ISSN 0291 - 5413

Publication du Département de Mathématique & d'Informatique

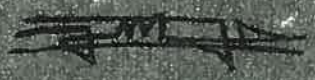
MARS 2005 EMPR.
JUIN 2005 RETOUR.

123466

BULLETIN d'INFORMATIQUE

approfondie & applications

J.-M. KNIPPEL

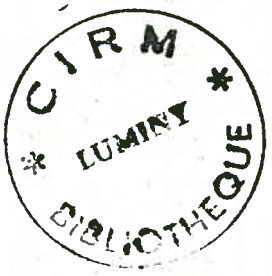


Nº 2

Laboratoire d'Informatique Théorique

ISSN 0291 - 5413

Ana 20 Gdb



Informatique fondamentale et Applications

Editeur
L.I.T.L.

Comite de Redaction :

E. Bianco
G. Cousin
F. Donnat
P. Isoardi
J.P. Lehmann
J. Roller
R. Stutzmann

Depositaire

G. Ambard

Sommaire

J.-M. KNIPPEL

Mars 1982

Faculté des Sciences de Luminy
Département de Mathématique-Informatique
70, route Léon-Lachamp - Case 901
13288 MARSEILLE CEDEX 9

LA RECHERCHE ET LE CHERCHEUR.

E. BIANCO

Il n'est pas vraiment simple, tout de go, de réussir à établir un lien, si ténu soit-il, entre des mots comme "recherche", "équipe de recherche", "connaissance", "progression dans la connaissance", "projets", "programmes", etc... Je dis bien mots car il ne m'est pas évident déjà que les notions que l'on est censé leur faire correspondre, n'aient pas une certaine tendance à fluctuer selon la bouche ou la plume du personnage qui les exprime. Et encore les circonstances ont-elles peut-être une influence.

J'aimerais traduire le doute qui habite mon esprit en posant quelques questions.

Prenons par exemple La Connaissance, et La Recherche comme moyen d'accroître la connaissance. Il paraît raisonnable alors de mesurer la connaissance afin simplement de pouvoir constater qu'il est apparu quelque part un accroissement.

Pour que l'on ne me fasse pas sur le champ un mauvais procès, j'insiste sur le fait que je ne suis pas le promoteur de la machine permettant de mesurer les connaissances d'un individu par lecture de la déviation d'une aiguille sur un écran gradué. A chaque mesure ses moyens appropriés, et l'on sait faire, je crois pour beaucoup que l'on y réfléchisse.

Dans ce champ de la connaissance qui est la propriété d'un groupe, chaque individu du groupe va pouvoir puiser sa part. Nous voyons poindre de nouveaux mots tels que : "pédagogie", "contrôle des connaissances", etc...

En ce point de ma réflexion apparaît une contradiction que j'introduis brutalement : nombre de grands esprits ont déclaré, en substance, que plus ils avançaient dans le domaine de la connaissance, plus cela mettait en lumière leur ignorance. La connaissance apparaissant ainsi comme un moyen d'évaluer l'ignorance.

J'essaierai de souligner, de mon modeste point de vue, qu'à partir de ce moment de nombreuses contradictions commencent à affleurer.

Il est un phénomène bien connu dans l'enseignement officiel : les connaissances, entendons par là un certain niveau de compréhension pour sujet en général très réduit, ne sont ressenties comme vraiment acquises qu'après le passage d'un examen. Et cela m'autorise à quelques commentaires sur l'interprétation d'un tel fait. Il semble que ce soit la concentration qui soit à l'origine du miracle. Ce qui importe c'est que l'on décide de se concentrer à ce moment là plutôt qu'au moment où sont exposées les questions. Une enquête un peu poussée montre que le véritable moteur est en fait l'obtention du diplôme en tant qu'agent abstrait de valorisation. Et ce n'est pas, comme on pourrait croire, la manifestation d'une curiosité profonde envers la connaissance.

On serait tenté alors de dire qu'importe le moteur ... Oui mais le résultat n'est pas du tout le même. Dans un cas on a une acquisition parcellaire avec incompréhension totale sur les liens qu'on peut établir entre diver lambeaux épars. En fait, on a appris à passer des examens, ce qui est une technique comme une autre à laquelle on peut se révéler brillant, c'est un jeu aux règles tout aussi abstraites et rigoureuses que celles de n'importe quelle autre branche de la connaissance, mais ce n'est pas la connaissance. On peut être un remarquable passeur d'examens, sans pour autant maîtriser cette connaissance dont le diplôme devrait être la garantie d'acquisition.

Dans l'autre cas il faut pouvoir prendre le temps de réfléchir afin de mener à bien deux tâches simultanées qui sont en fait de taille. Essayer d'établir des ponts solides entre tous ces ilôts de connaissance clairsemés, et entreprendre l'exploration de nouveaux domaines. Cela en tant que tel représente une expérience totalement neuve pour l'étudiant et il semble vain de vouloir à tout prix la borner dans le temps de manière arbitraire. Les meilleurs résultats obtenus, c'est-à-dire l'éclosion d'une personnalité n'étant en rien liés à la rapidité de la transformation.

Quelles sont les raisons d'une telle situation ? Sans doute sociales, culturelles, historiques, en tout cas complexes.

Nous entrevoyons déjà la difficulté qu'il va y avoir à accroître des connaissances incertaines. Je précise : comment, possédant des connaissances incertaines, participer efficacement à l'accroissement du fond commun ? C'est précisément un de ces mystères que j'aimerais cerner dans ces lignes.

Je vais essayer de tirer quelques conséquences, bien sûr un peu hâtives, de cet état de fait mais il faut faire quelques hypothèses pour tenter de progresser si peu que ce soit.

Il résulte des choses présentées comme je viens de le faire, qu'il y a beaucoup de chances de rencontrer deux catégories d'attitudes chez les gens qui "mènent" la recherche. Il y aura ceux qui, poursuivant la méthode scholastique, en feront réponse à un questionnaire préétabli. Ce sont des gens qui vont demander à leurs étudiants de répondre aux questions qu'eux-mêmes ont eu beaucoup de mal à se poser. Il y aura de la même manière ceux qui, voulant rompre avec la sacro-sainte méthode, essayeront de créer un climat favorable au sein duquel les étudiants pourront essayer de se poser eux-mêmes des questions, à charge ensuite d'y répondre. Sans aller jusqu'à envisager d'habiles mélanges des deux méthodes, on peut imaginer de comparer leurs applications dans la réalité.

S'appliquant à des groupes d'étudiants dont les modes de formation sont évidemment identiques, ces deux méthodes auront certainement des conséquences très différentes : dont la pire est que la seconde demandera beaucoup plus de temps que la première. Apprendre à se poser des questions n'est pas chose couramment enseignée dans nos civilisations. On pourrait tout de même penser que la deuxième méthode est plus enthousiasmante dans la mesure où elle devrait permettre de développer la curiosité inhérente de chaque individu. Or, qu'en est-il en réalité ? Eh bien, lorsqu'ils ont le choix, les étudiants préfèrent la première méthode. Est-ce un constat d'échec ? Autre question importante.

Je ne le pense pas. Une petite enquête auprès des étudiants peut faciliter la compréhension. A l'exposé des attitudes différentes qu'on peut ainsi avoir et à l'exposé de leurs raisons, les étudiants présentent deux sortes de réponses.

Ils expriment leur surprise, on en leur a jamais tenu un tel langage auparavant.

Ils montrent bien sûr un intérêt poli, mais ils n'ont pas de temps à consacrer à des choses somme toute secondaires.

Là, je me permets encore quelques commentaires.

Elle n'a pas été abrogée cette prescription qui oblige à passer un 3ème cycle en deux ans. Quelle était donc l'intention du législateur de l'ancien régime ? Obtenir la formation de techniciens "supérieurs" ? Ce voeu était, je crois, clairement exprimé. Mais alors pourquoi se plaindre si le niveau de "l'innovation" paraît à l'étiage. Se figure-t-on que l'on apprend à avoir des idées personnelles, (peut-on innover sans avoir des idées personnelles), en ressassant et en ânonnant des choses bien établies, qui paraissent tourner bien rond, bref de belles théories, quel que puisse être l'intérêt de celles-ci par ailleurs.

Et puis il y a la question des moyens de survie. Celui qui persiste malgré tout, au-delà de la durée allouée, devra se prendre en charge lui-même. L'originalité coûte cher sous nos régimes. Montrant bien comment les discours gargarisants de mots creux comme "recherche", "savoir", ne visent finalement qu'à délivrer un satisfécit au support de l'idée en place, soulignant cependant la peur profonde de tout ce qui pourrrait, peut-être, faire un peu avancer la réflexion. Mais qui risquerait de remettre en question bien des vides.

Quelle magnifique occasion manquée de combler ces vides, ou tout au moins d'essayer.

En effet l'informatique était une occasion de sortir si c'est possible de la gangue. Née, je dirai même surgie brusquement, elle ne laissait pas par son irruption le temps à des cadres traditionnels de se trouver formés aux bonnes traditions. C'est donc un sang neuf, comme on dit, qui irriguait ces nouvelles artères. Les informaticiens s'étant formés sur le tas, méthode qui en vaut une autre, on pouvait espérer voir poindre d'autres manières d'enseigner ; au moins saluer l'apparition d'une branche neuve de la connaissance par l'imagination et l'expérimentation d'un moyen neuf de communication.

L'informatique n'est elle pas ...
La communication ne passe-t-elle pas par l'information, dont l'outil de traitement le plus puissant est l'informatique ?

Mais il reste bien vrai que les informaticiens, ces nouveaux hommes, ont une culture, solidement ancrée, qui remonte à bien plus loin que l'apparition d'une nouvelle technique. Et il aura suffi que les élèves des plus anciens se trouvent dans la place pour que déjà le ronron soit bien régulier. Combien, parmi ces anciens, n'avaient pas honte d'être des "informaticiens" terme qu'on ne pouvait accoler à aucune échelle de valeurs bien solide et qui préféraient le masque rassurant du mathématicien, même appliqué.

Mais peut-être suis-je trop pessimiste, et il existe dans l'ombre des équipes qui travaillent un peu autrement. Nos colonnes leur sont ouvertes pour parler de leur expérience et de leurs expériences, dans un échange qui pourrait être à la fois réconfortant et fructueux.

= : = : = : = : = : =

Machines , Langages

Compilation

L. NOLIN

C.R Subject Classifications Informatics - 4.21 4.22 5.26

Résumé. Programmes signifie mettre en oeuvre des moyens d'analyse et des moyens d'expression. L'expérience montre que l'on peut réunir les types de problèmes en grandes classes pour lesquelles il vaut la peine de construire un langage de programmation. Ces langages spécialisés offriront une pertinence déterminée dans le cadre de ce que l'on veut obtenir. Et les règles qui permettent d'aboutir à cette pertinence constituent le fond de cet article. L'abondance des exemples étaye fortement le raisonnement qui aboutit à l'obtention d'une construction, en fait d'un moyen d'expression qui assure les trois propriétés essentielles :

facilité, sûreté, efficacité d'expression.

L'auteur fabrique sous nos yeux la machine qui tient compte de ce que la pratique de l'informatique apporte de fondamental au programmeur.

= ; = ; = ; = ; = ; =

MACHINES - LANGAGES

COMPILATION

I. LA PROGRAMMATION.

On ne programme bien que si l'on peut s'exprimer et se faire entendre de la machine de façon aisée, sûre et efficace. Voyons cela.

LA FACILITE.

S'exprimer de manière aisée c'est, à première vue, le faire de façon "naturelle", comme on parle ou comme on écrit dans le jargon de sa spécialité. Et il semble que l'on a tout fait pour permettre cela quand on a fourni à l'usager un "langage symbolique" qui reflète ce jargon : la machine n'est-elle pas alors ignorée sauf en ce qui concerne certaines limitations (taille d'une case de mémoire, nombre de cases ...) ?

C'est un leurre, et ce pour trois raisons.

a) Tout d'abord le jargon d'une spécialité ne reflète pas, sauf exception, le fait essentiel que "calculer" signifie, précisément, "calculer la valeur d'une fonction en un point", chose qui se fait selon des règles immuables bien qu'ignorée du public.

☐ Exemple :

Soit q la fonction définie ci-dessous. C'est une fonction de deux entiers naturels (le second différent de 0) dont la valeur est un entier naturel (i.e. 0, 1, 2, ...).

(df) $q(x,y) = [\text{SI } x < y \text{ ALORS } 0 \text{ SINON } q(x-y,y) + 1]$

où encore, dans la manière d'ALGOL 60 :

```
ENTIER NATUREL.   PROCEDURE  q (x, y)  ;
ENTIER NATUREL.   x, y  ;
q := SI  x < y  ALORS  0  SINON  q (x - y, y) +1  .
```

On calcule $q(7,3)$ de la façon suivante :

$q(7,3)$	
$= [\text{SI } 7 < 3 \text{ ALORS } 0 \text{ SINON } q(7-3,3)+1]$	- df -
$= q(7-3,3)+1$	- car $7 < 3$ est FAUX -
$= q(4,3)+1$	- car $7 - 3 = 4$ -
$= [\text{SI } 4 < 3 \text{ ALORS } 0 \text{ SINON } q(4-3,3)+1]+1$	- df -
$= [q(4-3,3)+1]+1$	- car $4 < 3$ est FAUX -
$= [q(1,3)+1]+1$	- car $4 - 3 = 1$ -
$= [[\text{SI } 1 < 3 \text{ ALORS } 0 \text{ SINON } q(1-3,3)+1]+1]+1$	- df -
$= [0+1]+1$	- car $1 < 3$ est VRAI -
$= 2$	- car $0+1=1$ et $1+1=2$ -

□

Cet exemple illustre toutes les règles qu'il faut et qu'il suffit d'appliquer pour calculer la valeur d'une fonction en un point, à savoir :

R1 : changer les variables qui apparaissent dans une définition.
[par exemple, réécrire la définition ci-dessus comme suit :

```
q (dividende, diviseur) =
SI dividende < diviseur ALORS  0
   SINON  q (dividende-diviseur, diviseur) +1 ].
```

R2 : substituer des expressions à chacune des variables.
[par exemple 4 à x et 3 à y ; dans le RAPPORT ALGOL 60 c'est ainsi qu'on explique les appels de procédure, par la REGLE DE RECOPIE].

R3 : simplifier des expressions en utilisant tout ce que l'on sait dans sa spécialité.
[par exemple, puisqu'il s'agit ici d'arithmétique, remplacer $4 < 3$ par FAUX, $1 + 1$ par 2].

A l'heure actuelle on emploie encore des langages de programmation dont les auteurs paraissent ignorer ce fait essentiel.

b) La signification - la sémantique - de la plupart des langages de programmation en usage aujourd'hui est loin d'être entièrement fixée.

□ Exemple :

Le RAPPORT ALGOL 60 donne deux explications contradictoires des effets de l'instruction :

" POUR i := 1 PAS 1 JUSQU'A n FAIRE instruction ". □

La raison en est simple : c'est que les présentateurs de ces langages ignorent un précepte que deux mille ans de constructions mathématiques mettent en évidence, qui est de commencer par les objets les plus simples et les plus aisés à connaître, pour monter peu à peu et comme par degrés jusqu'à la connaissance des plus composés.

c) S'exprimer dans un langage de programmation est une chose ; faire exécuter le programme ainsi obtenu par une machine est une toute autre affaire. Point n'est besoin ici d'exemples : tout programmeur a les siens.

Programmer facilement c'est donc, certes, le faire dans un langage de programmation qui reflète le jargon de sa spécialité ; mais aussi dans un langage complet, sans ambiguïtés et tel que les programmes qu'il permet d'écrire puissent être mis en oeuvre de façon rationnelle.

LA SURETE.

Ce que l'on écrit pour calculer la valeur d'une fonction en un point doit vérifier, d'évidence, un certain nombre de conditions :

a) Un programme ne peut pas modifier un programme, quel qu'il soit.

b) Une instruction d'un programme ne peut renvoyer hors de celui-ci que dans deux cas :

- ou bien il s'agit de l'appel à un autre programme ; en vertu de la règle R2, une telle instruction n'est qu'une simple abréviation ;

- ou bien c'est le retour au programme qui a provoqué l'appel au présent programme ; l'instruction, ici, n'est qu'une parenthèse fermante.

c) Un programme ne peut employer ou modifier que les seules données spécifiées par l'auteur.

L'emploi d'un "langage symbolique" fournit une façon simple et sûre de remplir les deux premières conditions : le compilateur - ou l'interpreteur - y veille. Point n'est besoin pour cela d'un "langage de haut niveau", un simple "langage d'assemblage" suffit.

Mais, à l'opposé de ces conditions vérifiables à priori, la condition c ne peut être vérifiée le plus souvent qu'à posteriori, c'est à dire au moment de l'exécution.

□ Exemple :

Un programme emploie un tableau t - c'est à dire une fonction - de $[0 \text{ à } 7]$ dans $\mathbb{R}$. Ce n'est qu'au moment où l'on va prendre ou modifier t_i qu'il convient de vérifier que i est un élément de l'ensemble $\{0,\ldots,7\}$. □

Beaucoup de compilateurs ou d'interpreteurs négligent tout ou partie de ces vérifications. Non par ignorance, mais parce que la machine ne leur permet pas de le faire de façon efficace.

L'EFFICACITE.

Si donc un programme écrit dans un langage symbolique se traîne, la faute en est pour une bonne part à la machine. Mais elle n'est pas seule en cause.

Un compilateur a fort à faire pour trouver, d'une phrase en langage de départ, une traduction fidèle et concise en une phrase du langage d'arrivée. L'usager complique inutilement la tâche du compilateur lorsqu'il omet d'exprimer ses intentions de façon claire. Et la responsabilité retombe sur le langage qu'il emploie lorsque celui-ci ne lui en fournit pas les moyens.

Ainsi, les définitions de fonctions (donc de procédures) peuvent être classées suivant différents critères. Contentons nous ici de distinguer les définitions relatives des définitions absolues.

L'exemple donné plus haut est celui d'une définition absolue (celle du quotient entier de x par y , à une unité près par défaut). On peut l'employer (ou encore appeler la procédure qui l'exprime dans un langage de programmation) à n'importe quel moment, dans n'importe quel programme.

□ Un autre exemple de définition absolue est la suivante :

$$S\ (p, f) = [SI\ \emptyset = 0\ ALORS\ 0\ SINON\ S\ (p - 1, f) \div f\ (p)]$$

où p est un entier naturel et f une fonction à argument et valeur entiers (en fait, $S\ (p, f) = \sum_{i=1}^{p} f\ (i)$).

Considérons alors la définition suivante :

```
r (m, n, g) = s (m, k)
   [AVEC  k (i) = s (n, h)
        [AVEC  h (j) = g (i, j) ]] ,
```

ou encore, en ALGOL 60 :

```
ENTIER PROCEDURE  r (m, n, g) ;
    ENTIER  NATUREL  m, n ; PROCEDURE  g ;
  ┌ENTIER  PROCEDURE  k (i) ;
  │    ENTIER  NATUREL  i ;
  │  ┌ENTIER  PROCEDURE  h (j) ;
  │  │    ENTIER  NATUREL  j ;
  │  │    h := g (i, j) ;
  │  └
  │    k := s (n, h) ;
  └
   r := s (m, k)
```

La définition de r (qui calcule $\sum_{i=1}^{m} \sum_{j=1}^{n} g\ (i, j)$) est absolue ; celle de k , relative à r ne peut être employée que dans la procédure r ou dans une procédure relative à r ; celle de h est relative à k : elle elle ne peut être employée que dans k ou dans une procédure relative à k .

□

Dans le langage d'arrivée comme dans le langage de départ, les procédures k et h ci-dessus sont intimement liées à la procédure r ; et un procédé d'appel très simple, existant dans toute machine, suffit à les activer : mettre en évidence leur relativité, c'est donc fournir au compilateur une indication précieuse.

De façon analogue, employer dans le langage symbolique une instruction de répétition dont la "variable contrôlée" ne peut pas faire l'objet d'une affectation (à l'intérieur de la boucle), incite le compilateur à la traiter d'une façon spéciale donc avantageuse (par exemple en l'associant à un registre d'index, chose dont les machines actuelles sont abondamment pourvues).

En somme nous devons attendre d'un langage-machine qu'il permette d'appliquer de façon efficace toutes les règles du calcul des fonctions et d'effectuer rapidement les vérifications indispensables ; nous devons exiger des langages symboliques - et tout particulièrement de ceux dont les auteurs prétendent qu'ils sont "de haut niveau" - qu'ils nous permettent d'exprimer des différences subtiles touchant les définitions, les boucles et bien d'autres choses encore. Et l'idéal serait pour l'usager qu'il fût seul responsable, par ignorance ou négligence, du manque d'efficacité de sa programmation.

RESOLUTION.

Forts de ces principes nous allons tout d'abord définir, dans ce qui suit, une machine - donc un jeu d'instructions - qui permette de traduire de façon efficace les exigences formulées par l'usager dans un langage symbolique approprié.

Nous définirons ensuite, pas à pas, un tel langage. "Pas à pas", cela signifie que nous partirons d'un langage analytique et fruste - un langage d'assemblage pour notre machine - langage que, peu à peu, par le biais de définitions nous enrichirons de moyens d'expression commodes pour les processus les plus fréquemment employés. La sémantique du langage final sera ainsi absolument claire et la tâche de son compilateur parfaitement définie.

Cette méthode a fait ses preuves ailleurs ; il est grand temps qu'on l'applique sans faiblir en informatique.

= : = : = : = : = : =

II. LA MACHINE ET SON EMPLOI.

Nous allons faire quelques hypothèses de travail qui pourront être révisées quand notre étude sera plus avancée.

LA MACHINE.

La machine que nous allons décrire ressemble, quant à sa configuration, aux micro-ordinateurs actuels comme d'ailleurs, peu ou prou, à presque toutes les machines que nous connaissons depuis vingt ans. Sa seule originalité tient à ce que son langage est conçu tout spécialement pour servir le dessein que nous venons d'avouer.

Pour fixer les idées, nous supposons qu'elle contient les éléments ou dispositifs suivants.

1. Une mémoire (RAM) de 2^{16} cases pouvant contenir chacune un octet (i.e. un nombre entier naturel ≤ 255), cases qui sont numérotées de 0 à $2^{16} - 1$; chacun de ces numéros est l'adresse de la case auquel il est associé. Le contenu de la case d'adresse i est désigné par m_i $(0 \leq i \leq 2^{16} - 1)$; celui des cases i et $i + 1$, par M_i .

2. 32 registres contenant deux octets chacun, numérotés de 0 à 31. Le contenu du registre i est noté r_i , sa partie droite (r_i mod 256) r_i D , sa partie gauche ($r_i/256$, division euclidienne) r_i G .

A quelques exceptions près (voir plus loin) on peut, entre autres choses :

. les charger du contenu de deux cases de mémoire consécutives, l'adresse de la première étant fournie (en abrégé : $r_J \leftarrow M_i$), ou de celui d'un autre registre ($r_J \leftarrow r_i$) ;

. charger leur partie droite (en conservant leur partie gauche ou en la mettant à zéro, au choix) du contenu d'une case d'adresse donnée (r_J D $\leftarrow m_j$ ou $r_J \leftarrow m_J$) ou bien encore de la partie droite d'un autre registre (r_J D $\leftarrow r_i$ D ou $r_J \leftarrow r_i$ D) ;

. effectuer les opérations inverses des précédentes ;

. effectuer sur deux d'entre eux un certain nombre d'opérations (arithmétiques, logiques, ... : voir plus loin) dont le résultat est chargé dans un troisième (par exemple, $r_J \leftarrow r_i + r_k$) ;

. tester le contenu d'un registre, ou de sa partie gauche, ou de sa partie droite, et, suivant qu'il vérifie ou non une condition fixée, sauter à une instruction d'adresse donnée ou passer à l'instruction suivante ;

. les employer dans le calcul des adresses.

3. Un registre de conditions dont le contenu, noté T , est modifié quand s'effectuent certaines opérations (arithmétiques, logiques, ...) et peut être testé comme celui d'un registre.

4. Deux bases pour les adresses, chacune de deux octets, désignées par I et D .

5. Des supports pour des stockages auxiliaires et des dispositifs d'accès.

6. Un dispositif permettant de réagir de façon automatique ou programmée à des sollicitations extérieures.

7. Un jeu d'instructions qui permettent de commander les opérations auxquelles nous venons de faire allusion. Toute instruction est codée en 4 octets ; son adresse, lorsqu'elle est en mémoire, est celle de la première des 4 cases consécutives qui la contient.

8. Un compteur ordinal qui contient à tout moment -lorsque la machine est active- l'adresse de l'instruction dont l'exécution est en cours.

9. Un programme d'ordres initiaux, qui constitue l'amorce de tout système d'exploitation. Quand la machine est mise en route, ce programme -placé en ROM pendant l'arrêt de la machine- est mis en mémoire à partir de la case 0 . Alors sont posés D = 0 , I = 0 , et le compteur ordinal reçoit 1 . Cela déclenche le programme d'ordres initiaux qui prépare la réception d'une requête de l'usager (lire quelque part et placer à tel autre endroit programmes ou données, activer un programme, ...) et se met en attente. La requête reçue, il s'exécute et (sauf bouclage) se replace

en attente de la prochaine requête.

LES PROGRAMMES.

Un programme - entendez par là un programme produit par un compilateur ou un assembleur - est en principe une suite d'instructions numérotées de 4 en 4 à partir de 0 , instructions qui sont soumises à des contraintes que nous allons exprimer peu à peu.

Supposons qu'un programme soit placé en mémoire à partir de la case i (donc jusqu'à la case $i + 4n - 1$, s'il comporte n instructions) ; pour le faire dérouler on fait exécuter une instruction qui signifie :

$$I \leftarrow i \ ; \ \text{VERS } I$$

(i.e. I devient i puis, sans interruption possible, le compteur ordinal reçoit I).

Alors la première instruction du programme est exécutée, puis la suivante (si la précédente n'est pas un saut, un appel ou un retour) et ainsi de suite. Lorsqu'il s'agit d'un saut à l'instruction numérotée k , celui-ci est interprété pour la machine comme un passage du compteur ordinal à la valeur $k + 1$.

Lorsqu'il s'agit d'un appel le présent programme, P_k , crée pour le programme appelé, P_{k+1} , une nouvelle zone de données, ZD_{k+1} , commençant à la première case laissée libre par la zone de données, ZD_k , du programme appelant.
(Il faut parfois quelques calculs - qu'on peut placer au début du programme P_{k+1} , pour déterminer l'étendue de cette nouvelle zone ; tel est le cas lorsque ce programme manie des tableaux dont la longueur n'est connue qu'au moment de son activation).

Désormais activé, le programme P_{k+1} peut seulement modifier, de façon directe, le contenu des cases de sa zone associée, soit ZD_{k+1} , et de façon indirecte (car cette zone peut lui fournir, de proche en proche, des adresses de cases situées dans des zones antérieures) les contenus des cases des zones ZD_k , ZD_{k-1} , ... , Z_1 ; cette dernière étant associée au premier programme appelé par le programme d'ordres initiaux.

Soit J_{k+1} l'adresse de (la première case de) la zone de données ZD_{k+1}. L'appel du programme P_{k+1} se fait par des instructions qui signifient :

D devient J_{k+1}

I devient i_{k+1} ; VERS I

Ainsi, toute référence à des données dans le programme P_{k+1} peut-elle être faite à des adresses commençant à 0, si l'on convient qu'à l'exécution ces adresses relatives sont automatiquement indexées par D. (Ainsi, $r_3 \leftarrow m_{10}$ signifie : $r_3 \leftarrow m_{10+D}$). Les programmes sont donc réentrants et l'exécution de procédures récurrentes ne pose aucun problème (si ce n'est le coût de leur exécution).

De plus, comme les zones $ZD_1, \ldots, ZD_{k+1}$ sont choisies de façon telle qu'elles ne recouvrent pas les emplacements des programmes correspondants $P_1, \ldots, P_{k+1}$, il en résulte qu'un programme reste invariant au cours de toute exécution.

C'est, naturellement, le compilateur (ou l'interpréteur, ou l'assembleur) du langage symbolique employé par l'usager qui vérifie que toutes les conditions énumérées ci-dessus sont remplies (quoique certaines précautions puissent être confiées aux bons soins de la machine : voir plus loin).

Revenir du programme P_{k+1} au programme P_k qui l'a appelé consiste à rétablir les valeurs précédentes de D et de I et à sauter à l'instruction qui suit l'appel de P_{k+1} dans P_k.

Une telle façon de faire constitue (à quelques variantes près) la seule manière de satisfaire aux règles générales du calcul des fonctions (voir plus haut R1 et R2).

LES INSTRUCTIONS.

Occupons-nous pour l'instant des instructions qui opèrent sur des données placées en mémoire ou dans des registres, données qui sont les opérandes de ces instructions. Ces opérandes sont désignés par des expressions.

Pour les opérandes-registres, c'est tout simple ; nous avons déjà vu

$$r_i \quad , \quad r_i\,G \quad , \quad r_i\,D \qquad (0 \leq i \leq 31) \ .$$

Mais certaines opérations dont nous verrons bientôt des exemples concernent deux registres étroitement associés (par exemple, lorsqu'on explore un tableau, on augmente de 1 l'adresse de l'élément courant tout en diminuant de 1 le nombre des éléments restants). On associe donc deux à deux les registres de numéros i et $i + 1 \bmod 32$, désignant les 4 octets qu'ils contiennent par

$$R_i$$

(lorsque $i = 31$, alors $i + 1 \bmod 32 = 0$).

Et, par analogie avec la notation adoptée pour les registres proprement dits, on écrit aussi :

$$R_i\,G \text{ (i.e. } r_i) \quad \text{et} \quad R_i\,D \text{ (i.e. } r_{i+1 \bmod 32}) \ .$$

Ainsi les opérandes registres sont-ils désignés par l'une ou l'autre des expressions suivantes :

$$\left\{\begin{matrix} R \\ r \end{matrix}\right\} i \left\{\begin{matrix} \text{rien} \\ G \\ D \end{matrix}\right\} \qquad (0 \leq i \leq 31) \ .$$

C'est un peu plus compliqué pour les opérandes-mémoire. L'expérience montre que, dans ce cas, quatre types d'expressions sont particulièrement employés :

type 1 : $r_i + d$

type 2 : $r_i + \left\{\begin{matrix} M \\ m \end{matrix}\right\} (d + B)$

type 3 : $\left\{\begin{matrix} M \\ m \end{matrix}\right\} (r_i + d + B)$

type 4 : $\left\{\begin{matrix} M \\ m \end{matrix}\right\} (r_i + M\,(d + B) + B)$

où d est une constante désignant un octet (i.e. un entier compris entre 0 et 255) et B l'une des bases D ou I. (On notera que tout appel à la mémoire comporte l'indexation par une base).

Ce n'est pas un hasard. En effet :

1. - On obtient un facteur immédiat d'un octet (d) avec $i = 0$, si on pose par principe $r_0 = 0$;

- On a un opérande-registre (r_i) si $d = 0$;

- On exprime l'adresse d'un élément d'un tableau limité, t, lorsque r_i est l'adresse de t_0 et d l'indice de cet élément (ou inversement).

2. Ce type d'expressions fournit l'adresse d'un tableau quelconque, t, lorsque $M(d + B)$ est l'adresse de t_0 et r_i l'indice de l'élément.

Les types d'expressions 3 et 4 permettent donc de désigner, entre autres choses, des éléments de tableaux.

Naturellement, dans ces cas là, la base est D (on verra pourtant que cette remarque souffre quelques exceptions).

Les types d'expressions 1 et 3 peuvent également servir à désigner des adresses d'instructions ; dans ce cas la base est I : ce sont des renvois

En principe, une instruction comporte trois parties, à savoir :

- un type d'opération (TO)
- un opérande-registre (OR)
- un opérande-mémoire (OM) ou un renvoi (RV)

ce qui permet de la coder en 4 octets comme on l'a dit plus haut.

□ Exemples :

- SI OR remplit une condition VERS RV
- idem, SINON opération sur OR
- recopier les k registres { suivant / précédant } OR dans OM
- opération inverse de la précédente
- OR ← OR opération OM
  (cas particuliers : OR ← OR opération { facteur immédiat / OR }
- OM ← OM opération OR

Etc... □

Il nous faut maintenant justifier la présence de ces instructions... et d'autres encore.

= : = : = : = : = : = : =

III. LES INSTRUCTIONS.

A. L'ITERATION.

L'itération d'un calcul est un procédé fondamental. Un exemple va nous permettre d'imaginer un moyen simple et efficace de la noter.

□ DONNEES

Deux suites : $s = s_1, s_2, \ldots, s_n$ et $s' = s'_1, s'_2, \ldots, s'_{n'}$
où les s_i et les $s'_{i'}$ sont des éléments d'un ensemble ordonné.

PROBLEME

Les comparer lexicographiquement
(réponse : pp, eg, pg pour : $s < s'$, $s = s'$, $s > s'$).

SOLUTION 1

(qui n'est rien d'autre qu'une formulation plus détaillée, sinon plus précise du problème) :

comp (s, s') - def
SI s = ∅ ALORS test (s')
SINON
SI s' = ∅ ALORS pg
SINON
SUIVANT QUE tête (s) [= , < , >] tête (s') :
[comp (queue (s), queue (s')), pp, pg] ;
AVEC
test (s') = SI s' = ∅ ALORS eg SINON pp .

NB. Lorsque $s \neq \emptyset$:
tête (s) est le 1er élément de s ,
queue (s) est s amputée de son 1er élément. □

C'est là une solution récurrente : si on la traduit telle quelle en ALGOL 60 par exemple, on obtient une PROCEDURE qui contient un appel à cette procédure même ; cela manque d'efficacité.

Et tout programmeur sait en tirer une solution itérative. L'analyse d'une telle transformation est très instructive ; d'autant qu'on peut en suivre facilement les étapes si on passe des suites à leur représentation en mémoire.

Supposons pour simplifier que les s_i et les s'_J sont tous représentables par des octets. Alors les suites s et s' occupent en mémoire les cases J à $J + n - 1$ et J' à $J' + n' - 1$, respectivement. (Si quelqu'un objecte que, ce faisant, nous bricolons, nous reprendrons volontiers le même discours en termes de fonctions, de composition et de prolongement ou de restriction). A tout moment, on en considère des queues de longueurs $\lambda = n - i$ et $\lambda' = n' - i$, ayant leurs premiers éléments dans les cases $\beta = \gamma + i$ et $\beta' = \gamma' + i$, respectivement, i prenant successivement les valeurs 0, 1, ..., minimum (n, n').
Ainsi - la possibilité d'accéder à la mémoire étant sous entendue - ces suites sont caractérisées à tout instant par les couples $< \beta, \lambda >$ et $< \beta', \lambda' >$ respectivement. D'où une nouvelle façon de formuler la solution.

□ SOLUTION 2

comp ($< \beta, \lambda >$, $< \beta', \lambda' >$) = def
SI $\lambda = 0$ ALORS test (λ')
SINON
SI $\lambda' = 0$ ALORS pg
SINON
SUIVANT QUE m_β [= , < , >] $m_{\beta'}$:
[comp ($< \beta + 1, \lambda - 1 >$, $< \beta' + 1$, $\lambda' - 1 >$) , pp, pg]
AVEC
test (λ') = SI $\lambda' = 0$ ALORS eg SINON pp ,

solution qu'il suffit d'appliquer aux valeurs initiales :

$< \beta , \lambda > = < \gamma , n >$
$< \beta' , \lambda' > = < \gamma' , n' >$ □

Pour transformer cette nouvelle récurrence en itération, il suffit de substituer, dans le cours de la boucle, $< \beta + 1 , \lambda - 1 >$ à $< \beta , \lambda >$ et $< \beta' + 1 , \lambda' - 1 >$ à $< \beta' , \lambda' >$.

On y parvient aisément en considérant à tout moment les couples $< \alpha , \lambda > = < \beta - 1 , \lambda >$ et $< \alpha' , \lambda' > = < \beta' - 1 , \lambda' >$ au lieu des couples $< \beta , \lambda >$ et $< \beta' , \lambda' >$. D'où une nouvelle solution :

□ SOLUTION 3

comp ($< \alpha , \lambda >$, $< \alpha' , \lambda' >$) = def
SI $\lambda = 0$ ALORS test (λ')
SINON
$< \alpha, \lambda > \leftarrow < \alpha + 1 , \lambda - 1 >$;
SI $\lambda' = 0$ ALORS pg
SINON
$< \alpha' , \lambda' > \leftarrow < \alpha' + 1 , \lambda' - 1 >$;
SUIVANT QUE m_α [= , < , >] $m_{\alpha'}$:
[comp ($< \alpha , \lambda >$, $< \alpha' , \lambda' >$) , pp, pg] ,

solution qu'il suffit d'appliquer aux valeurs initiales

$< \alpha , \lambda > = < \gamma - 1 , n >$
$< \alpha' , \lambda' > = < \gamma' - 1 , n' >$. □

Il suffit d'un petit nombre d'instructions d'une machine bien conçue pour l'exprimer.

On représente un couple $< \alpha , \lambda >$ par $< R_J G , R_J D >$.
Alors :

. Un test suivi de la substitution de $< \alpha + 1 , \lambda - 1 >$ à $< \alpha , \lambda >$, par exemple, peut s'exprimer dans une seule instruction

SI $R_J D = 0$ VERS renvoi
SINON $R_J G \leftarrow R_J G + 1$ et $R_J D \leftarrow R_J D - 1$

. En mettant en évidence le rôle joué dans l'algorithme par les couples $< \alpha , \lambda >$ ou $< \beta , \lambda >$, on a rappelé opportunément qu'à toute suite est associée sa longueur et qu'on ne fait que concrétiser la chose en plaçant ladite longueur en tête de la représentation d'une suite en mémoire. Nous appellerons désormais chaîne un tel couple < longueur , suite > .

En supposant pour simplifier que nos suites ont des longueurs d'un octet au plus, nous aurons donc, par exemple :

$$c = < n , s_1 , \dots , s_n > = < m_\mu , m_{\mu+1}, \dots , m_{\mu+n} >$$

pour une certaine adresse μ .

Alors on peut initialiser la boucle ci-dessus par une seule instruction :

$$R_J G \leftarrow \mu \ ; \ R_J D \leftarrow m_\mu$$

ou, plus généralement :

$$\boxed{R_J G \leftarrow \text{une adresse} \ ; \ R_J D \leftarrow m\,(R_J G)}$$

. Comme l'aiguillage final contient deux opérandes mémoire et trois renvois (qu'on peut réduire à deux) il faut le décomposer ; voici une façon de faire :

$r_k \leftarrow m\,(R_J G)$

$r_k \leftarrow r_k - m\,(R_J{}^1 G)$

SI $r_k D = 0$ VERS reprise

SI $r_k G = 0$ VERS pp

pg : ……

i.e. $r_k G \leftarrow 0 \ ; \ r_k D \leftarrow m\,(r_J)$

i.e. $r_k \leftarrow r_k + (256 - m\,(r_{J'}))$

car, après la construction :

$r_k D = 0$ ssi $m\,(r_J) = m\,(r_{J'})$

$r_k G = 0$ ssi $m\,(r_J) < m\,(r_{J'})$

$r_k G = 1$ ssi $m\,(r_J) \geq m\,(r_{J'})$

En supposant enfin que la réponse pp, eg, pg est notée - 1, 0, 1 et qu'elle est placée dans la case, nous aurons donc comme solution du problème posé le programme en langage machine qui suit.

PROGRAMME

$r_1 \leftarrow 0$

$R_2G \leftarrow \mu$; $R_2D \leftarrow m(R_2G)$

$R_4G \leftarrow \mu'$; $R_4D \leftarrow m(R_4G)$

reprise : SI $R_2D = 0$ VERS test SINON $R_2G \leftarrow R_2G + 1$ et $R_2D \leftarrow R_2D - 1$

SI $R_4D = 0$ VERS pg SINON $R_4G \leftarrow R_4G + 1$ et $R_4D \leftarrow R_4D - 1$

$r_6 \leftarrow m(r_2)$

$r_6 \leftarrow r_6 - m(r_4)$

SI $r_6D = 0$ VERS reprise

SI $r_6G = 0$ VERS pp

pg : $r_1 \leftarrow 1$

VERS fin

test : SI $R_4D = 0$ VERS fin

pp : $r_1 \leftarrow -1$

fin : $m(\nu) \leftarrow r_1$

□

Chaque ligne de ce programme est une instruction-machine. Pour l'incorporer dans un programme particulier, il suffit de remplacer μ , μ' , ν et les étiquettes par des valeurs convenables (et peut-être changer les numéros des registres pour éviter de détruire le contenu de ceux qui, de 1 à 6, seront encore utiles dans la suite du programme en question) ; on peut laisser ce soin à un programme d'assemblage, les adresses μ , μ' , ν seront évidemment indexées par D et les étiquettes par I .

S'adressant à un programme d'assemblage on peut omettre de mentionner ces bases ; c'est ce que nous ferons désormais.

Examinons un problème un peu plus complexe qui prolonge en quelque sorte le précédent.

☐ DONNEES

S est une suite de chaînes $c_1, \ldots, c_p$ $(0 \leq p \leq 255)$ précédées de deux indications :

$$S = M_\sigma , \underbrace{m_{\sigma+2}}_{c_1} , \underbrace{m_{\sigma+3} , \ldots , \ldots , \ldots}_{c_p} m_{\gamma'-1}$$

p = nombre de chaînes dans S

γ' = adresse/D de la première case libre après S (2 octets)

Une chaîne c' suit immédiatement S en mémoire :

$$c' = \underbrace{m_{\gamma'}}_{= n'} , m_{\gamma'+1} , \ldots , m_{\gamma'+n'}$$

□

☐ PROBLEME

. Si c' est identique à un terme de s :

réponse (en ν) ← 1 ; place (en $\nu+1$) ← adresse/D du terme

Sinon :

réponse ← 0 ; place ← adresse de c'

Et c' est ajoutée à S .

(lors de l'analyse syntaxique d'un programme, les identificateurs déjà trouvé et examinés sont les termes de S - au début, $\gamma' = \sigma + 3$ et $p = 0$ - et c' est le dernier identificateur trouvé : il s'agit de le comparer aux précédents et de le leur adjoindre éventuellement).

□

PROGRAMME

	$r_1 \leftarrow \sigma$	$r_1 \leftarrow$ adr/D de S
	$r_2 \leftarrow M(r_1 + 3)$	$R_2 \leftarrow$ <adr/D de c_i, nb de c_i restants>
	$r_3 \leftarrow m(r_1 + 2)$	
uivant :	SI $r_3 = 0$ VERS ajouter	s'il n'y en a plus, ajouter c' à S
	$R_4G \leftarrow r_2$; $R_4D \leftarrow m(R_4G)$	$R_4 \leftarrow$ <adr/D de c_i ; long de c_i>
	$R_6G \leftarrow M(r_2)$; $R_6D \leftarrow m(R_6G)$	$R_6 \leftarrow$ <adr/D de c' ; long de c'>
	$T \leftarrow r_4D - r_6D$	T ← long c_i - long c' (r_5, r_7 inchangés) ; si $T \neq 0$ passer à $c_i + 1$
	SI $T \neq 0$ VERS préparer suivant	
eprise :	SI $R_4D = 0$ VERS test	. si les chaînes c_i et c' ne sont pas vides, comparer leurs queues.
	SINON $R_4G \leftarrow R_4G + 1$; $R_4D \leftarrow R_4D - 1$	.. si elles sont différentes, préparer la comparaison de c_{i+1} et de c'.
	SI $R_6D = 0$ VERS préparer suivant	.. si elles sont identiques, terminé (réponse ← OUI).
	SINON $R_6G \leftarrow R_6G + 1$; $R_6D \leftarrow R_6D - 1$	
	$r_8 \leftarrow m(r_4)$	
	$r_8 \leftarrow r_8 - m(r_6)$	
	SI $r_8 = 0$ VERS reprise	
	VERS préparer suivant	
test :	SI $R_6D \neq 0$ VERS préparer suivant	
	$m(\nu) \leftarrow 1$	
	VERS fin	
er		
ant :	$R_2G \leftarrow R_2G + m(R_2G) + 1$; $R_2D \leftarrow R_2D - 1$; VERS suivant	$R_2 \leftarrow$ < adr/D de c_{i+1} , nb de c_i restants> reprendre
outer :	$m(\nu) \leftarrow 0$	réponse ← NON
	$R_6G \leftarrow M(r_1)$; $R_6D \leftarrow m(R_6G$:	$R_6 \leftarrow$ < adr/D de c' ; long de c'>
	$M(r_1) \leftarrow R_6G + R_6D + 1$	adr/D de la 1ère place libre après S augmentée de long c' + 1 ; le nbe de termes dans S est augmenté de 1
	$M(r_1 + 2) \leftarrow M(r_1 + 2) + 1$	
fin :	$M(\nu + 1) \leftarrow r_2$:	place de l'occurrence identique à c' ← adrD de c_i (qui peut être c' elle même)

Cette analyse succincte de l'itération a mis en évidence un certain nombre d'opérations fondamentales encadrées propres à traiter des chaînes (tableaux complets) ou des suites de chaînes (analogues à des listes).

Pour simplifier, nous avons supposé que les éléments et les longueurs des chaînes étaient d'un octet. Il faudra donc envisager des variantes des instructions ci-dessus ; nous le ferons plus tard.

= : = : = : = : = : =

IV. L'APPEL DES PROCEDURES.

A. SON PRINCIPE : LA SUBSTITUTION

Les programmes de l'épisode précédent (comparer deux chaînes, chercher une chaîne dans une suite de chaînes) ne sont pas absolument indépendants ; le second d'entre eux contient une version du permier adaptée à l'emploi précis qu'on en fait (à savoir : répondre NON d'emblée quand les suites comparées n'ont pas même longueur, réduire la comparaison au dilemne IDENTITE/NON-IDENTITE, exploiter à deux fins différentes la non-identité).

S'agissant de programmes plus étendus que ceux-là, on pourrait penser que les quelques améliorations apportées par l'adaptation du premier pour l'insérer effectivement dans le second ne présentent guère d'intérêt et qu'on peut se contenter de l'y insérer formellement, c'est-à-dire d'y faire appel.

Nous allons décrire, en nous appuyant sur cet exemple, un procédé dont la généralité est évidente et qui réalise vraiment la SUBSTITUTION au cours de l'exécution des programmes.

Pour cela, nous appelons P_0 le programme principal (le second de l'épisode précédent) et P_1 le sous-programme (le premier cité).

Les zones de données qui leur sont attachées lors d'une exécution particulière ont l'aspect suivant.

ZD_0 (données de P_0)

adresses absolues	contenu	
$d_0 + \sigma$	γ' = adresse/d_0 de la 1ère case libre après la dernière chaîne dans S	suite S de chaînes
$d_0 + \sigma + 2$	p = nombre de chaînes dans S	
$\sigma + 3$	c_1 : 1ère chaîne dans S	
.....	cp : dernière chaîne dans S	

	____ γ'	chaîne c' (long c' + 1 octets)
	____ ν	réponse (1 octet)
	____ $\nu + 1$	adresse/d_o de l'occurrence de c' dans S initiale ou prolongée (2 octets)
	____ $\nu + 3$	rangement des 3 premiers registres (6 octets)
d_1 =	____ $\nu + 9$	(première case libre)
		ZD_1 (données de P_1)
	$d_1 + 0$	réservé - voir plus loin (2 octets)
	____ 2	$(d_1 - d_o)$ = long (ZD_o) (2 octets)
	____ 4	i_o = adresse absolue de P_o (2 octets)
	____ 6	48 = adresse/i_o de retour dans P_o (2 octets)
	____ 8	adresse/d_1 de la chaîne courante de S - donc $\gamma + d_o - d_1$ (2 octets)
	____ 10	adresse/d_1 de c' - donc $\gamma' + d_o - d_1$ (2 octets)
	____ 12	adresse/d_1 de REPONSE - donc $\nu + d_o - d_1$ (2 octe
d_2 =	____ 14	(première case libre)

LE PROGRAMME PRINCIPAL (P_o)

(Il s'exécute avec I = i_o et D = d_o)

adresses absolues	Instructions	effet
	Initialisation	
$i_o + 0$	$r_1 \leftarrow \sigma$	: $r_1 \leftarrow \sigma + D$
____ 4	$r_2 \leftarrow r_1 + 3$	: $r_2 \leftarrow \sigma + 3 + D$ = adr 1ère chaîne de S (s'il y en a ; sinon adresse de c')
____ 8	$r_3 \leftarrow m(r_1 + 2)$	: $r_3 \leftarrow$ nombre de chaînes dans S
____ 12	SI $r_3 = 0$ VERS ajouter	
(= suivant)		

appel de P_1 :

____ 16 $r_{29} \leftarrow d_1 - d_0$; $r_{30} \leftarrow i_0$; $r_{31} \leftarrow 44$ (adresse retour/i_0)

____ 20 Ranger les 3 premiers registres à partir de la case ($r_{29} - 6 + D$) (car r_1, r_2, r_3 sont réutilisés par P_i)

____ 24 Ranger les 3 derniers registres à partir de la case ($r_{29} + 2 + D$) (on les utilisera pour le retour)

____ 28 $M(8 + r_{29}) \leftarrow r_2 - r_{29}$: $M(8 + d_1) \leftarrow$ adr/d_1 de la chaîne de S

____ 32 $M(10 + r_{29}) \leftarrow M(r_1) - r_{29}$: $M(10 + d_1) \leftarrow$ adr/d_1 de la chaîne c'

____ 36 $M(12 + r_{29}) \leftarrow \nu - r_{29}$: $M(12 + d_1) \leftarrow$ adr/d_1 de REPONSE

____ 40 $D \leftarrow D + r_{29}$; $I \leftarrow i_1$; VERS 0 (i.e. $0 + I = i_1$) : $D \leftarrow d_1$; $i \leftarrow$ adr $P_1 = i_1$; VERS 1ère instruction de P_1

modification de P_1 incorporée à P_0 :

____ 44 Rétablir les 3 premiers registres à partir de la case ($r_{29} - 6 + D$)

____ 48 $r_4 \leftarrow m(\nu)$; $r_1 \leftarrow$ REPONSE (-1, 0, 1 pour pp, eg, pg)

____ 52 SI $r_4 \neq 0$ VERS préparer suivant

____ 56 $m(\nu) \leftarrow 1$

____ 60 VERS fin

fin de P_1 :

____ 64

= préparer suivant } comme dans la version primitive

etc...

LE SOUS-PROGRAMME (P_1)

(Il s'exécute avec $I = i_1$ et $D = d_1$)

Programme primitif

(dans lequel M8, M10, M12 remplacent μ, μ', ν)

$i_1 + 0$	$r_1 \leftarrow 0$
____ 4	$R_2G \leftarrow$ [M8] ; $R_2D \leftarrow m(R_2G)$
____ 8	$R_4G \leftarrow$ [M10] ; $R_4D \leftarrow m(R_4G)$
____ 12	SI $R_2D = 0$ VERS test SINON $R_2G \leftarrow R_2G + 1$ et $R_2D \leftarrow R_2D -$
= reprise	
____ 16	SI $R_4D = 0$ VERS pg SINON $R_4G \leftarrow R_4G + 1$ et $R_4D \leftarrow R_4D - 1$
____ 20	$r_6 \leftarrow m(r_2)$
____ 24	$r_6 \leftarrow r_6 - m(r_4)$
____ 28	SI $r_6D = 0$ VERS reprise
____ 32	SI $r_6G = 0$ VERS pp
____ 36	$r_1 \leftarrow 1$
= pg	
____ 40	VERS fin
____ 44	SI $R_4D = 0$ VERS fin
= test	
____ 48	$r_1 \leftarrow -1$
= pp	
____ 52	$m($[M12]$) \leftarrow r_1$
= fin	

retour au programme principal

____ 56	Rétablir les 3 derniers registres à partir de M2
____ 60	$D \leftarrow D - r_{29}$; $I \leftarrow$ R30 ; VERS R31

Quelques commentaires sur ce programme :

en $i_1 + 4$: $R_2G \leftarrow$ M8 ... i.e., sans abréviation : $R_2G \leftarrow M(8 + D)$ i.e. $R_2G \leftarrow M(8 + d_1)$ i.e. $R_2G \leftarrow$ adr/d_1 de la chaîne courante dans S . Alors :

$R_2D \leftarrow m(R_2G)$ i.e., sans abréviation : $R_2D \leftarrow m(R_2G + D)$
i.e. $R_2D \leftarrow m$ (adr absolue de la chaîne courante dans S,
i.e. $R_2D \leftarrow$ long de cette chaîne.

en $i_1 + 8$: $R_4G \leftarrow$ M10 ... i.e. $R_4G \leftarrow$ adr/d_1 de la chaîne c: ; etc...

en $i_1 + 52$: $m(M12) \leftarrow r_1$ i.e., sans abréviation : $m(M(12 + D) + D) \leftarrow r_2$
i.e. $m(M(12 + d_1) + d_1) \leftarrow r_1$
i.e. m ((adr/d_1 de REPONSE) $+ d_1$) $\leftarrow r_1$ i.e. $m(v + d_o) \leftarrow r_1$.

en $i_1 + 56$: $r_{29} \leftarrow M(2 + D)$; $r_{30} \leftarrow M(4 + D)$; $r_{31} \leftarrow M(6 + D)$
i.e. $r_{29} \leftarrow d_1 - d_o$; $r_{30} \leftarrow i_o$; $r_{31} \leftarrow 44$.

en $i_1 + 60$: $D \leftarrow D - (d_1 - d_o) = d_o$; $I \leftarrow i_o$; VERS $44 + I = 44 + i_o$.

Le reste semble aller de soi.

Cette façon de concevoir l'appel de sous-programmes fait apparaître un nouveau groupe d'instructions ad hoc qu'un compilateur n'aura pas de peine à utiliser. Les voici :

en $i_o + 16$: $r_{29} \leftarrow d_1 - d_o$; $r_{30} \leftarrow i_o$; $r_{31} \leftarrow 44$

Il suffit, pour se conformer à notre règle (au plus un opérande registre et un opérande mémoire par instruction) d'utiliser les deux premières cases de chaque zone de données (cases d'adresses $0 + D$ au moment où s'exécute le programme auquel la zone est associée) à stocker la longueur de cette zone ; ainsi l'instruction ci-dessus devient tout simplement :

$r_{29} \leftarrow M0$; $r_{30} \leftarrow I$; $r_{31} \leftarrow$ un renvoi

en $i_o + 20$, $i_o + 24$, $i_o + 44$, $i_1 + 56$:

{Ranger / Rétablir}	les	{n / Rn}	{premiers / derniers}	registres à partir de (une adresse)

en $i_o + 40$: $D \leftarrow D + r_{29}$; $I \leftarrow i_1$; VERS 0

Nous proposons de placer, en tête de chaque programme, après l'instruction de protection :

" VERS première instruction effective "

et avant cette première instruction effective, les constantes utilisées par le programme et les adresses absolues des sous-programmes utilisés (ces adresse seront fixées lors du chargement en mémoire des dits sous-programmes). Dès lors l'instruction ci-dessus devient :

$$\boxed{D \leftarrow D + r_{29} \ ; \ I \leftarrow \text{une adresse/I} \ ; \ \text{VERS } 0}$$

On peut aussi envisager des solutions plus raffinées faisant appel au chargement des programmes en mémoire. Nous verrons cela plus loin.

en $i_1 + 60$: $\boxed{D \leftarrow D - r_{29} \ ; \ I \leftarrow R30 \ ; \ \text{VERS } R31}$

en $i_0 + 28$, $i_0 + 32$, $i_0 + 36$, apparaissent des instructions dont l'effet est de transmettre des adresses en les modifiant aux fins d'emploi dans une prochaine zone de données. Dans chacune d'elles le registre 29 peut être spécialisé donc sous-entendu ; l'opérande mémoire μ est, suivant le cas, r_2, $M(r_1)$ ou ν . Restent à traiter les constantes telles que 8, 10 ou 12. Nous reviendrons là-dessus quand nous verrons d'autres cas analogues, lors de la transmission d'adresses comme résultats.

Ce mode d'appel s'applique naturellement aux procédures "récursives". Ainsi, on peut calculer $\sum_{p=1}^{n} p^i$ grâce à la définition suivante :

$$f(n, i) = (\text{SI } n = 0 \text{ ALORS } 0 \text{ SINON } f(n-1, i) + n^i)$$

qu'on peut traduire en ALGOL 60 par :

la procédure

```
ENTIER PROCEDURE f (n, i);
ENTIER  n, i ;
VALEUR  n, i ;
DEBUT
   ENTIER p, q ;
   SI n = 0 VERS e1 ;
   p := n - 1 ;
   q := f (p,i) ;
   p := n ↑ i ;
   q := p + q ;
   VERS e2 ;
e1 :  q := 0 ;
e2 :  RESULTAT := q
FIN
```

ou encore :

```
ENTIER PROCEDURE f (n, i) ;
ENTIER  n, i ;
DEBUT
    ENTIER n', i', p, q ;
    n' := n ; i' := i ;
    SI n' = 0 VERS e1 ;
    p := n' - 1 ;
    q := f (p, i') ;
    p := n' ↑ i' ;
    q := p + q ;
    VERS e2 ;
e1 :  q := 0 ;
e2 :  RESULTAT := q
FIN
```

La règle de recopie conduit à calculer $s := f(2,3)$ de la façon suivante :

$$
\text{f (2, 3) i.e.}\left\{\begin{array}{l}
\left.\begin{array}{l} n'_0 := 2 \\ i'_0 := 3 \\ \text{comme } n'_0 \neq 0 : \\ p_0 := n'_0 - 1 = 1 \end{array}\right] \text{①} \\
q_0 := f(p_0, i'_0) = f(1,3) \text{ i.e.} \left\{\begin{array}{l}
\left.\begin{array}{l} n'_1 := 1 \\ i'_1 := 3 \\ \text{comme } n'_1 \neq 0 : \\ p_1 := n'_1 - 1 = 0 \end{array}\right] \text{②} \\
q_1 := f(p_1, i'_1) = f(0,3) \text{ i.e.} \left\{\begin{array}{l}
\left.\begin{array}{l} n'_2 := 0 \\ i'_2 := 3 \\ \text{comme } n'_2 = 0 : \\ \\ \\ q_2 := 0 \\ q_1 := q_2 \end{array}\right] \text{③} \\
\left.\begin{array}{l} p_1 := n'_1 \uparrow i'_1 = 1^3 \\ q_1 := p_1 + q_1 = 0 + 1^3 \\ q_0 := q_1 = 1^3 \end{array}\right] \text{④}
\end{array}\right. \\
\left.\begin{array}{l} p_0 := n'_0 \uparrow i'_0 = 2^3 \\ q_0 := p_0 + q_0 = 1^3 + 2^3 \\ s := q_0 = 1^3 + 2^3 \end{array}\right] \text{⑤}
\end{array}\right.
$$

Les calculs s'effectuent dans l'ordre :

① , ② , ③ , ④ , ⑤ .

Un compilateur peut donner de cette procédure la traduction suivante :

i + 0	VERS e0	
4	adresse du programme = i	
6	$r_1 \leftarrow 18$	
= e0		$M(0 + d_J) \leftarrow 18$ = longueur de ZD_J
10	$M0 \leftarrow r_1$	
14	$r_1 \leftarrow M(M8)$	$M(8 + d_J) \leftarrow M(M(8 + d_J) + d_J) = n$ i.e. VALEUR n
18	$M8 \leftarrow r_1$	
22	$r_1 \leftarrow M(M10)$	VALEUR i
26	$M10 \leftarrow r_1$	
30	$r_1 \leftarrow M8$	SI n = 0 VERS e1
34	SI $r_1 = 0$ VERS e1	
38	$r_1 \leftarrow r_1 - 1$	$p \leftarrow n - 1$
42	$M14 \leftarrow r_1$	

46	R29 ← M0 ; R30 ← 1 ; R31 ← 70	R29 ← long ZD_J = 18 ; R30 ← 1 ; R31 ← adr retour/1
50	Ranger R29 à 31 en M (R29 + 2 + D) sqq	ranger dans les cases $2 + d_{J+1}$ sqq
54	M (8 + R29 ← 14 - R29	$M(8 + 18 + d_J = 8 + d_{J+1}) \leftarrow$ adr $p/d_{J+1} = -4$

58 M (10 + R29) ← 10 - R29 } M (10 + d_{J+1}) ← adr i/d_{J+1} = - 8
62 M (12 + R29) ← 16 - R29 } M (12 + d_{J+1}) ← adr résultat/d_{J+1} = - 2
66 D ← D + M0 ; I ← M4 ; VERS 0 } D ← d_J + 18 = d_{J+1} ; I ← M (4 + i) = i ;
VERS i + 0

---

70 r_1 ← M8
74 r_1 ← r_1 ↑ M10 } q ← n ↑ i
78 M14 ← r_1
82 r_1 ← r_1 + M16 } q ← p + q
86 M16 ← r_1
90 VERS e2
94 r_1 ← 0
= e1 } q ← 0
98 M16 ← r_1
102 r_i ← M16
= e2 } résultat ← q
106 M (M12) ← r_1
110 R29 à 31 ← M2 sqq } les registres R29 à 31 sont rétablis
114 D ← D - R29 ; I ← R30 ; VERS R31 { D ← d_{J+1} - (d_{J+1} - d_J) = d_J ;
I ← I ;
VERS i + 70

Chaque zone de donnée correspondant à une exécution de ce programme se présente ainsi (en supposant qu'on calcule sur des entiers représentés dans deux cases successives)

ZD_J :

d_J + 0 : longueur de ZD_J = 18
+ 2 : longueur de ZD_{J-1}
+ 4 : adresse du programme appelant = i_{J-1}
+ 6 : retour dans le programme appelant /i_{J-1}
+ 8 : adresse de n/d_J PUIS n
+ 10 : adresse de i/d_J PUIS i
+ 12 : adresse du résultat/d_J
+ 14 : p
+ 16 : q
+ 18 : première case libre

Dans un certain programme initial, on a par exemple :

$$D = 100 \quad , \quad M(20 + D) = 2 \quad , \quad M(22 + D) = 3$$

et on veut effectuer l'opération

$$M(30 + D) \leftarrow f(2, 3)$$

Ce premier appel crée une zone de données allant par exemple de la case 200 à la case 217 et place en 202, 204 et 206 la valeur 100, l'adresse du programme appelant et le retour dans ce programme puis en 208, 210, 210 les adresses des deux opérandes et du résultat (i.e. - 80, - 78, - 70) et ainsi de suite. Les modifications de la mémoire peuvent être résumées de la façon suivante :

ordre des modif / adresses	0 appel	1	appel	2	appel	3	4	5
200		18						
2	100							
4	adr p. a.							
6	retour p.a.							
8	- 80	$n_o = 2$						
10	- 78	$i_o = 3$						
12	- 70							
14		$p_o = 1$						$1^3 + 2^3$
16							1^3	$1^3 + 2^3$
18				18				
20			18					
22			1					
24			70					
26			adr n_1 = p4	$n_1 = 1$				
28			adr i_1 = -8	$i_1 = 3$				
30			adr res_1 = -2					
32				$p_1 = 0$			1^3	
34							1^3	
36						18		
38					18			
40					1			
42					70			
44					adr n_2 = -4	$n_2 = 0$		
46					adr i_2 = -8	$i_2 = 3$		
48					adr res_2 = -2			
50								
52						$q_2 = 0$		
D =	200		218		236		218	200

et finalement :

$$M\,(-70 + D = -70 + 200 = 30 + 100 = 30 + d\ \text{initial}) \leftarrow 1^3 + 2^3 .$$

C'est l'ordre même des calculs prévu par l'application de la règle de recopie.

à suivre

= : = : = : = : = : = : =

AUTOMATE PROGRAMMABLE

(2ème partie)

J.C. FUMANAL

Lors de la première partie, nous avons abordé le formalisme des réseaux de petri (RDP) et défini les principes généraux de la description d'un automatisme à partir de ses caractéristiques fonctionnelles (début du paragraphe VI).

Nous allons maintenant, après avoir précisé le choix de la représentation, définir les idées directrices de la réalisation.

1. Description du réseau.

La position des marques étant représentative de l'état de la machine simulée, la description du RDP qui lui est associée doit en mentionner les chemins et les conditions d'évolution.

Les noeuds du réseau peuvent être utilisés comme base de représentation. Dans ce cas, la description aura l'un des aspects suivants :

* représentation centrée sur les transitions.

A chaque transition est associée une fonction ΦE décrivant les places sources, l'évènement lié à la transition et les places d'arrivée. Φ_E constitue la "phrase" associée à la transition t_n ; conventionnellement nous écrirons $t_n : < \Phi_{E_n} >$, la notation $< \Phi_{E_n} >$ correspondant à l'expression de Φ_{E_n} selon le langage de l'automate.

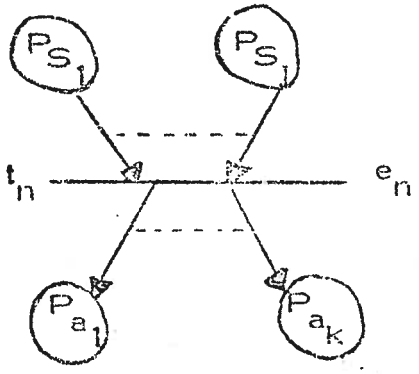


figure 3

Dans le cas de la figure 3, où $P_{S_1} \ldots P_{S_j}$ sont les places amont et $P_{a_1} \ldots P_{a_k}$ les places aval, on a :

$$t_n : < P_{S_1} \ldots P_{S_j} > < e_n > < P_{a_1} \ldots P_{a_k} > .$$

e_n est l'évènement permettant de valider la transition. La description de l'ensemble des transitions permet d'obtenir une structure de données suffisante pour la gestion des marques.

* représentation centrée sur les places.

La fonction associée à chaque place décrit toutes les transitions de sortie de celle-ci. Par analogie au cas précédent, on appellera φ_e la définition d'une transition de sortie qui doit mentionner les autres places sources, l'évènement et les places aval.

Dans le cas de la figure 3 la phrase φ_e associée à P_{S_1} est de la forme :

$$P_{S_1} : < P_{S_2} \ldots P_{S_j} > < e_n > < P_{a_1} \ldots P_{a_k} > .$$

Il y a autant de phrases φ_e que de transitions de sortie de la place. Pour m transitions de sortie d'une place P_{S_i}, la fonction qui lui est associée peut s'écrire :

$$P_{S_i} : < \varphi_{e_1} > \ldots < \varphi_{e_m} > .$$

L'optimisation de l'encombrement mémoire d'une telle représentation est liée à la description unique d'une transition partagée entre plusieurs places source Dans l'exemple de la figure 3, la transition t_n sera décrite J fois : une pour chaque place source pour laquelle elle constitue une transition de sortie. Le choix de la place de description, appelée encore place pivot, dépend du réseau et du logiciel de l'automate.

La gestion des marques est obtenue par l'analyse des transitions de sortie des places marquées.

2. Déclaration des sorties.

A chaque place est associée une phrase Φ_S précisant les conditions d'activité des sorties sur la place. Lorsque le franchissement d'une transition doit provoquer une modification des sorties , celles-ci seront déclarées au niveau des fonctions d'évolution φ_e ou Φ_E .

Dans la phrase de traitement, seules les phrases Φ_S associées aux places marquées sont examinées.

VII. ELABORATION DU LOGICIEL DE L'AUTOMATE : LES THEMES DIRECTEURS.

1. Les configurations analysables.

D'une part, l'automate doit pouvoir traiter les convergences et les divergences d'arcs à partir des noeuds du réseau qui correspondent aux deux formes suivantes :

transition place

figure 4

D'autre part, les arcs, constituant les chemins qu'empruntent les marques, pourront être affectés d'une capacité de transfert.

Enfin, les variables liées aux transitions (évènement) agissent par les niveaux logiques "0" et "1" . Une transition peut être franchie à chaque réalisation de l'évènement qui lui est associé.

2. Modélisation.

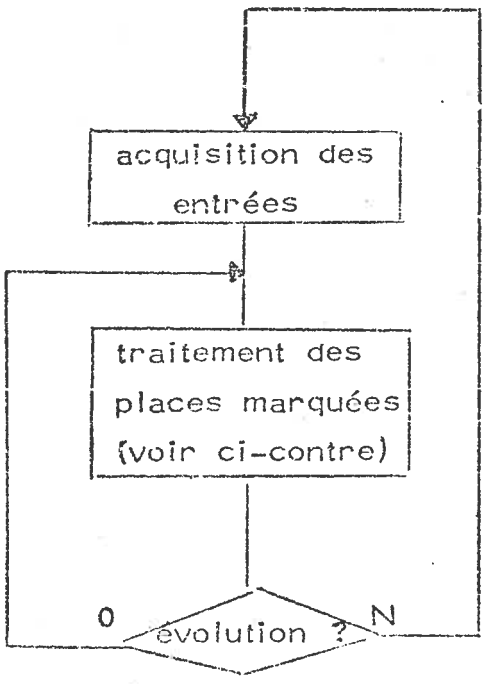
Elle est basée sur la représentation centrée sur les places qui s'adapte très bien aux RDP représentés sous forme graphique. A chaque place est associée la phrase contenant les relations d'évolution et les déclarations des sorties :

$$P_J : < \Phi_{E_J} > < \Phi_{S_J} > .$$

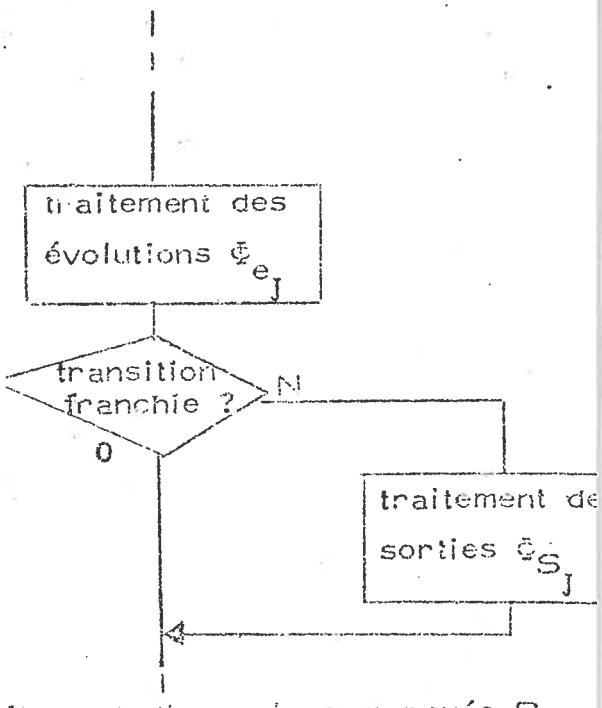
Au stade du traitement, l'automate examine les phrases associées aux places marquées.

3. Simulation synchrone.

L'analyse du réseau et le déplacement des marques correspondent à une procédure séquentielle dont le temps d'exécution ne peut être négligé mais qui ne doit en aucune façon intervenir sur l'évolution des marques. Il est alors nécessaire que le contexte extérieur paraisse figé pendant ce temps là et l'acquisition des entrées est faite lorsque le marquage du réseau est stabilisé. La notion de temps réel peut être considérée comme respectée si le contexte extérieur (le processus) a un temps d'évolution très supérieur à la période d'échantillonnage des entrées, ce qui est le cas de bien des systèmes industriels.
Le programme d'analyse peut être défini par les organigrammes suivants :



Réseau



traitement d'une place marquée P_J

Figure 5

Pour chaque transition de sortie franchie, le déplacement des marques se fait en deux temps :

- prélèvement du marquage amont,
- marquage des places aval.

Le nouveau marquage du réseau n'est considéré que lorsque toutes les transitions franchies d'une place ont été traitées.

Le traitement des sorties est obligatoirement réalisé avant l'échantillonnage des entrées, une transition validée ne pouvant être franchie qu'une fois.

4. Franchissement des transitions.

Compte tenu de l'exploration cyclique des places marquées, il importe qu'une transition validée ne soit franchie qu'une fois. En effet, si le marquage des places sources le permet, lors du cycle d'exploration suivant, il y aura à nouveau franchissement. C'est par exemple le cas de marquages multiples sur une place, ou aussi plus simplement le cas de la figure 6 :

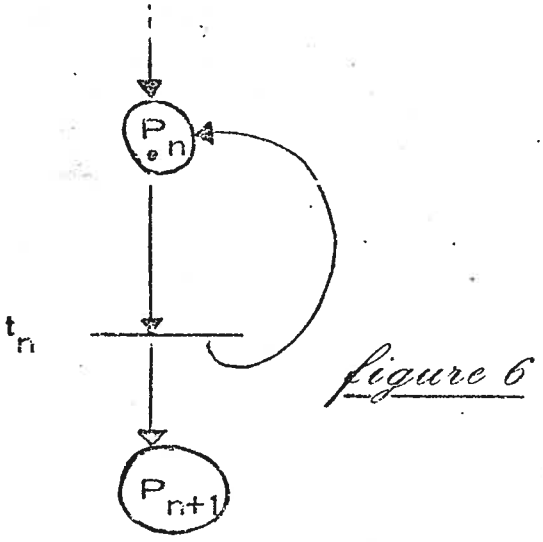


figure 6

Piège à marque

L'évènement t_n étant réalisé, d'après les organigrammes de la figure 5, à chaque examen de la place P_n il y aura à nouveau franchissement et dépôt d'une marque dans la place P_{n+1} .

Il faut créer une procédure telle qu'une transition validée et franchie ne soit pas franchie de nouveau au cycle suivant si le contexte le permet encore.

Dans le cas où plusieurs transitions de sorties d'une place sont simultanément franchies, c'est le contexte après franchissement de toutes les transitions qui doit être analysé.

5. Langage et communication.

Les données sont introduites selon le formalisme classique de l'écriture des équations booléennes, notamment les opérateurs "+" (OU), "." (ET), "( )" mise en facteur d'expression logique. La longueur et le nombre d'équations peuvent être quelconques dans une phrase

6. Traitement et analyse des phrases.

L'interprétation est imposée par la faible capacité de mémoire disponible. Chaque caractère dans une équation est examiné syntaxiquement. L'analyse sémantique et le traitement de la variable sont réalisés sur l'opérateur fermant.

Le traitement d'une équation d'évolution est arrêté si l'état logique du premier membre ne permet pas la validation (relation d'incidence).

Par contre, pour des équations configurant les sorties, les variables déclarées dans le deuxième membre suivent l'état logique du premier membre.

7. Temporisateur.

La gestion d'automatisme faisant appel très souvent à des temporisateurs, il est nécessaire d'introduire cette fonction.

Ces temporisateurs "logiciels" sont nécessairement gérés par un programme d'interruption associé à une horloge temps réel.

Deux fonctions de temporisations sont définissables :

T : temporisateur simple : la transition n'est franchie qu'une fois si elle est de nouveau franchissable.

/ T : temporisateur réarmable : la transition sera franchie, tous les intervalles de temps précisés, si elle est de nouveau franchissable.

L'exemple des figures 7 et 8 permet de mieux comprendre la différence :

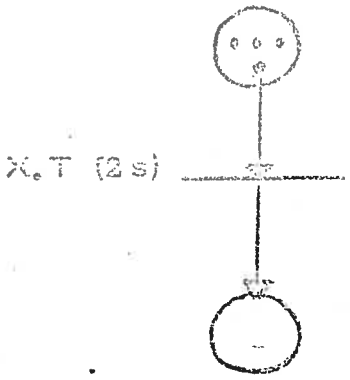


figure 7

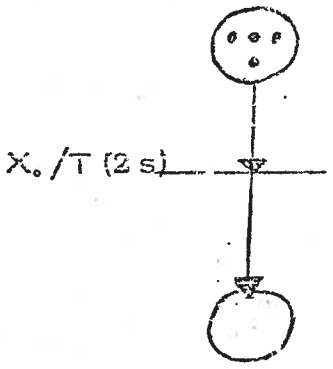


figure 8

Dans les deux cas il faut, pour valider la transition, que X soit au niveau logique 1. Si cette condition est réalisée :

Figure 7 : il y aura un transfert d'une marque au bout de 2 secondes.

Figure 8 : il y aura un transfert toutes les 2 secondes tant que X = 1.

Pour la figure 6, si l'évènement t_n est un temporisateur réarmable, nous avons un générateur périodique de marques.

Dans le cas général, selon la procédure d'interprétation du langage, un temporisateur est activé ou maintenu actif si il est analysé. Pour cela, deux conditions sont nécessaires :

- l'évènement qui le précède dans la phrase est réalisé,
- la phrase correspond à une place marquée.

Une temporisation peut donc être effectuée par rapport à des états de marquage et de variables.

8. Réceptivité sur les transitions et sur les places.

Elle peut s'effectuer par rapport à des états logiques de variables d'entrée et de sortie, de l'état de marquage de place sans démarquage, de temporisateur.

9. Messages d'erreur.

Ils doivent atteindre principalement les objectifs suivants :

- permettre à l'utilisateur d'identifier et de localiser rapidement les phrases incorrectes ;
- signaler des dépassements de capacité de l'automate ;
- détecter des structures de réseau incorrectes, notamment les conflits de transitions pour les réseaux, sauf ceux auxquels se rattache le grafcet.

En effet, pour la figure ci-dessous, si l'évènement T est réalisé, il n'y a pas de conflit pour le cas du grafcet, alors que, pour un RDP, le franchissement simultané des deux transitions demanderait une deuxième marq sur la place P_i .

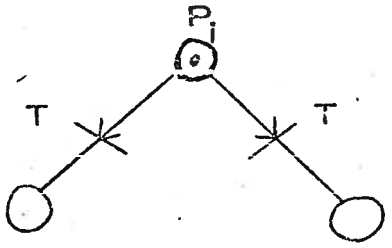


figure 9

La détection d'une erreur doit se traduire par l'émission d'un message en indiquant l'origine et la cause.

VIII. REALISATION MATERIELLE DE L'AUTOMATE.

1. La carte microprocesseur.

Il n'entre pas dans le cadre de cet article de prétendre déterminer les critères de choix d'un microprocesseur. La formation de l'utilisateur et le contexte de son environnement, disponibilité d'outils de développement par exemple, constituent en général des éléments déterminants dans la sélection d'un produit.

Il était impératif, par contre, que, la réalisation étant avant tout un outil d'enseignement, elle soit simple à mettre en oeuvre et d'un prix de revi peu élevé.

La carte TM 990/189 de Texas Instrument, dite encore carte "Université", a été choisie pour les aspects suivants :

- prix de revient modique,
- liaison RS232 prévue sur la carte (il suffit de monter les éléments),
- support pouvant recevoir sans modification une EPROM de 2K octets,
- capacité de mémoire RAM de 2K octets,
- 21 lignes d'entrée/sortie disponibles.

2. Caractéristiques des entrées/sorties de l'automate.

Tenant compte du choix précédent, l'automate comporte 8 entrées (X0 à X7) et 8 sorties (Y0 à Y7) qui correspondent respectivement aux lignes P8 à P15 et P0 à P7 du connecteur P2 (niveaux TTL).

3. Modification et environnement.

Le logiciel de l'automate est dans une EPROM de 2K 0 qu'il suffit d'enficher sur son support. La ROM du moniteur (4K 0) doit être en place. Il faut compléter la carte en mémoire RAM et câbler les éléments permettant de réaliser l'interface RS232. Toutes ces opérations sont matériellement prévues sur la carte et décrites dans le manuel d'utilisation. Un terminal et un câble de liaison permettent de dialoguer avec l'automate (RS232, V24).

IX. REALISATION LOGICIELLE DE L'AUTOMATE.

Le logiciel de l'automate comporte deux parties :

1. L'éditeur (700 octets environ).

Il permet à l'utilisateur de dialoguer aisément avec l'automate :

- introduction, modification, annulation, impression des phrases et du marquage ;
- départ, arrêt, reprise de la simulation.

L'éditeur gère la zone mémoire de phrase selon les principes utilisés sur les disques souples. L'encombrement des informations est toujours optimisé quelques soient les corrections apportées par l'utilisateur.

2. L'interpréteur (1400 octets environ).

Basé sur les thèmes directeurs abordés précédemment, il accepte jusqu'à 9 marques par place. La validation d'une transition peut être réalisée sur des combinaisons de variables d'entrée ou de sortie, d'état de marquage de places avec ou sans prélèvement de marques, de temporisateurs simples ou réarmables.

La capacité de transfert des arcs peut aller de 0 à 9 marques.

L'activité des sorties sur une place peut être également effectuée à partir de combinaisons de ces paramètres.

A suivre...

3ème partie : Utilisation - Exemples.

= : = : = : = : = : =

Souvenirs ...

Patrick ISOARDI

Ces derniers temps, j'ai eu quelque envie ... envie de gens sérieux bien entendu : ressortir mes vieux jouets. Ces jouets, ce sont les outils décrits dans ma thèse de 3ème cycle en 1978 : Structure de Commutation (*). Ils m'avaient tant amusé par le passé, que j'ai peine à croire qu'ils subissent comme nous le poids des ans.

Pour renouer avec ce passé mais aussi peut-être pour renforcer ma conviction, j'ai décidé de réaliser quelques petits montages simples et de les regarder... tout simplement les regarder.
Peut-être cela vous amusera t-il de jouer avec moi.

La commutation contenue dans un programme peut être entièrement décrite à partir des instructions bien connues :

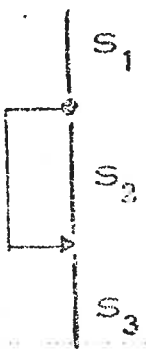
Si < condition > vers e ; et vers e ;

Cette commutation peut-être mise en évidence en l'accentuant dans son environnement et dont les graphes suivants en sont une représentaation.

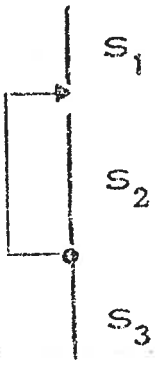
type A

Si < condition > vers e ;

e plus bas dans le programme



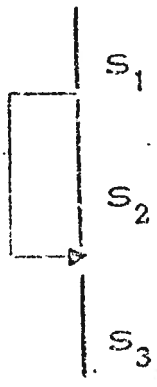
e plus haut



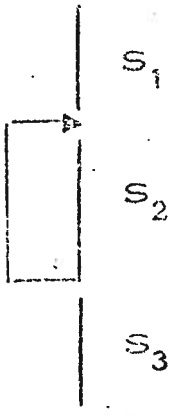
type B

vers e ;

e plus bas



e plus haut

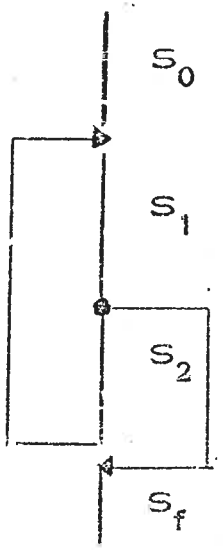


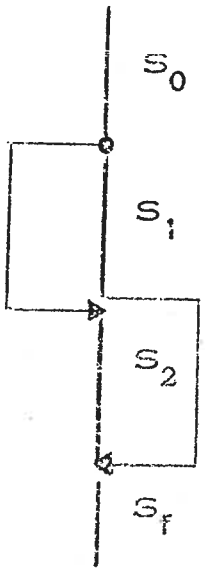
où les S_i représentent des séquences du programme.

Si je considère la notion de graphe de programme définie dans ma thèse, les graphes du type A sont sémantiquement cohérents. Ce n'est pas le cas des schémas de type B ; ils n'auront de sens qu'à condition que l'on puisse atteindre respectivement S_2 et S_3 d'une manière tôt ou tard conditionnelle.
C'est là que le jeu commence.

Considérez les quatre schémas précédents comme des objets élémentaires, comme axiomes en quelque sorte et réalisez toutes les constructions possibles qui mettent en présence un graphe de type A et un graphe de type B .

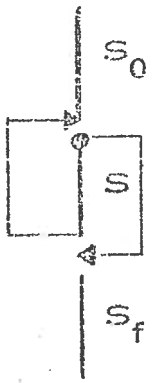
Parmi tous les montages, vous obtiendrez essentiellement les produits cohérents suivants :





Partant de là, si vous cherchez tous les cas particuliers, vous retrouveriez très facilement les deux graphes du type A ainsi qu'un intrus. Lequel ?

Le voici avec son compère :



Plaçons nous dans le cadre très précis où les S_i sont des séquences de programme ne comportant pas d'instruction de commutation et nous sommes en présence d'objets que beaucoup appellent boucles et dont les supports [cf. *] sont respectivement les suivants :

$$S_0 < \{\circ >\ ,\ S_f\} \qquad \text{et} \qquad S_0 < S\{\ >\ ,\ S_f\}$$

et les traces [cf. *]

$$S_0\ (S)^i\ S_f \quad \forall\ i \in \mathbb{N} \qquad \text{et} \qquad S_0\ (S)^i\ S_f \quad \forall\ i \in \mathbb{N}^*$$

Le chemin [cf. *] $S_0 S_f$ n'existe que dans le premier graphe de programme puisque là i peut y prendre la valeur nulle. Curieux, non ! Certains seraient alors tentés d'affirmer que la première "boucle" est plus puissante que la seconde. Là encore, la notion de puissance n'a de sens que si l'on précise les critères sur lesquels sont fondés ce jugement.

Mais au fait : qu'est-ce qu'une boucle ? et... où est la boucle ?

Attention, pas d'énervement, ce n'est qu'un jeu !
Voici un amusement qui pousse à la reflexion. N'est-ce pas en quelque sorte une certaine forme de recherche ?

P.S. Contrairement à certaines revues, vous ne trouverez pas la solution à la fin de ce bulletin, ni dans les prochains, mais ... peu-être au détour d'un chemin [cf. *]

Autojection et Compilateurs

E. BIANCO

C.R Subject classification informatics 4.10 4.11 4.12 4.31 4.32 4.42 5.24 5.26

Résumé. Cet article vise à montrer l'application de résultats théoriques fondamentaux à la structure des compilateurs, et à la fois à définir des moyens pratiques de les construire. Les produits obtenus : des compilateurs autojectifs sont prêts à être insérés dans des systèmes dont la structure sera en conséquence considérablement simplifiée. Sont utilisés les résultats d'une théorie où l'on a complètement séparé les notions afférentes aux compilateurs et celles afférentes aux systèmes. L'exposé montre, exemples à l'appui, l'intérêt que présente la mise en oeuvre de notions fondamentales ainsi dégagées.

= : = : = : = : = : = : =

AUTOJECTION ET COMPILATEURS

INTRODUCTION.

Ce que j'aimerais exposer dans les lignes qui suivent, c'est une description, un fil en quelque sorte, qui permette d'arriver à la construction d'un compilateur, par un chemin où les choses puissent apparaître simples. Bien sûr, un compilateur n'est pas un objet simple, mais en se donnant une sorte de systématique, on en réalise l'approche en économisant bien des souffrances.

La notion de compilateur est déjà fort ancienne (20 ans et plus c'est beaucoup en informatique), mais je voudrais aller plus loin, et replacer cette notion dans son contexte informatique. C'est pourquoi, je vais introduire cette notion en même temps que celle de système informatique, et en la détachant bien de cette dernière.

Je vais pousser la dissociation de ces deux notions aussi radicalement que possible. Mais, en contrepartie, je vais être amené à décrire d'autant plus complètement le protocole d'échange entre ces deux organes. En passant d'ailleurs je soulignerai quelques propriétés importantes des algorithmes et, pour cela, je ferai référence à l'ouvrage cité en [1].

En bref, dans un ordinateur qui tourne, on trouve un ensemble composite de compilateurs d'analyse et de déroulement, avec un système ou moniteur, matériel de base pour assurer le service exigé par l'utilisateur. En fait, tout cela doit permettre d'aboutir au déroulement de tâches plus ou moins complexes, sous forme de programmes pris en charge avec plus ou moins de facilités. Entendons par là que, selon le degré de sûreté que possède le programme que l'on veut faire dérouler, on assurera un contrôle qui ira du minimum normal au maximum qui est l'obtention d'une trace complète.

Le minimum normal consiste en la vérification sémantique réalisée au déroulement, et qui dépend essentiellement des propriétés du langage pris en compte. Par exemple en Algol 60 le compilateur de déroulement vérifie qu'une variable indicée se trouve bien prise dans le tableau auquel elle appartient.

Le maximum, lui, devrait donner tous les détails du calcul au fur et à mesure qu'il se déroule. Bien entendu ceci est très coûteux et ne sert qu'aux tous premiers pas de la mise au point.

Pour la réalisation de tout ceci, on dispose globalement de deux méthodes : l'une, de loin la plus répandue, est celle des interruptions ; l'autre, qui reste pour l'instant objet de laboratoire, n'a pas à mon avis dit son dernier mot. Très grossièrement on peut se faire une idée des avantages et inconvénients de ces deux méthodes. La première est plus rapide d'application, car les compilateurs n'ont pas à être structurés spécialement ; le système les interrompt de force, par contre il est obligé de prendre des précautions de sauvegarde et de rétablissement pour le travail dans lequel il intervient et qu'il faudra reprendre. Très généralement tout processus prioritaire se manifeste en interrompant ce qui se déroule. C'est le cas pour les échanges, par exemple. On a vu des systèmes ainsi bloqués par des trains d'interruptions suffisamment rapprochés pour dépasser la capacité de traitement de la machine.
Dans l'autre méthode, les travaux du système et des compilateurs sont parfaitement séparés, les compilateurs sont préstructurés, ce qui signifie que le travail du système se réduit à la seule gestion des tâches, aussi bien de traitement que d'échanges. Le système fonctionne comme un programme dans lequel s'insèrent des sortes de procédures dont le temps de déroulement est strictement borné. Le travail complet de la tâche étant réalisé par l'insertion d'autant de ces procédures qu'il est nécessaire. Le système ne peut plus être bloqué par un afflux quelconque d'information puisqu'il demeure toujours maître de la situation.

Raisons de l'état de fait.

Il est un état de fait qu'il faut bien constater, c'est la structure des ordinateurs tels que nous les connaissons depuis Von Neumann. Dans l'ouvrage cité en [2], je fais une étude qui recadre la théorie de cette forme d'ordinateurs par rapport à d'autres théories : les fonctions récursives, la machine de Turing, les systèmes de Post, etc...

Ce que j'essaie de mettre en évidence, c'est un chemin possible qui relie une construction comme la machine de Turing, par exemple, avec un ordinateur standard, en passant par plusieurs constructions intermédiaires, que l'on obtient à partir de la précédente par adjonction d'un organe supplémentaire.

C'est ainsi que, partant de la machine de Turing, on crée la machine à cases adressables (cf. [3]), en coupant un morceau de ruban que l'on baptise mémoire centrale, et qui a pour propriété essentielle d'avoir une longueur bornée, de contenir un nombre borné de cases. C'est justement cette propriété qui permet d'introduire celle d'adresse en un premier temps. Dans un second temps, elle permet d'introduire celle d'index, qui devient d'ailleurs indispensable pour retrouver la notion de procédure qui existait normalement dans la machine de Turing, et qui disparait dans la machine à cases adressables.

Ceci appelle à quelques commentaires. Un algorithme donné de la machine de Turing peut s'appliquer sur autant de configurations compatibles qu'on désire, et qui se trouvent sur le ruban. Il suffit, pour commencer le calcul, que la tête de lecture soit positionnée au bon endroit de la configuration choisie. Passer d'une configuration à une autre n'exige qu'un changement de place de la tête de lecture. L'algorithme demeure inchangé. C'est en cela que je dis : la machine de Turing contient la notion de procédure.

Ce n'est plus vrai pour la machine à cases adressables. En effet, il est facile de constater que changer de configuration en mémoire centrale exige de modifier l'algorithme, au moins pour ce qui est de la désignation des opérandes.

Mais alors, que permet de gagner la machine à cases adressables ? Quelque chose d'important tout de même : imaginons une machine de Turing qui travaille sur deux configurations à la fois et imaginons la manière dont on doit programmer l'aller-retour incessant de l'une à l'autre.

La machine à cases adressables contient d'ailleurs ce qu'il faut pour recevoir le perfectionnement qui va permettre de retrouver notre fameuse notion de procédure.

On s'arrange pour que le nombre maximum de cases de la mémoire centrale puisse être contenu dans une case spécifique d'abord, et puis dans une case quelconque ensuite. Et on vient d'inventer l'index, et en outre la possibilité de mémoriser l'index. L'adresse devient calculable.

On en arrive ainsi à la procédure formelle, langage de la machine formelle qui est un véritable ordinateur complet.

Nous arrivons là au coeur du problème. Les ordinateurs sont ce qu'ils sont. Et c'est la conséquence d'un ensemble complexe d'efforts dans lesquels interviennent des choses aussi disparates que de l'économie, de la technologie, des nécessités commerciales, des besoins d'ordre scientifique, des nécessités de structuration sociale, de la progression hiérarchique et toutes autres conditions qu'il est possible d'imaginer. En plus, une certaine inertie compréhensible a réussi à figer dans une certaine forme la notion d'ordinateur, qui, du coup, apparaît comme naturelle. Et c'est finalement cette conséquence, dont rien ne prouve pour l'instant qu'elle n'est pas strictement liée à ces conditions, que nous allons tenter de prendre en charge dans les lignes qui suivent.

L'AUTOJECTION, PROPRIETE LOGICIELLE.

Je vais donc essayer de décrire une structure de logiciel qui permette une description complète du système informatique. Pour arriver à ce résultat il faudra disposer d'un ensemble complexe de moyens, incluant un langage de programmation adapté, et des jeux de notions trouvant leur description naturelle dans le langage. Je ne me préoccuperai, dans le présent article, que des notions qui vont apparaître dans le compilateur.

Je pars d'idées simples : je vais faire en sorte que tout programme soit pré-structuré. Ayant subdivisé le champ informatique en deux domaines, celui du fini borné et celui du fini illimité, je spécialise mes deux fonctions fondamentales, chacune dans l'un de ces domaines : le compilateur travaille dans le fini borné, le système, lui, prend en charge le fini illimité.

Le compilateur devient une sorte de procédure qui s'insère normalement dans le système. Le système fournit les données et récupère les résultats de la compilation, et il prend en charge tous les côtés conversationnels et interactifs. Le compilateur, lui, compile la phrase symbolique donnée par le système, et rend un programme compilé. Il signale au passage tous incidents qui obligent à des décisions qui ne lui appartiennent pas, telles que débordements, erreurs syntaxiques ou autres. S'il s'agit d'un compilateur de déroulement; la vérification syntaxique devient pratiquement inexistante, mais nombre de vérifications sémantiques sont possibles.

Je reprends l'exemple cité plus haut de la constatation qu'un élément de tableau est dans le tableau ou non, chose qu'on ne sait qu'au déroulement. Je vais établir les notions qui me sont nécessaires pour faire d'un compilateur un outil commode à utiliser pour cet usage. Et je vais montrer l'implication de leur utilisation dans la construction d'un tel compilateur, que j'appellerai : compilateur autojectif.
Je donnerai ensuite une technique pour construire des compilateurs dont on puisse être sûr qu'ils sont autojectifs. Je les appellerai compilateurs compacts et leur définition est en même temps une technique constructive.

DEFINITION DE L'AUTOJECTIVITE.

Je considère un algorithme avec une seule entrée et une seule sortie. On peut toujours se ramener à ce cas. Et j'appelle chemin, la suite des instructions prises en compte dans un seul parcours allant du début jusqu'à la fin. Un même algorithme peut donc recouvrir plusieurs chemins différents. L'algorithme étant écrit, je dirai qu'il est autojectif si, et seulement si, ses chemins sont finis et bornés dans le temps.
De plus, il sera, en général, muni de commutation globale.

En conséquence, cela signifie, soit que les chemins ne contiennent pas de boucles, soit qu'ils contiennent des boucles dont le nombre de tours est borné à l'avance.

Alors se pose la question de savoir comment ramener un algorithme quelconque à une forme autojective. De fait, comment prendre en compte la boucle à priori illimitée.

Cela se fait tout simplement en ouvrant celle-ci pour la faire se refermer dans le système. Donc, de la munir de commutation globale.

En conséquence, on voit transparaître une partie des relations qui interviennent dans la liaison fini-borné et fini-illimité. Relations qui sont matérialisées entre système et compilateurs, et qui servent à gérer l'approvisionnement de la mémoire centrale à partir des files externes.
D'un point de vue pratique, j'aimerais souligner les commodités ainsi introduites.

De l'extérieur, le système peut facilement compter les tours de boucles, puisqu'il contrôle la commutation globale. Il n'a pas à se préoccuper, comme dans le cas de l'utilisation des interruptions, de protéger ce qu'il vient d'interrompre.

De l'intérieur les chemins peuvent être arrangés au mieux de l'utilisation. Ce qui implique une étude poussée de la structuration des algorithmes Etude qui est facilitée toutefois par le fait que l'ouverture du système de boucles est systématisé. Il ne reste à calculer que la longueur du chemin pour l'adapter à la classe du problème. Un calcul plus strict est exigé pour du temps réel que pour du temps partagé, par exemple.

Je vais maintenant définir le compilateur compact et puis j'appliquerai cette définition à un langage de programmation simple à titre d'illustration. Plus exactement, je vais définir un jeu complet de langages qui permette de programmer, de créer des données, et de les exploiter, et enfin d'appliquer des programmes sur des données.

COMPILATEUR COMPACT.

La définition que je donne de ce genre de construction est aussi et surtout une méthode de structuration. Je vais donner un aspect synthétique de cette définition, et montrer comment l'appliquer sur un exemple. Au préalable remarquons très simplement que, quelque soit le langage qu'on utilise, ce langage sera toujours construit sur un alphabet fini.

Partant de là, je considère que le quantum normal de travail va coïncider avec le traitement qui correspond à la prise en charge d'une nouvelle lettre dans une phrase symbolique. Je me donne donc le traitement de la lettre comme élément de travail à prendre en considération dans la structure du compilateur.

Je vais alors me donner les moyens de procéder ainsi. Les conditions sont les suivantes : il faut qu'à chaque apparition d'une nouvelle lettre, je puisse vérifier qu'elle est compatible avec ce qui précède, sinon émettre un diagnostic précis ; si oui, alors il faut apporter la modification qu'introduit cette nouvelle lettre. Je dirai que je me donne un ensemble cohérent de variables d'état qui me sert à noter l'état de la phrase. Dans le premier cas je consulte l'état de la phrase où j'en étais resté, et dans l'autre, je modifie la configuration des variables d'état pour tenir compte de l'apparition de la nouvelle lettre.

C.I.R.M. LUMINY BIBLIOTHEQUE

Un ensemble d'objets vont me servir pour construire la phrase compilée, bien que ne faisant pas partie du résultat final, ils sont utiles pendant tout le traitement de la phrase symbolique. C'est le cas des identificateurs, qui sont d'abord déclarés puis utilisés pour construire les instructions. La déclaration précise en général les types, dimensions et autres caractéristiques que l'on utilise au moment où on construit l'instruction qui contient la variable. Il faut donc pouvoir stocker l'information qui se rattache à celle-ci au moment où l'identificateur qui la représente apparaît. Cela rendra possible ultérieurement la récupération de cette information chaque fois que cela sera nécessaire. Je baptiserai le support d'une telle information : variables de sémantique immédiate.

J'aurai besoin de variables de sémantique définitive pour noter le résultat final de la compilation, autrement dit le programme traduit.

En résumé, le compilateur compact se caractérise par :

a) Une analyse de la phrase lettre à lettre.

b) La constitution d'un ensemble de variables d'état syntaxique.

c) Création d'un ensemble de variables de sémantique immédiate.

d) Création d'un ensemble de variables de sémantique définitive.

Au niveau de la sémantique immédiate, nous verrons apparaître des boucles non bornées et j'introduirai de la commutation globale pour chacune d'elles.

L'application au langage que je vais définir maintenant montrera la signification de ces définitions.

à suivre...

Bibliographie.

[1] Structures de commutation - Thèse 1978 - Patrick ISCARDI.

[2] Informatique fondamentale - Birkhaüser Verlag 1979 - Edmond BIANC

[3] Notion d'algorithme et de programme - Gauthier Villard - Louis NOLIN.

= : = : = : = : = : =