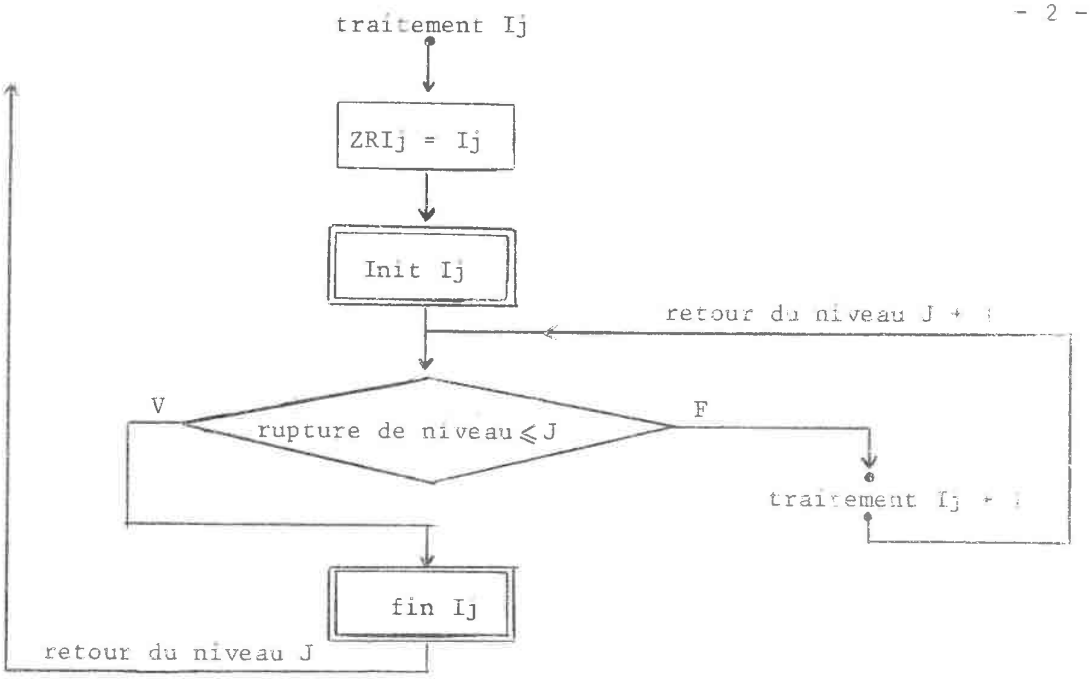
Soit un groupe d'instructions imbriquées, attachées, dans cet ordre aux indicatifs I_1, I_2, ... I_n d'un même ensemble.

A chaque indicatif I_j, associons une variable de travail ZRI_j locale au traitement de l'instruction d'indicatif I_j et de même type que I_j : on aurait donc ZRI_j type I_j (cf. chapitre 4).

24.21. Effet d'une instruction attachée à un indicatif central.

L'effet de l'instruction d'itération logique simple d'indicatif I_j ($1 < j < n$) peut être décrit comme suit :

Les traitements encadrés doublement sont à décrire par l'utilisateur.



Init Ij : c'est la suite d'instructions qui séparent le mot "faire" de l'instruction d'indicatif Ij + 1. Init Ij peut être vide.

Traitement Ij + 1 :

là sera placée la description du traitement de l'instruction d'itération attachée à l'indicatif Ij + 1.

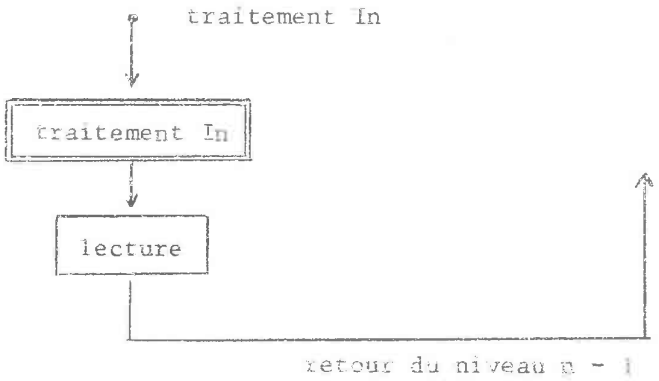
Rupture Ij :

il y a rutpure pour un niveau k, $1 \leqslant k \leqslant j$, si $Ik \neq ZRIk$.

Fin Ij : c'est la suite d'instructions qui séparent le mot "fpc" de l'instruction correspondant à l'indicatif Ij + 1 du mot "fpc" de l'instruction d'indicatif Ij.

24.22. Effet de l'instruction attachée à l'indicatif mineur.

L'instruction d'itération logique simple d'indicatif In a un effet plus simple :



traitement In : il s'agit là des instructions écrites entre faire et fpc dans l'instruction d'indicatif In. Rappelons que l'indicatif In est associé à un élément courant de la file support des sous-ensembles logiques d'indicatif I_1 à In - 1.

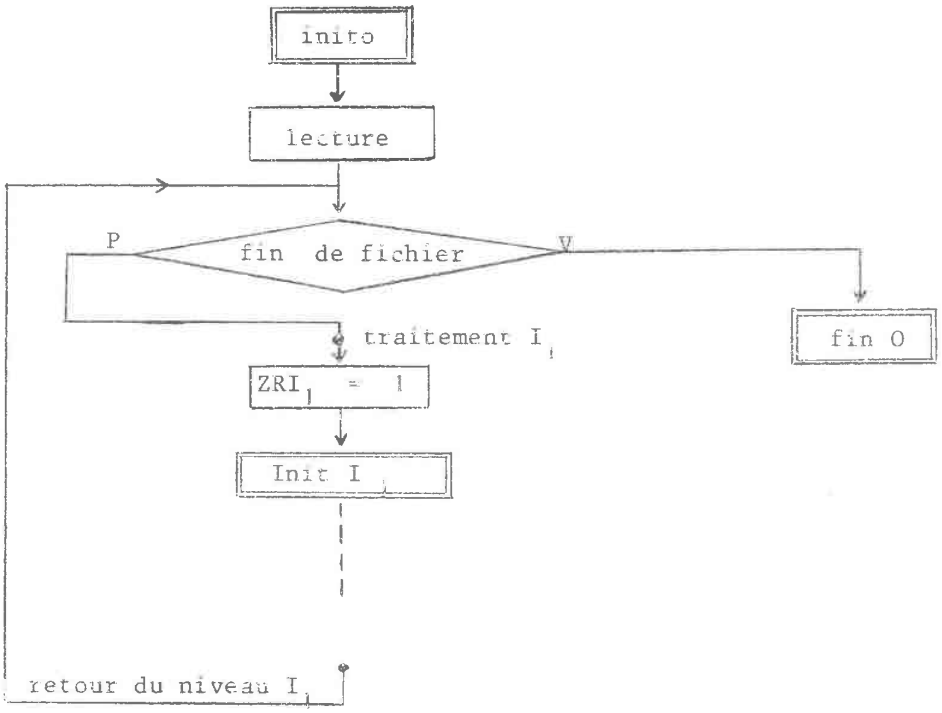
lecture : lire (file support) ; si Sdebfile alors FFIn = 1 ;

Les sous-ensembles logiques d'indicatif In étant en fait du type des éléments de l'ensemble F, le passage d'un sous-ensemble au suivant se traduit donc bien par une lecture (déplacement de l'élément courant), tandis que, pour les autres niveaux, il se traduit uniquement par une modification de la variable de travail : c'est donc au niveau de l'indicatif mineur que se placent les lectures.

24.23. Effet de l'instruction attachée à l'indicatif majeur.

Le traitement de l'instruction d'itération logique d'indicatif I_1 doit contenir des initialisations et tests de fin, en plus du traitement d'une instruction d'indicatif Ij. La forme donnée ci-dessus correspond au cas où n >1.

En fait, on pourrait considérer que l'indicatif majeur I_1 est précédé d'un indicatif implicite Io (c'est FFIn) pour lequel une rupture correspond à une fin de fichier.



init o : entier FFIn ; co FFIn est une variable de travail locale au traitement décrit ici oc ;

FFIn = o ; lire (file support) ; si $debfile alors FFIn = 1 ;

fin de fichier : il y a fin du fichier si FFI_n = 1 ;

fino : ce sont les instructions qui suivent le fpc de l'instruction d'indicatif I_1 (le traitement est terminé).

24.24. Récapitulation.

On peut donc considérer qu'un groupe d'instructions d'itération logique simple imbriquées, attachées dans cet ordre aux indicatifs I_1, I_2, ... In, est équivalente à la suite des déclarations et de l'appel de l'appel de procédure suivants :

FFIn entier ; ZRI_1 type I_1 ; ZRI_2 type I_2 ; ... ; ZRIn - 1
type In - 1 ;

procédure proc 0 ; FFIn = 0 ;
lire (file support) ; si $ debfile alors FFIn = 1 ;

décision

FFI n ≠ 0	F	V
fin 0		1
proc 1	1	
retable	2	

ft.

fin proc ;

procédure proc 1 ; .... fin proc ; procédure proc 2 ; .... fin proc ;

procédure proc j ;
ZRIj = Ij ;
Init Ij ;
tant que ∄ rupture de niveau ≤ j faire proc j + 1 fpc ;
fin Ij fin proc ;

procédure proc In - 2 ; .... fin proc ;

procédure proc n - 1 ;
ZRIn - 1 = In - 1 ;
Init In - 1 ;
tant que test faire
traitement In ;
lire (file support)
si $ debfile alors FFIn = 1 finproc ;

booléen procédure test ;
test = ZRI_1 == I_1 et ZRI_2 == I_2 et .... et ZRIn - 1 ==
In - 1 et
FFIn / 1 finproc ;

proc 1 ;

On a vu, au paragraphe 9.645., une forme très condensée utilisant le métalangage de CIVA et décrivant un ensemble de procédures permettant de gérer les ruptures d'une seule file support dans le cas particulier où chaque indicatif constitue une identification des

éléments de la file (auquel cas une rupture de niveau j correspond toujours à une rupture de niveau j - 1). Cette forme ne correspond pas au cas général : la programmation des points de rupture est souvent réalisée, à tort, de cette façon, sans que l'on soit sûr que l'hypothèse ci-dessus est toujours vérifiée.

24.3. EVALUATION D'INSTRUCTIONS D'ITERATION LOGIQUE MULTIPLE IMBRIQUEES.

24.31. Organisation générale.

On pourrait envisager de réaliser effectivement l'interclassement des différentes files qui interviennent dans un groupe d'instructions imbriquées mais cela nous amènerait à construire effectivement une file qui n'aurait d'intérêt que pour le traitement décrit par les instructions imbriquées. Il est donc préférable de réaliser cet interclassement au moment du traitement effectif.

Nous retrouvons ici l'organisation des traitements décrite en 24.2., complétée de deux façons :

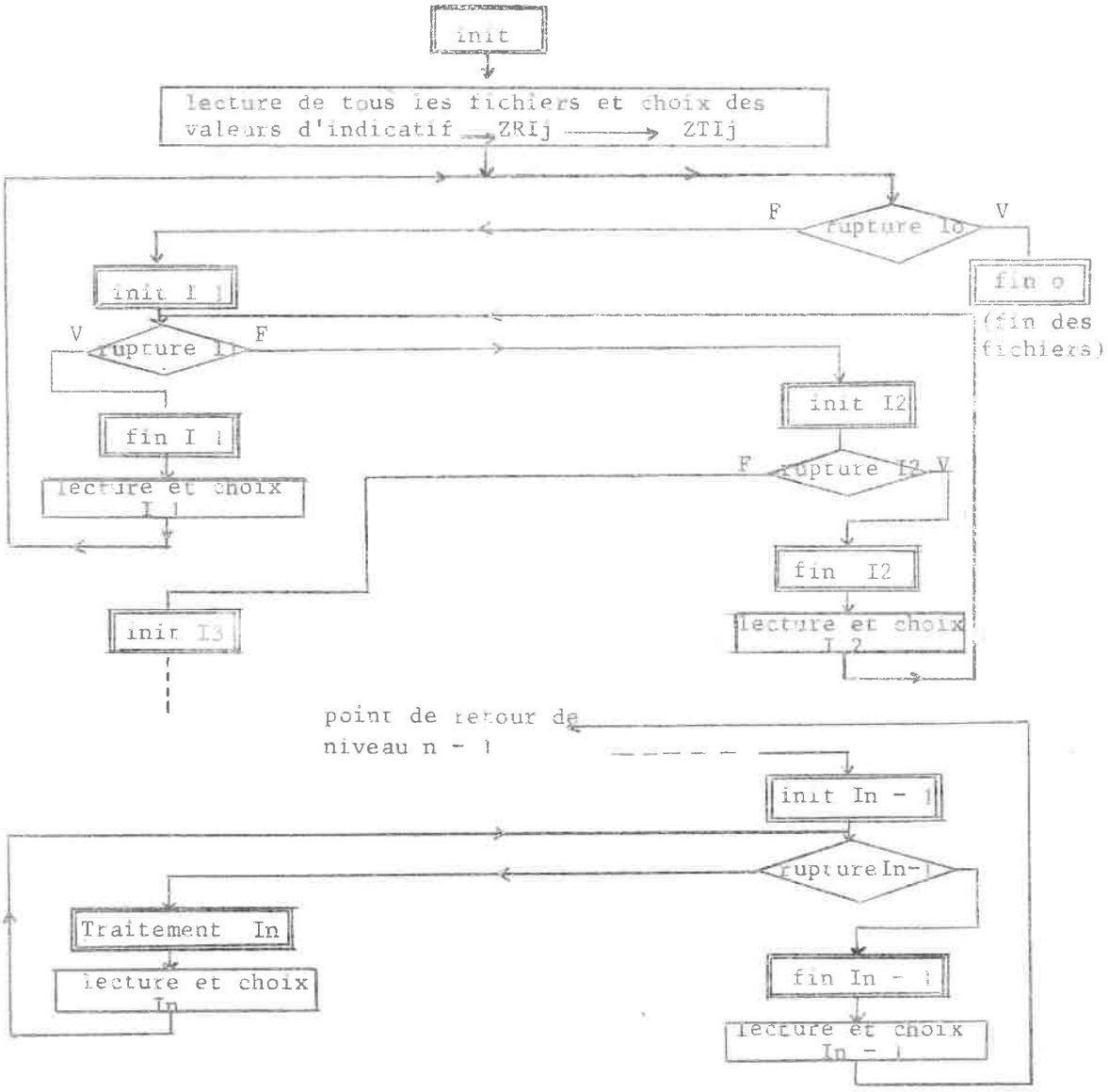
- avant d'entreprendre une itération pour l'indicatif I_j il faut déterminer quelle valeur de cet indicatif va être traitée : la plus petite (resp. la plus grande) des valeurs d'indicatifs possédées par les éléments courants des sous-ensembles logiques d'indicatif I_j si l'option d'ordre pour cet indicatif est croissant (resp. décroissant).
  Cette valeur déterminée, l'itération peut être entreprise.

- l'étude de la présence d'une rupture devant être faite aussitôt après une lecture, le choix des valeurs d'indicatifs étant nécessaire pour cette étude, celui-ci devra donc être fait, pour tous les indicatifs, immédiatement après les lectures.

- après avoir exécuté le traitement de l'itération pour l'indicatif I_j il faut procéder aux déplacements des éléments courants. Tous les éléments courants ne progressent pas : uniquement ceux dont l'indicatif I_j avait la valeur choisie pour cette itération.

Dans les descriptions qui suivent, nous nous plaçons, comme en 24.2., dans le cas général de l'utilisation des instructions

d'itération logique multiples et non seulement dans le cas d'instructions appliquées à des sous-ensembles de CIVA. La différence essentielle réside dans le fait qu'on doit placer effectivement des ordres de lecture dans le cas général. Les lectures sur un fichier se placent toujours au niveau de l'indicatif mineur pour ce fichier.



En plus des variables de travail ZRIj introduites en 24.2., nous utiliserons des variables ZTIj :

ZRIj contient la plus petite des valeurs d'indicatifs de niveau Ij, c'est la valeur choisie pour être traitée.

ZTIj contient après le choix d'une valeur de niveau j + 1, la valeur de l'indicatif Ij dans l'élément choisi au niveau j + 1.

24.32. Opérations de lectures et de choix.

"Lectures Ij" désigne les opérations de lecture à faire pour tous les fichiers qui ont Ij pour indicatif mineur. Parmi ces fichiers, seuls doivent être lus ceux pour lesquels les valeurs de l'indicatif Ij dans les enregistrements présents sont égales à la valeur qui vient d'être choisie et traitée, c'est-à-dire ZRIj, et qui ne sont pas encore terminés.

"Choix Ij" désigne le choix de tous les indicatifs d'ordre inférieur ou égal à j de la nouvelle valeur à traiter. Ce choix n'est à faire que si Ij est un indicatif mineur d'un des fichiers et si, à ce niveau, il y a eu une lecture effective sur ce fichier. Le choix s'opère en commençant par les indicatifs majeurs :

- parmi toutes les valeurs d'un indicatif, dans les enregistrements présents, choisir la plus petite (resp. la plus grande) si les fichiers sont classés selon l'ordre croissant (resp. décroissant) de cet indicatif : la valeur choisie est attribuée à ZRIj.

- choisir ensuite la valeur de l'indicatif d'ordre supérieur j + 1.

Toutes les valeurs d'indicatifs étant choisies jusqu'au niveau j, un test de rupture permet de déterminer quel traitement entreprendre : une rupture sur le niveau j est caractérisée par la non égalité de ZRIj et ZTIj ; s'il y a rupture, il convient de faire les opérations de fin du niveau j sinon on peut traiter le niveau j + 1.
On constate donc que seuls les indicatifs et les fichiers auxquels ils sont attachés déterminent l'organisation de la solution et que celle-ci est indépendante des traitements précis à effectuer à chaque niveau.

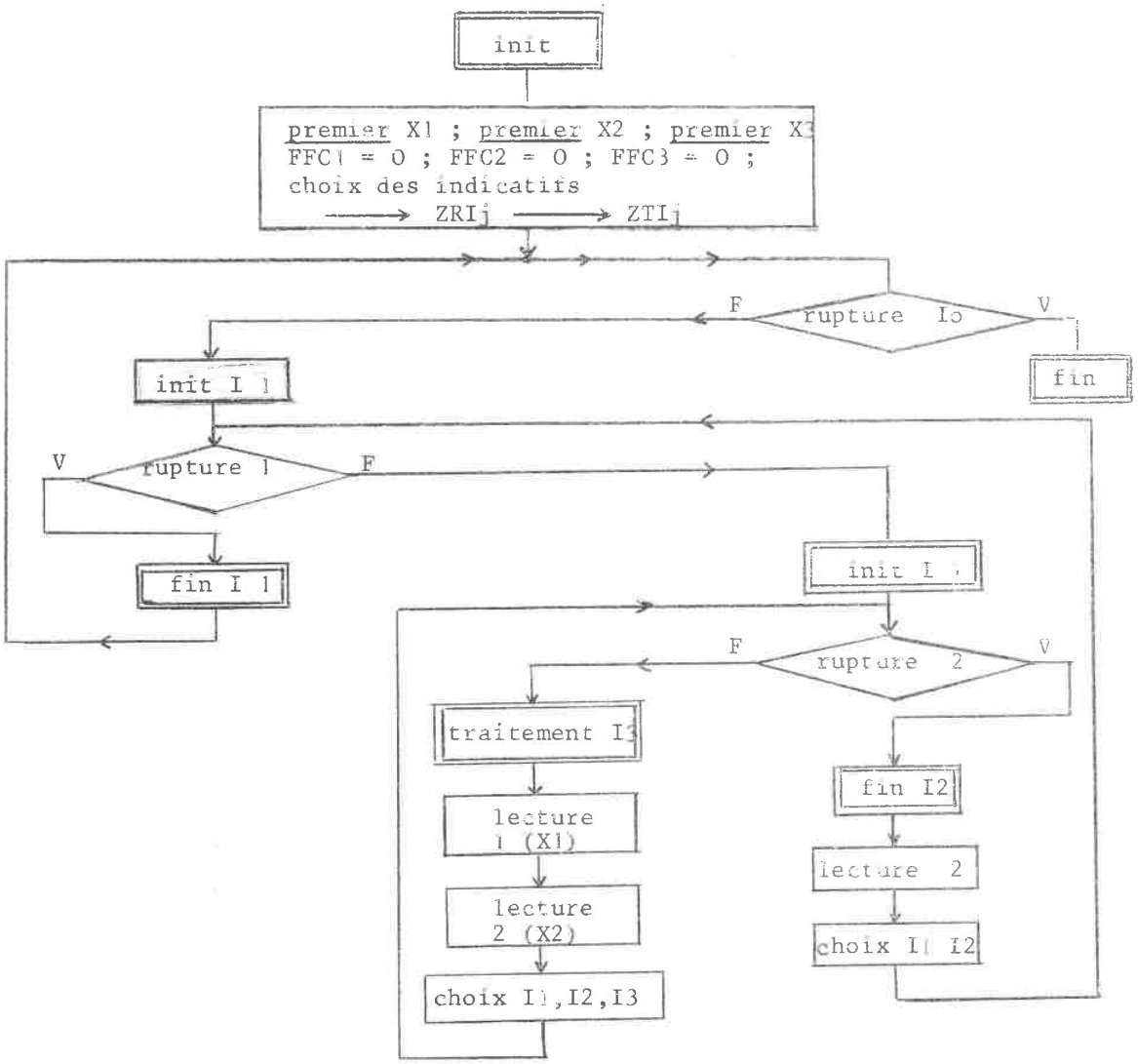
24.33. Exemple d'organisation.

Envisageons l'exemple de trois fichiers F1, F2, F3, F1 et F2 ont pour indicatifs I1, I2, I3 que l'on supposera entiers, tandis que F3 n'a que les indicatifs I1, I2. Les fichiers sont supposés triés

par I1, I2, I3 croissant. Examinons l'organisation des traitements dans le cas de 3 instructions d'itération logique multiple imbriquées.

F1, F2, file article 1 ; F3 file article 2 ; X1 de F1 ; X2 de F2 ; X3 de F3 ; I1, I2, I3 indicatif de F1 ; I1, I2, I3 indicatif de F2 ; I1, I2 indicatif de F3 ; init ;
Pour chaque I1 faire
    init I1 ; pour chaque I2 faire
        init I2, pour chaque I3 faire
            traitement I3 fpc ;
        fin I2 fpc ;
    fin I1 fpc ; fin ;

On suppose donc implicitement que les trois fichiers sont directeurs et que tous les "sens" sont croissant ; cette séquence d'instructions conduit alors à un traitement qui pourrait être décrit par le schéma suivant :



Les procédures utilisées sont définies ci-dessous :

```
procédure lecture 1 (X) ;
        si I1 de X == ZR1 et I2 de X == ZR2 et I3 de X == ZR3
        alors X en X + 1 ;
             si $ debfile alors FFC1 = 1 ; I1 de X =
                                        max entier fsi
        fsi
        fin proc ;
```

La procédure lecture 1, ci-dessus, n'effectue le déplacement de l'élément courant X que pour les files dont les valeurs d'indicatifs sont celles qui viennent d'être traitées (ZRI, zone de référence).

```
procédure lecture 2 ;
        si I2 de X 3 == ZR2
        alors X 3 en X 3 + 1 ;
              si $ debfile alors FFC3 = 1 ; I1 de X 3 =
                                              maxentier fsi
        fsi
        fin proc ;

procédure choix des indicatifs ;
        ZT1 = ZT2 = ZT3 = maxentier ;
        test 1 (X 1) ; test 1 (X 2) ; test 1 (X 3) ;
        ZR1 = ZT1 ; ZR2 = ZT2 ; ZR3 = ZT3 fin proc ;

procédure test 1 (X) ;
```

sélection	I1 de X ZT1	V	+F	+F
	I1 de X = ZT1	+F	V	V
	I2 de X ZT2	-	V	+F
	I2 de X = ZT2	-	+F	V
	I3 de X ZT3	-	-	V
action	ZT1 = I1 de X	1		
	ZT2 = ZT3 = maxentier	2		
	ZT2 = I2 de X		1	
	ZT3 = maxentier		2	
	retable	3	3	
	ZT3 = I3 de X			1

```
ft fin proc ;

booléen procédure rupture 2 ;
        rupture 2 = (FFC1 ≠ 0 et FFC2 ≠ 0) ou
                    ZR1 ≠ ZT1 ou ZR2 ≠ ZT2
        fin proc ;
```

procédure choix I1 I2 ;

sélection	I1 de X3 ZT1	V	+ F
	I1 de X3 = ZT1	+ F	V
	I2 de X3 ZT2	-	V
action	ZT1 = I1 de X3 ;	1	
	ZT2 = I2 de X3 ;	2	1
	ZR2 = ZT2	3	2

ft

fin proc ;

booléen procédure rupture 1 ;

rupture 1 = (FFC1 ≠ 0 et FFC2 ≠ 0 et FFC3 ≠ 0) ou
ZR1 ≠ ZT1

fin proc ;

booléen procédure rupture Io ;

rupture Io = FFC1 ≠ 0 et FFC2 ≠ 0 et FFC3 ≠ 0

fin proc ;

procédure choix I1, I2, I3 ;

ZT1 = ZT2 = ZT3 = maxentier ;

test 1 (X 1) ; test 2 (X 2) ; test 3 (X 3) ; ZR3 = ZT3 ;

fin proc ;

On pourrait donc décrire ce traitement par la procédure :

```
procédure exemple ;
        init ;
        premier X 1 ; premier X 2 ; premier X 3 ;
        FFC1 = 0 ; FFC2 = 0 ; FFC3 = 0 ;
        choix des indicatifs ;
        tant que non rupture Io faire
                init I1 ;
                tant que non rupture 1 faire
                        Init I2 ;
                        tant que non rupture 2 faire
                                traitement I3 ;
                                lecture 1 (X 1) ;
                                lecture 1 (X 2) ;
                                choix I1, I2, I3 fp ;
                        fin 2 ;
                        lecture 2 ;
                        choix I1 I2 fp
                fin I1 fp ;
        fin fin proc ;
```

La complexité de la solution est importante. Elle est due essentiellement au fait que c'est l'ensemble des indicatifs qui constitue une identification des objets (cf. 8.23.) et non chaque indicatif séparèment.
Remarquons que nous avons pourtant introduit une simplification en supposant que les trois fichiers F1, F2, F3 étaient des fichiers directeurs. Si un ensemble n'est pas directeur pour un indicatif donné, la valeur issue de cet ensemble ne doit pas intervenir dans le choix des valeurs d'indicatif à traiter. Si la valeur choisie est plus grande (ou plus petite si le "sens" de cet indicatif est décroissant) que celle issue de cet ensemble, on est en présence d'une erreur ; d'où la nécessité de faire progresser de 1 l'élément courant de cet ensemble et d'éditer un message d'erreur.
La complexité de la solution ne doit pas faire oublier la simplicité d'emploi des instructions d'itération logique multiple imbriquées.

REFERENCES DU CHAPITRE 24

HERTSCHUH N., 74 Problèmes d'analyse dans l'utilisation du projet CIVA.
Thèse 3ème cycle. NANCY 1. 1er trimestre 74.

FIEGEL C., 74 Traduction des instructions d'itération dans le projet CIVA. Application à Cobol.
Thèse 3ème cycle. NANCY 1. 4ème trimestre 74.

CHAPITRE 25

LA RELIURE

CHAPITRE 25

LA RELIURE

25.1 INTRODUCTION.

L'étude de l'implantation des objets en mémoire a été confiée à Jacques DENDIEN. Il en a rendu compte dans (Dendien J., 73). Nous en présenterons ici les grandes lignes et les principaux résultats. Nous présenterons ensuite quelques aspects de la réalisation qui ont été décrits dans un document interne (Fiegel C., 73) et les résultats des essais effectués.

Nous avons vu en 16.32, que la compilation d'un module ou d'une classe conduit à la production d'un module objet repéré par une définition externe et à la définition d'une zone des variables de l'unité compilée : elle est repérée par une référence externe et sa taille est indiquée dans le fichier descriptif des modules et des classes de son application. L'éditeur de liens de Siris 7 conduira, à partir de tous les modules objets des classes et modules impliqués dans une chaîne de traitement, à la constitution d'un module de chargement dans lequel toutes les références externes doivent être satisfaites : il convient donc de placer les unes par rapport aux autres toutes les zones de constantes et les zones de variables de la chaîne.

Habituellement, c'est le rôle de l'utilisateur que de déterminer l'implantation des différents constituants d'un programme : soit consécutivement, soit en suivant un arbre de recouvrement qu'il précise à l'éditeur de liens.

En Civa, le fichier descriptif des classes et des modules contient les tailles des zones de constantes et des zones de variables de taille fixe pour toutes les unités de l'application, en particulier pour celle de la chaîne de traitement à éditer. Il contient également le graphe des relations "Appelle" et "Utilise" sous forme de listes de sucesseurs. Il est donc possible d'automatiser la recherche d'une

structure de recouvrement et partant, de minimiser la place occupée en mémoire par le module de chargement obtenu.
Le rôle du pré-éditeur de liens est de fixer les emplacements relatifs des différentes zones de constantes et de variables à l'intérieur du module de chargement d'une chaîne de traitement en cherchant à déterminer un recouvrement optimal.

Il permet donc de dégager le programmeur des difficultés d'utilisation d'un éditeur de liens qui peuvent être grandes dans le cas de grands programmes.

Nous appelons opération de reliure l'ensemble des traitements permettant de passer des informations fournies par le compilateur pour les modules et classes d'une unité de traitement, à un module de chargement exécutable. La pré-édition et l'édition de liens constituent en Civa l'opération de reliure.

Le langage n'autorisant pas d'appel récursif de modules ni de procédures, il est possible de prévoir l'emplacement des constantes et des variables de taille fixe avant de passer à l'exécution : leur nombre et leur taille sont connus à la compilation.
Pour les objets de taille variable, un pointeur situé dans la zone des objets de taille fixe permet de les situer dans une zone réservée à l'ensemble des objets de tailles variables d'une chaîne de traitement (cf. chap. 23). Le pré-éditeur de liens ne s'en préoccupe donc pas.
Assurer une gestion statique de la mémoire augmente le temps de préparation d'un module de chargement, mais aussi, accèlère considérablement son exécution. En effet, le travail de gestion de la mémoire est transféré de l'exécution à la préparation, pour les objets de taille fixe et de plus l'accès à ces objets peut être réalisé par adressage direct, plus rapide.

L'intérêt de la pré-édition de liens est donc évident pour les objets variables de taille fixe : gain de mémoire grâce au recouvrement, gain de temps à l'exécution.

Pour les constantes, il est peut-être moins évident qu'il faille toujours proposer leur recouvrement : leur texte doit être stocké sur une mémoire secondaire et l'exécution du module de chargement ainsi obtenu est allongé par des chargements depuis cette mémoire secondaire. Un recouvrement ne sera proposé pour ces objets que lorsque l'insuf-

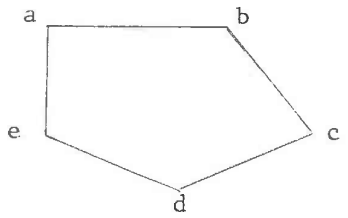
fisance de mémoire disponible l'imposera.

Dans la présentation qui suit on ne trouvera que les énoncés de propriétés des graphes des relations "appelle" et "utilise" qui serviront à définir l'algorithme employé dans le pré-éditeur. Les démonstration sont données dans (Dendien J., 73).

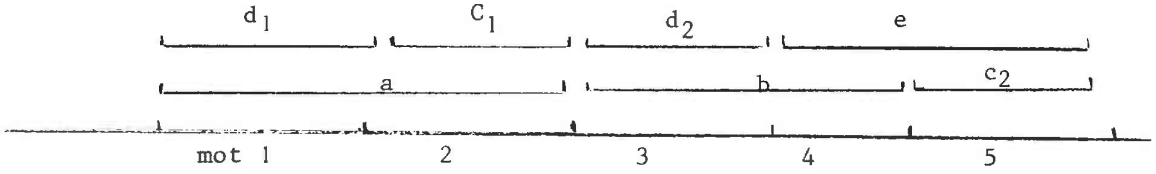
25.2 REGROUPEMENT DES OBJETS D'UNE MÊME UNITE.

Nous avons vu en (1.5), que la durée de vie des objets Civa ne dépend que de la durée d'exécution des modules. Tous les objets déclarés dans une même classe, ou tous les objets déclarés dans un même module, ont donc la même durée de vie. Plutôt que des objets indépendants, nous considérerons donc tous les objets de même nature (constante ou variable) d'une même unité comme un seul élément : une zone, dans laquelle ces objets seront placés de façon contiguë.

Ceci peut entraîner une légère perte de place en mémoire comme on peut le constater sur l'exemple ci-dessous :
Soient 5 zones de variables à implanter, chaque zone contenant deux objets occupant chacun un mot, et soit le graphe exprimant les contraintes de non recouvrement :



dans lequel deux sommets sont joints par une arête s'ils correspondent à deux zones nécessairement disjointes en mémoire, c'est-à-dire ne pouvant pas être en recouvrement. Il est possible de touver une implantation en mémoire des dix objets correspondants avec un recouvrement ne nécessitant que 5 mots :



alors qu'on montre qu'il est impossible de le faire si on impose que les deux objets d'une même zone soient contigus (Dendien J., 73).

25.3 GRAPHE DE COEXISTENCE DES ZONES EN MEMOIRE.

Les zones de variables de deux classes ou modules u_1 et u_2 sont dits disjointes si elles ne peuvent pas être en recouvrement.
Ces deux zones ne peuvent pas être en recouvrement s'il existe un instant t de l'exécution de l'unité de traitement, appartenant à la fois à la durée de vie des objets de u_1 et u_2, ce qui peut s'exprimer sous la forme :

$$\exists x \in M \quad \exists y \in M \quad \big((x \hat{A} y \ \underline{ou}\ y \hat{A} x) \ \underline{et}\ x \hat{U} u_1 \ \underline{et}\ y \hat{U} u_2 \big) \Longrightarrow u_1 \text{ et } u_2 \text{ disjoints.}$$

La relation $u_1 R u_2 \Longleftrightarrow u_1$ et u_2 sont disjoints est symétrique et elle peut être représentée par un graphe non orienté, qui exprime donc les règles de coexistence des zones en mémoire.
La relation peut donc être définie par

$$u_1 R u_2 \Longleftrightarrow u_1 \ \hat{U}^{-1} \ . \ \hat{A} \ . \ \hat{U} \ u_2 \ \underline{ou} \ u_1 \ \hat{U}^{-1} \ . \ \hat{A}^{-1} \ . \ \hat{U} \ u_2$$

Numérotons les modules d'une unité de traitement de 1 à m et ses classes de m+1 à m+c.
La matrice booléenne associée au graphe ($M \cup C$, R) est telle que

$$[R](i, j) = 1 \quad \Longleftrightarrow \quad x_i \ R \ x_j$$
$$[R](i, j) = 0 \quad \Longleftrightarrow \quad \underline{non}\ x_i \ R \ x_j$$

Soit $[A]$ la matrice asscciée au graphe $(M, \hat{A})$ et $[\tilde{U}]$ la matrice associée à la fermeture transitive de "utilise" pour les modules : c'est la matrice associée du graphe $(M \cup C, \hat{U})$ réduite à ses m premières lignes

et soit $[\tilde{A}] = [A] + [A]^t$
la matrice sysmétrique, somme booléenne de A et de sa transposée.

$$[R] = [\tilde{U}]^t \cdot [\tilde{A}] \cdot [\tilde{U}]$$

nous verrons en 25.53, que cette matrice peut être calculée facilement.

25.4 RECHERCHE DE L'IMPLANTATION DES ZONES DE VARIABLES.

Soit E l'ensemble des zones de variables de taille fixe des modules et des classes d'une unité de traitement, et soit $|E| = n$. Appelons $O(x_i)$ l'adresse, dans la partie du module de chargement réservé à ces zones, de l'origine de l'emplacement affecté à la zone correspondant à x_i.

Chaque zone étant munie d'une longueur entière $\ell(x_I)$, on appelle <u>répartition pour E, R, ℓ</u> un ensemble de n valeurs entières $O(x_1), \ldots$ $O(x_n)$ telles que :

$$x_i \; R \; x_j \Longrightarrow \begin{cases} O(x_i) + \ell(x_i) \leq O(x_j) \\ \underline{\text{ou}} \; O(x_j) + \ell(x_j) \leq O(x_i). \end{cases}$$

On appelle <u>longueur</u> d'une répartition V, la taille de mémoire nécessaire pour contenir cette répartition, c'est-à-dire

$$L(V) = \max_{i \in [1,n]} \left(O(x_i) + \ell(x_i) \right) - \min_{j \in [1,n]} \left(O(x_j) \right)$$

Le problème de l'implantation optimale des zones de variables d'une unité de traitement consiste donc à chercher une répartition pour E, R, ℓ, la plus courte possible.
Pour trouver un algorithme général nous utiliserons les notions de graphe représentatif d'une famille d'intervalles.
Soit I un ensemble d'intervalles fermés de l'ensemble des réels. On appelle <u>graphe représentatif de I</u>, un graphe non orienté dont les sommets sont les éléments de I et où deux sommets sont joints par une arête si et seulement si les deux intervalles correspondants sont disjoints.

Le graphe complémentaire d'un graphe représentatif d'une famille d'intervalles est un graphe de comparabilité, c'est-à-dire un graphe

non orienté pouvant être orienté pour en faire le graphe d'une relation d'ordre.

Considérons une répartition V pour (E, R, ℓ). Les valeurs $O(x_i)$ déterminent une famille d'intervalles dont on peut construire le graphe représentatif (E, G). Soit (E, R') son graphe complémentaire, c'est-à-dire un graphe tel que :

$$\forall x, y, (x\ G\ y \Longleftrightarrow \underline{\text{non}}\ x\ R'\ y)$$

(E, R') est un graphe partiel de (E, R), et l'on a :

$$x\ R'\ y \Longrightarrow \begin{cases} O(x) \geqslant O(y) + \ell(y) & (a) \\ \underline{\text{ou}}\ O(y) \geqslant O(x) + \ell(x) & (b) \end{cases}$$

Pour définir une relation d'ordre T, choisissons par exemple d'orienter l'arête xy de x vers y si l'inégalité (b) est vérifiée et de y vers x si c'est (a). Le graphe (E, T) caractérise une répartition.

Appelons <u>longueur d'un chemin par les sommets</u>, la somme des longueurs $\ell(x_i)$ de ses sommets.

On démontre (Dendien J. 73) les propositions suivantes :

<u>Proposition 1</u> :

L (V) $\geqslant$ Lo où Lo est la longueur du plus grand chemin par les sommets du graphe(E, T).

Soit V, une répartition pour (E, R, ℓ), on appelle <u>graphe associé à la répartition V</u>, le graphe (E, S) défini par

$$x\ S\ y \Longleftrightarrow x\ R\ y\ \underline{\text{et}}\ O(y) \geqslant O(x) + \ell(x).$$

(E, S) est un graphe partiel de (E, T). On a :

<u>Proposition 2</u> :

L (V) est au moins égal à la longueur du plus grand chemin par les sommets du graphe (E, S) associé à la répartition V.

<u>Proposition 3</u> :

Une répartition V est de longueur minimale (dans l'ensemble des répartitions possibles pour (E, R, ℓ), si et seulement si sa longueur est égale à la longueur du plus grand chemin par les sommets de son graphe associé.

Proposition 4 :

De tout graphe sans circuit (E, S) déduit de (E, R) en orientant ses arêtes, on peut déduire une répartition dont la longueur est la longueur du plus grand chemin par les sommets de (E, S).

Le problème est alors de trouver, parmi toutes les orientations possibles de (E, R) celles qui conduisent au plus grand chemin par les sommets, le plus court possible. On en déduit alors une répartition de longueur minimale.
La proposition suivante permet de retrouver la solution classique d'un problème très proche du nôtre et appelé "problème des sessions d'examens" (Kaufmann A., 68) et (Dufourd J. F., 71). Elle consiste à colorer le graphe (E, R), de telle sorte que deux sommets adjacents n'aient pas la même couleur. On appelle nombre chromatique d'un graphe le nombre minimum de couleurs permettant de le colorer de cette manière (Berge C. 68). Les épreuves peintes de la même couleur peuvent se dérouler simultanément, ce qui conduit, en orientant les arêtes du graphe à une organisation de la session. On suppose dans ce cas que les longueurs des sommets sont toutes égales. La solution trouvée est alors la plus courte possible.

Proposition 5 :

Si (E, R) est p- chromatique, les graphes sans circuit déduits de (E, R) par orientation de ses arêtes, comportent un chemin passant par p sommets au moins, et il existe des orientations de (E, R) donnant des graphes sans circuit dont tout chemin comporte au plus p sommets.

Lorsque les longueurs sont inégales, le choix de l'orientation est déterminant. Ce choix de l'orientation se ramène au choix d'un ordre sur les p couleurs d'un graphe, où p est le nombre chromatique de ce graphe. Un ordre conduisant au même résultat que son ordre inverse, il faudrait donc examiner 1/2 p! orientations possibles et chercher chaque fois le plus grand chemin par les sommets, ce qui est impensable.

Nous utiliserons donc la méthode décrite ci-dessus en faisant l'hypothèse que les longueurs par les sommets sont toutes égales. L'écart entre la longueur de la répartition trouvée et la répartition optimale peut être évaluée (Dendien J. 73).

Pour l'implantation des zones de constantes, les problèmes posés sont les mêmes et la même méthode sera donc employée. Néanmoins, la réalisation du relieur nous amène à distinguer le cas des constantes à cause essentiellement du chargement de ces zones pendant l'exécution.

25.5 REALISATION DU RELIEUR.

25.51 Organisation générale du pré-éditeur.

L'organisation générale du pré-éditeur peut être décrite sommairement par le module suivant. Le pré-éditeur et les liaisons que l'éditeur de liens de Siris 7 sont décrites précisément, en Civa, dans un document interne (Fiegel C. 73).

module pré-éditeur ; $ classes utilisées ;
calcul fermeture transitive (M, A, A^*) ;
calcul fermeture transitive (E, U, U^*) ;
graphe de coexistence (E, R) ;
coloration (E, R) ;
orientation (E, R) ;
essai implantation (E, R, ℓ_v) ;

si mémoire suffisante alors recouvrement sur les variables ; sortir fsi ;
essai implantation (M, A, ℓ_c) ;

si mémoire suffisante alors recouvrement sur les variables, et les
constantes des modules ; sortir fsi ;
essai implantation (E, R, ℓ_c) ;
si mémoire suffisante alors recouvrement sur toutes les zones
sinon imprimer ('place insuffisante').
fin mod ;

ℓ_v (resp. ℓ_c) représente la file des longueurs des zones variables (resp. des zones de constantes) des classes et modules de l'unité de traitement à éditer.

25.52 Calcul des fermetures transitives.

Pendant la reliure, les modules et les classes d'une unité de traitement à relier sont numérotés de 1 à n pour les modules, de n à n + c pour les classes. Les graphes (M, A) et (E, U) sont représentés par leur matrice associée (file de files de booléens).

En fait, cette matrice étant représentée en mémoire comme une matrice de bits, l'algorithme de Marshall nous semble préférable (Derniame - Pair 1971).

25.53 Recherche du graphe de coexistence.

On a vu en 25.3 que $[R] = [\tilde{U}]^t \cdot [\tilde{A}] \cdot [\tilde{U}]$.
Or $\tilde{U}$ se présente ainsi $[\tilde{U}] = \left[\begin{array}{c|c} I_n & \Delta \\ \hline & c \end{array}\right]^n$ dans laquelle seule la partie Δ intervient dans le calcul . D'où :

$$[R] = \begin{bmatrix} \bar{A} & \bar{A}.\Delta \\ \Delta^t.\bar{A} & \Delta^t \bar{A} \Delta \end{bmatrix}$$

soit B une matrice $B_{[n, p]}$ et $C_{[m, p]}$, définissons le produit $\odot$ par :

$$D_{[n, m]} = B \odot C = B \cdot C^t = \left[\bigvee_{k=1}^{p} B[i, k] \wedge C[j, k] \right]$$

$$R = \begin{bmatrix} \bar{A} & A \cdot {}^t \\ \Delta^t \odot \bar{A} & (\Delta^t \odot \bar{A}) \odot \Delta^t \end{bmatrix}$$

Le calcul de $[R]$ sous cette forme est beaucoup plus rapide, car il permet d'utiliser les instructions AND et OR de CII 10070 qui effectuent m et (resp. ou) sur des chaînes de 32 bits.
Le temps de calcul de $[R]$ sous cette forme est proportionnel à $m * c$ alors qu'il serait proportionnel à $n^2 * c$, par un calcul direct.

25.54 Coloration de (E, R).

Le graphe (E, R) est coloré par une méthode heuristique due à Welsh et Powell, qui donne d'assez bons résultats comme nous le verrons ci-dessous.

Algorithme :

(1) Numéroter de 1 à n les sommets, par degrés décroissants ;

i = 1 ;

(2) Colorer le point d'indice le plus faible à l'aide d'une couleur C_i, et colorer avec la même couleur le point d'indice le plus faible non adjacent aux points déjà colorés par C_i et recommencer (2), tant qu'un tel point existe.

Quand il n'en eixiste plus, on passe à la couleur suivante :

i = i + 1 et on recommence (2), s'il reste des points non colorés.

On montre que la dernière valeur de i ainsi obtenue est un majorant du nombre chromatique du graphe.

Des essais ont été effectués portant sur des familles de graphes de 32, 64, 128 et 256 sommets générés au hasard, mais en possédant un nombre chromatique et une densité d'arcs [1] fixés à l'avance. 80 graphes ont été construits. Dans 74 cas le nombre trouvé par la méthode de Welsh et Powell est identique au nombre chromatique connu. Dans les 6 autres cas la différence n'est que d'une unité.

25.55 Implantation des objets.

Le graphe (E, R) est préalablement orienté pour en faire un graphe sans circuit : nous prendrons l'orientation définie par le sens des numéros de couleur croissants :

s'il existe une arête de i à j dans (E, R) alors

si couleur (i) > couleur (j) alors R (i, j) = 0

sinon R (j, i) = 0 fsi fsi ;

L'adresse d'implantation de chacun des objets 0 (x_i) est donnée par la longueur par les sommets du plus grand chemin d'extrémité x_i. Ces longueurs sont données par un algorithme de cheminement utilisant une pile (Derniame - Pair, 71).

(1) La densité d'arcs d'un graphe sans circuit est $\frac{2 \times m}{n\,(n-1)}$, où m est le nombre d'arcs et n le nombre de sommets du graphe.

25.6 LIAISON ENTRE LE PRE-EDITEUR ET L'EDITEUR DE SIRIS 7.

25.61 Cas du recouvrement des zones de variables uniquement.

C'est le cas le plus simple, le pré-éditeur construit un module objet définissant l'adresse de l'origine de chacun des zones de variables des modules ou des classes et réservant une place totale égale à la longueur de la répartition obtenue. L'éditeur est alors appelé par une procédure M : Link (CII SIRIS 7 - 1) pour réaliser l'édition de ce module objet avec les autres modules objets de l'unité de traitement. La reliure est alors terminée. Le module de chargement obtenu est exécutable. En général, il est immédiatement exécuté.

25.62 Recouvrement des zones de constantes des modules seuls.

La structure de recouvrement défini par le pré-éditeur n'est, en général, pas une arborescence et l'édition avec recouvrement de Siris 7 ne peut être utilisée.

On procède alors à une édition séparée de chacun des modules, les modules de chargement obtenus sont rangés dans un fichier partitionné. Pendant l'exécution de l'unité de traitement, la présence d'un module est vérifiée et son chargement éventuel assuré par un programme résident.

Ce programme est édité avec les autres constituants de l'unité de traitement : le module objet des zones de variables, les zones de constantes des classes, les modules de librairie (sous-programmes de conversion, etc...)

Le résident utilise une table, $$PER, des points d'entrée dans le résident, contenant pour chaque module, une instruction BAL, RL1 $$T où $$T est l'adresse de début du résident. Un appel du module x_j est remplacé par un appel de $$PER + j.

Tout appel de module aboutit donc à l'adresse $$T avec dans le registre RL1, la quantité $$PER + j + 1, qui permet de déterminer quel était le module appelé. Une table de présence des modules $$PTMC permet de vérifier s'il faut charger le module appelé et une table des points d'entrée dans le module, $$PEM permet d'effectuer l'appel du module désiré.

Le résident, les tables $$PER, $$PEM, $$T doivent être fournis par le pré-éditeur à l'éditeur, pour l'édition du résident.

Pour les éditions séparées de modules, toute référence à un module est satisfaite par la définition $$PER + j. Le pré-éditeur fournit également pour chaque module, une liste des modules qu'il peut recouvrir ; cette information sera utilisée par le résident pour mettre à jour la table de de présence des modules lors d'un chargement.

25.63 Recouvrement des zones de constantes des modules et des classes.

Les zones de constantes des classes sont aussi éditées séparément, comme pour les modules. A l'appel d'un module m, le résident doit également vérifier que toutes les classes de U (m) sont présentes en mémoire. D'où la nécessité de conserver également la matrice U, pendant l'exécution, sous une forme appropriée. C'est donc un traitement encore plus lourd que dans le cas précédent, ce qui explique qu'il n'est entrepris qu'en cas de nécessité absolue.

25.7 CONCLUSION.

Le pré-éditeur est actuellement réalisé et fonctionne dans les trois cas ci-dessus et l'ensemble du relieur est opérationnel.

Remarquons cependant que la lourdeur des solutions apportées en 25.62 et 25.63 est essentiellement liée aux éxigences de l'éditeur de Siris 7, et qu'une version de Civa devant être souvent utilisée devrait comporter un relieur spécifique adapté aux règles de recouvrements de Civa et intégrant les fonctions du pré-éditeur et de l'éditeur de liens.

REFERENCES DU CHAPITRE 25

BERGE C. , 68 Théorie des graphes et ses applications
Dunod Paris 1968.

CII Procédures métasymbol Siris 7, 1

DENDIEN J., 73 a) Gestion statique de mémoire dans un système de programmation modulaire.
Doctorat d'Ingénieur - Université de Nancy 1
Mars 1973.

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

DERNIAME J. C., PAIR C., 71
Problèmes de cheminement dans les graphes
Dunod, Paris 1968.

DUFOURD J. F., 71 Traitement automatique de la gestion scolaire d'une université.
Doctorat de 3ème cycle - Université de Nancy 1. Déc. 1971.

FIEGEL C., 73 Le relieur. Etude et réalisation
Document interne Civa - Université de Nancy 1
Juillet 1973.

KAUFMANN A., 68 Introduction à la combinatorique en vue des applications.
Dunod Paris 1968.

WELSH et POWELL

CONCLUSION

Ceci ne peut être qu'une conclusion momentanée car, si de nombreux points du projet sont déjà étudiés, les travaux présentés ici ne peuvent constituer qu'une étape dans la réalisation du projet CIVA.

Tout d'abord, certains points doivent être complétés ou revus, comme par exemple l'acquisition des données, pour laquelle de nouveaux métamodules d'acquisition devront être proposés, ainsi que pour l'édition. La réalisation du système lui-même pourrait être améliorée. Le compilateur actuellement en voie de mise au point, complété par les procédures de la classe de modification, pourra être utilisé pour définir les versions suivantes du compilateur. Si l'on proposait alors une version plus générale et plus souple du métalangage permettant, en particulier, de définir de nouvelles macroinstructions dont l'appel pourrait prendre la présentation syntaxique des instructions, on aurait alors la possibilité d'étendre à volonté le langage et d'assurer au fur et à mesure la mise à jour du système.

Mais la poursuite des travaux suppose que l'on espère déboucher sur une utilisation importante de nos outils, ou, tout au moins, de l'esprit dans lequel nous proposons d'aborder la réalisation d'une application, esprit qui devrait pouvoir se répandre facilement, ce qui nous semblerait d'une grande utilité pour la réalisation d'applications importantes. Pour que le langage puisse également s'imposer, il devrait entrer maintenant dans une phase d'essais et de modifications et pour cela, il nous semble qu'il faudrait que le groupe de travail actuel s'élargisse et qu'un nombre important de réalisations soient effectuées.

La principale voie de travaux qui s'ouvre alors devant nous réside dans l'étude et la définition d'une méthode d'analyse qui permette d'établir facilement le lien entre l'énoncé d'un problème et la description de sa solution en CIVA. En effet, la proposition actuelle consiste en la définition d'un certain nombre d'outils sans toutefois avoir défini une méthode d'emploi. Cependant, ces outils ont été

conçus pour être bien adaptés à la définition d'une telle méthode, qu'il ne reste pratiquement plus qu'à mettre en forme. Elle ne devrait pas être tellement différente des méthodes existantes, si ce n'est par l'exploitation que l'on peut faire de la modularité des descriptions. Prenons par exemple le concept de "mot" dans la méthode MINOS : "un mot est la plus petite quantité d'information ayant une signification en soi du point de vue de l'entreprise qui puisse être utilisé de façon autonome". MINOS utilise l'idée naturelle de saisir et cataloguer tous les mots de l'entreprise une fois pour toutes. Le vocabulaire de l'entreprise ainsi défini est hiérarchisé : il est structuré par l'existence de différents services : le vocabulaire d'un service peut être décrit par une classe d'application ; la description des mots pourra être fournie progressivement dans d'autres classes.

L'étude des relations entre les mots conduit à un regroupement logique des données (lot d'information). La description de ces groupements logiques sous forme de structures élémentaires indépendantes peut être faite dans de nouvelles classes qui "utilisent" les précédentes.

La synthèse des lots d'information logiques conduit à dégager des lots d'information logiques plus importants qui permettent de passer aux fichiers logiques : ils sont décrits par des structures "plus grandes" dans des classes qui "utilisent" les précédentes.

Enfin, l'étude des contraintes de présentation des données liées à leur saisie, lorsqu'il s'agit de données "fraîches", et des contraintes de présentation dues aux traitements, conduit à préciser la présentation externe des enregistrements de ce fichier.

D'autre part, l'étude des circuits d'information dans l'entreprise peut mettre en évidence un certain nombre de stations liées entre elles pour assurer une "procédure" de traitement. Un système de traitement de l'information est composé d'un ensemble de procédures. Chaque procédure peut être décomposée en un certain nombre de tâches (tâches de décision, de contrôle, de mise à jour, de calcul, d'édition) et chacune de ces tâches peut être décrite par un (ou des) module (s). Il est alors possible de préciser quelles tâches sont automatisables et de définir (HERTSCHUH N., 74) des règles de regroupement des tâches en

unités de traitement s'appuyant sur la chronologie d'exécution des tâches et sur leurs "niveaux de traitement", c'est-à-dire sur les relations entre les indicatifs auxquels elles peuvent être rattachées.

Les traitements élémentaires constituant les tâches ayant été décrits par des modules, leur regroupement conduit à la définition d'un module (unité de traitement) qui "appelle" les précédents. Les travaux en cours de Messieurs HERTSCHUH, BARTHELEMY, CHABRIER s'inscrivent dans ce cadre et tendent à définir cette méthode.

Peut-être pourrait-on songer alors à étendre le domaine d'application de CIVA à la réalisation de banque de données, en définissant des structures de données peut-être moins rudimentaires que celles de file, la version actuelle étant surtout intéressante pour l'existence des classes d'application, la possibilité de décrire des calculs au niveau du module de commande (dont il faudrait sans doute rendre l'interprétation interactive, ce qui permettrait d'assurer un accès direct aux données pour les "interroger"), la possibilité de décrire des traitements différés (unité de traitement) dans le même langage que pour l'interrogation, la possibilité de décrire modulairement la structure des informations à traiter et d'en assurer facilement la modification.

NOM DE L'ETUDIANT : DERNIAME Jean Claude

NATURE DE LA THESE : DOCTORAT D'ETAT EN SCIENCES MATHEMATIQUES

VU, APPROUVE

& PERMIS D'IMPRIMER

NANCY, le 15 Janvier 1974

LE PRESIDENT DE L'UNIVERSITE DE NANCY I

UNIVERSITÉ DE NANCY I
Le Président
J.R. HELLUY