Institutional Repository - Research Portal
Dépôt Institutionnel - Portail de la Recherche
researchportal.unamur.be
UNIVERSITÉ DE NAMUR
University of Namur

THESIS / THÈSE

MASTER EN SCIENCES INFORMATIQUES

Définition d'un langage de programmation permettant l'expression d'assertions

Fisette, Denis

Award date:
1985

Awarding institution:
Universite de Namur

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal ?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 02. May. 2026

Facultés Universitaires Notre-Dame de la Paix - NAMUR

INSTITUT D'INFORMATIQUE

DEFINITION D'UN LANGAGE DE PROGRAMMATION PERMETTANT L'EXPRESSION D'ASSERTIONS

Denis FISETTE

Mémoire présenté en vue de l'obtention du grade de licencié et maître en informatique.

Année académique 1984-1985

Que toutes les personnes qui ont contribué à l'élaboration de ce mémoire trouvent ici l'expression de mes remerciements sincères. Je pense tout spécialement à B. Le Charlier pour l'intérêt qu'il a porté à ce travail, pour sa constante disponibilité et ses conseils fructueux.

TABLE DES MATIÈRES

Page

INTRODUCTION 1

Chapitre I CADRE ET OBJECTIFS 3

1.1. Considérations générales 4

1.2. Quelques notions intéressantes 6

a. La spécification d'un programme 6

b. Notion d'assertion 7

c. Propriétés "intéressantes" d'un programme 8

d. Quelques conventions de présentation supplémentaires 8

e. Point de bouclage, programme auxiliaire 10

1.3. La méthode de l'invariant 12

1. La correction partielle 12

2. La terminaison 16

1.4. La méthode du programme auxiliaire 18

1.5. L'outil proposé: un langage permettant l'expression d'assertions 22

Chapitre II ELABORATION D'UN LANGAGE PERMETTANT L'EXPRESSION D'ASSERTIONS 25

2.1. Démarche suivie 26

2.2. Un programme sans boucles 27

2.3. Deux programmes élémentaires avec boucles 30

2.4. Intérêt des concepts de suite et d'ensemble 33

2.5. Construction d'expressions utilisant des suites et des ensembles 38

2.5.1. Expressions booléennes avec "quantificateurs" 38

2.5.2. Opérateurs arithmétiques "itératifs" 41

2.5.3. Le domaine de référence d'une expression 43

2.5.4. Conditions limites 44

2.5.5. Traitement des infinis 46

2.5.6. Construction de suites et d'ensembles 50

2.6. Opérations définies par l'utilisateur 54

2.7. Quant à la terminaison des programmes 57

2.8. Deux programmes plus compliqués 61

Chapitre III EXTENSIONS POSSIBLES ?... 68

3.1. Pour illustrer la méthode du programme auxiliaire... 69

3.2. Un outil pour le développement de logiciel ?...

a. Adapter ce type d'outil à des programmes en vraie grandeur poserait de sérieux problèmes de coûts 72

b. Un langage de programmation d'assertions ne pourrait être "complet" qu'en limitant précisément les applications que l'on envisagera et non en restreignant les primitives du langage de programmation initial 73

c. Même en limitant le domaine d'application d'un tel outil, la vérification à l'exécution des assertions peut être très compliquée, voire impossible 73

d. Si un "bon" langage d'expression de spécifications ou d'assertions existait, il serait plus intéressant de l'utiliser comme langage de programmation 74

c. Quant aux démonstrateurs automatiques... 75

CONCLUSION 76

BIBLIOGRAPHIE 77

INTRODUCTION

Les échecs d'expériences passées tendent à souligner l'intérêt de s'assurer de la correction des programmes par des méthodes "rigoureuses". Il existe différentes méthodes de démonstration de programmes mais elles ont toutes le défaut d'être assez "lourdes" et recueillent, par la même occasion, peu d'adeptes. Pour suggérer l'intérêt de ces méthodes et familiariser les étudiants avec deux d'entre elles, se donne, en première licence en informatique, un cours intitulé "Séminaire de programmation". Ce mémoire tente de l'illustrer en proposant un outil qui permet aux étudiants de "tester" leur travail; il espère ainsi motiver quelque peu la résolution d'exercices supplémentaires nécessaires à la maîtrise de ces méthodes.

L'outil proposé est un langage de programmation permettant l'expression de certaines "propriétés" des programmes en vue de leur vérification à l'exécution. La majeure partie du travail a été consacrée à la définition de ce langage mais une telle définition est fastidieuse à lire. C'est pourquoi nous l'avons présentée en annexe.

Quant au mémoire "proprement dit", il est organisé de la manière suivante. Un premier chapitre situe le cadre de notre travail et présente deux méthodes de démonstration de programmes. Il précise ensuite plus explicitement quel outil nous proposerons en indiquant, dès ce moment, les limites inhérentes à ce genre de travail.

Un deuxième chapitre tente de cerner les caractéristiques principales du langage envisagé en considérant progressivement des exemples de plus en plus compliqués. Nous tâcherons ainsi d'indiquer l'utilité de differents concepts et opérations pour ce type de langage.

Le but premier du langage proposé était d'illustrer la méthode de l'invariant. Les modifications à effectuer pour illustrer l'autre méthode seront évoquées dans la première partie du chapitre III. La seconde partie, quant à elle, tentera d'expliquer les raisons pour lesquelles il semble trop ambitieux de vouloir étendre ce type de travail dans l'optique du développement de logiciels.

Notre travail peut être mis en rapport avec d'autres visant à démontrer formellement la correction de programmes (par exemple, [4]). Toutefois, notre objectif est plus modeste puisqu'il se limite à certaines vérifications à l'exécution. Cela nous a d'ailleurs permis de proposer un langage permettant l'expression de "propriétés" plus compliquées.

Chapitre I

CADRE

ET OBJECTIFS

1.1. CONSIDERATIONS GENERALES

Le problème de la fiabilité des programmes est de plus en plus à l'ordre du jour: non seulement, l'informatique a tendance à se répandre dans des domaines d'application où l'erreur est intolérable, mais surtout, tenant compte des échecs passés, on constate que la correction des erreurs de programmation détient une place beaucoup trop importante dans le coût d'un logiciel. Ce phénomène est d'autant plus accentué s'il s'agit d'un logiciel de grande taille. En effet, détecter une erreur est une chose, la localiser et la corriger en est une autre !...

Face à ce problème, deux styles de démarches sont proposées: l'élaboration et la conduite de tests et les démonstrations de programmes. Le désavantage principal de la première démarche provient du fait qu'un jeu de tests d'un programme de taille normale, aussi bon soit-il, ne peut être exhaustif. Le but de cette démarche sera donc de détecter des erreurs. Le but de la seconde, par contre, est de montrer l'absence d'erreurs, mais elle a le gros défaut d'être assez "lourde" et, par la même occasion, recueille moins d'adeptes. De plus, toute démonstration de programme est également sujette à l'erreur, surtout qu'en ce domaine interviennent des contraintes temporelles non négligeables: un logiciel est si vite démodé qu'on ne peut se permettre de s'assurer de sa fiabilité qu'en un laps de temps raisonnable. Ces idées sont proposées par le professeur H. Leroy dans la conclusion de son ouvrage intitulé "La fiabilité des programmes" [2].

Tâchant de cerner le pourquoi de la non-popularité des méthodes de démonstration de programmes, il nous a semblé qu'elle provenait principalement du style de raisonnement nécessité dans ces démonstrations: le raisonnement par récurrence ou, pour être plus général, par induction. Ce type de raisonnement permet de traiter "d'un seul coup, une infinité de cas"; c'est justement ce qui fait la puissance de cette démarche face aux tests.

Ce mémoire propose un outil d'illustration d'un cours, intitulé "séminaire de programmation", dont le but premier est de "démystifier" ce

type de raisonnement et, par la même occasion, certaines méthodes de démonstration de programmes. Deux d'entre elles y sont envisagées: la méthode dite de l'invariant et la méthode dite du programme auxiliaire (☆). Ces deux méthodes seront présentées brièvement dans les trois paragraphes suivants. Le premier d'entre eux "définit" quelques notions et fixe certaines conventions utilisées dans les deux autres. Ces deux derniers, troisième et quatrième paragraphe, se soucient de présenter, respectivement, la méthode de l'invariant et celle du programme auxiliaire.

Un dernier paragraphe indique plus explicitement quel type d'outil nous allons proposer: un langage de programmation permettant l'expression de certaines "propriétés" des programmes en vue de leur vérification à l'exécution. Nous aborderons dès ce moment les limites inhérentes à ce type de travail. Comme son objectif principal est de permettre l'expression aisée des problèmes envisagés dans le cours mentionné plus haut, il est intéresssant d'essayer de cerner plus ou moins finement la classe de ces problèmes avant de présenter ce langage: ceci fera l'objet de la fin de ce chapitre.

(☆) qualifiée également de méthode de raisonnement par récurrence descendante sur les données [1].

1.2. QUELQUES NOTIONS INTERESSANTES

a. La spécification d'un programme

La démonstration d'un programme consiste à montrer que ce dernier est conforme à sa spécification. Avant d'aller plus loin, il est donc intéressant d'indiquer ce que nous entendons par spécification d'un programme. Nous appellerons spécification d'un programme un énoncé ayant pour rôle de dire à quoi sert le programme et comment faire pour l'utiliser correctement (☆). Cette "définition" (☆☆) est celle proposée par B. Le Charlier comme point de départ pour les considérations du chapitre II de la seconde partie de sa thèse [1]. Ce même chapitre se penche notamment sur la "forme générale" de la spécification d'un programme en indiquant qu'elle doit toujours comporter deux parties jouant des rôles fort différents:

1. un énoncé indiquant à quoi sert le programme, c'est-à-dire quelle est l'information (☆☆☆) que l'on pourra tirer des résultats de son exécution;
2. une liste de conventions de représentation à respecter pour utiliser le programme correctement et, notamment, pour interpréter convenablement ses résultats.

L'auteur de la thèse en question insiste également sur l'importance de la "théorie" du problème à résoudre, précédant la spécification et nécessaire pour la rendre simple et compréhensible.

(☆) "A la fois son cahier des charges et son mode d'emploi" [2].

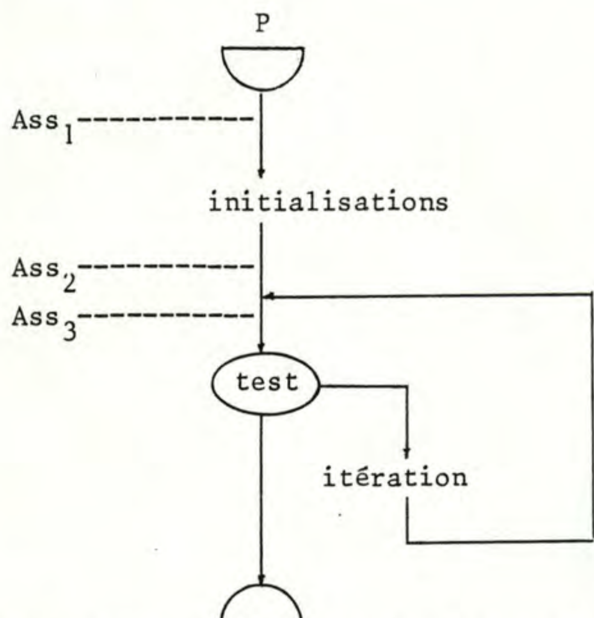
(☆☆) Le terme "définition" n'est sans doute pas approprié pour ce genre de notions.

(☆☆☆) Au sens commun du terme et non au sens qu'on lui donne dans le jargon informatique [3].

b. Notion d'assertion

Nous appellerons assertion d'un programme, une affirmation au sujet de l'état de ses variables à un moment donné de son exécution ou à un ensemble de tels moments.

C'est sous forme d'organigramme que nous présenterons les différents programmes envisagés. Les assertions d'un programme seront situées sur son organigramme par un trait discontinu les associant à certains de ces arcs. Nous indiquerons ainsi le ou les moments de l'exécution auxquels elles se réfèrent. Considérons l'exemple suivant:



L'assertion Ass_1 se réfère au début de l'exécution de P et Ass_2, au moment suivant l'exécution des initialisations. Quant à Ass_3, elle se réfère à l'ensemble des moments de l'exécution précédant chaque évaluation du test.

Remarques: 1. Une assertion peut ne pas avoir de sens notamment si elle énonce une affirmation au sujet de variables non initialisées.

2. Une assertion "utile" doit souvent faire référence aux valeurs qu'ont eues précédemment certaines variables. Dans ce paragraphe, nous contournerons ce problème en utilisant suffisamment de variables distinctes. Par exemple les variables contenant les "données" ne seront pas modifiées: d'autres variables seront utilisées dans les opérations internes au programme.

c. Propriétés "intéressantes" d'un programme

Avant de présenter ces deux méthodes, précisons les trois propriétés d'un programme qui nous intéresseront:

- Correction ou correction totale: On dit qu'un programme est (totalement) correct par rapport à une spécification ssi, pour toute valeur des données permise par la spécification, l'exécution du programme se termine et fournit un résultat conforme à la spécification.
- Correction partielle: On dit qu'un programme est partiellement correct par rapport à une spécification ssi, pour toute valeur des données permise par la spécification, si son exécution se termine, elle fournit un résultat conforme à la spécification.
- Terminaison: On dit qu'un programme est correct du point de vue de la terminaison par rapport à une spécification si son exécution se termine pour toute valeur des données permise par la spécification.

Pour démontrer la première propriété, la seule, en fait, qui soit réellement intéressante, certaines méthodes proposent de le faire en deux phases, l'une se souciant de démontrer la correction partielle et l'autre, la terminaison.

d. Quelques conventions de présentation supplémentaires

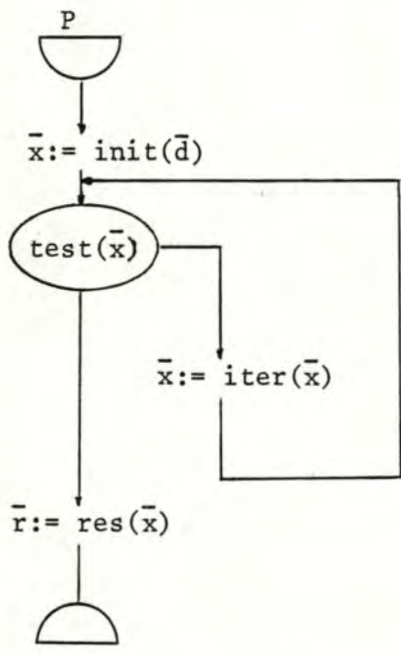
Les deux méthodes que nous présenterons ici visent à démontrer la correction d'un programme (organigramme (☆)) qui se résume en une boucle et quelques opérations d'initialisation et de clôture. La démonstration d'un programme comportant plusieurs boucles imbriquées pourra se faire en effectuant la démarche suivante: on considère la boucle la plus interne (plus certaines opérations d'initialisation ou de clôture) comme un sous-programme que l'on spécifie; après avoir démontré que ce dernier est conforme à cette

---

(☆) La présentation des programmes sous forme d'organigramme est souvent critiquée; elle me semble cependant bien appropriée lorsqu'il s'agit de raisonner au sujet de leur correction (et d'autant plus si ce raisonnement se base sur leur exécution!).

spécification, on envisage à nouveau la démonstration du programme tout entier mais, cette fois, en considérant la boucle la plus interne comme une opération primitive répondant à la spécification en question.

Nous nous intéresserons donc ici à des programmes ayant la forme suivante:



où $\bar{d}$ désigne l'ensemble des variables contenant les données du programme,

$\bar{r}$, celui des variables contenant les résultats et

$\bar{x}$, celui des variables internes au programme.

On supposera que ces ensembles sont disjoints deux à deux. On notera également $\bar{D}$, $\bar{R}$ et $\bar{X}$, les ensembles des valeurs pouvant être prises respectivement par $\bar{d}$, $\bar{r}$ et $\bar{x}$. De plus, init, test, iter et res seront considérées comme des "primitives" (☆) calculant des fonctions totales auxquelles on donnera le même nom:

init: $D \longrightarrow X$

res : $X \longrightarrow R$

test: $X \longrightarrow B = \{\text{vrai}, \text{faux}\}$

iter: $X \longrightarrow X$

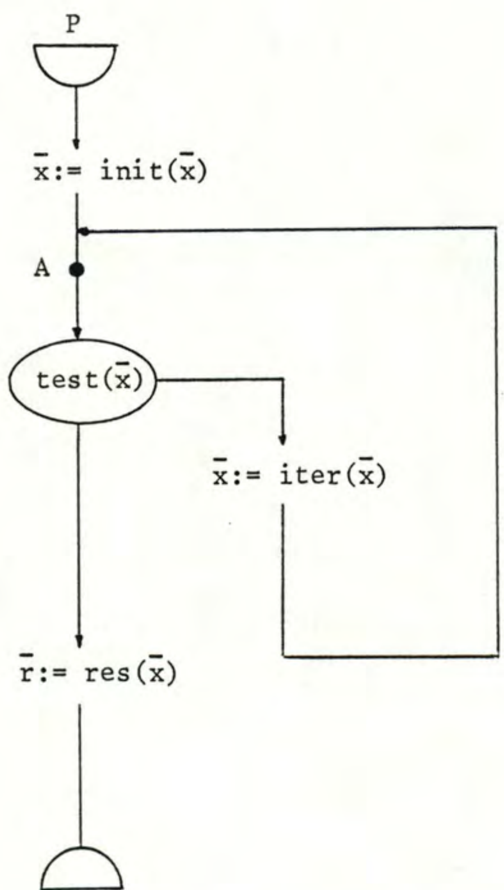
(☆) Les guillemets indiquent qu'on ne préjuge en rien de la complexité de ces opérations.

Enfin, nous supposerons dans ce qui suit que la spécification du programme P est un énoncé déterminant complètement la fonction partielle calculée par ce programme. Celle-ci sera notée f.

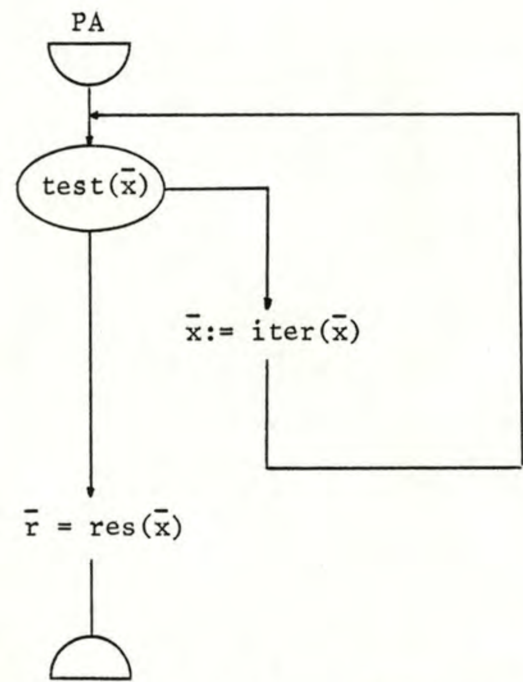
En se limitant à des programmes vérifiant ces contraintes, la présentation des deux méthodes de démonstration de programme est grandement tronquée mais elle permet quand même d'indiquer les raisonnements qui font l'essence de ces méthodes. Leurs présentations sans ces restrictions figurent dans l'ouvrage [1].

e. Point de bouclage, programme auxiliaire

Nous appellerons, point de bouclage d'un programme, le point de l'organigramme "situé juste avant" le test et noté A sur l'organigramme suivant:



Nous appellerons, enfin, <u>programme auxiliaire associé à un programme</u> du type envisagé, le programme ci-dessous, obtenu "en supprimant les instructions d'initialisation":



Ayant fixé ces quelques notions et conventions, nous sommes maintenant en mesure de présenter deux méthodes de démonstration de programme: le paragraphe suivant se préoccupe de la méthode de l'invariant et le paragraphe 4, de la méthode du programme auxiliaire. Nous tenons à souligner dès à présent que le contenu de ces paragraphes se limite à une approche intuitive de ces deux méthodes. Nous nous sommes principalement basé sur les ouvrages [1] et [2] où l'on en trouve une présentation plus complète et plus rigoureuse.

Nous utiliserons certaines notions telles celles de relation d'ordre, relation bien fondée ou encore de démonstration par récurrence ou par induction. Des précisions sur ces notions se trouvent notamment dans l'ouvrage [1].

1.3. LA METHODE DE L'INVARIANT

Cette méthode a la particularité de démontrer la correction d'un programme en deux phases distinctes: la première concerne la correction partielle et la seconde, la terminaison.

1. La correction partielle

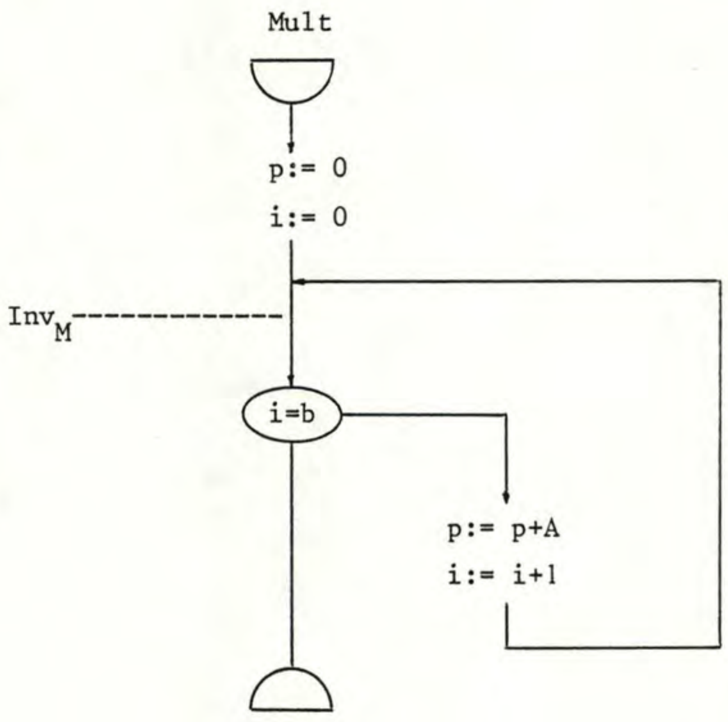
a. Une première approche...

La correction partielle du programme considéré se démontre en trouvant (☆) une assertion associée au point de bouclage du programme. Nous appellerons dorénavant cette assertion l'invariant du programme. L'invariant doit donc être vérifié à chaque passage au point de bouclage; il se doit d'exprimer "l'état d'avancement" de l'exécution du programme quel que soit le nombre de tours de boucle déjà effectués. Formulons cela autrement: il s'agit de répondre à la question suivante: en supposant que l'exécution du programme ne soit toujours pas terminée après un certain nombre d'itérations, qu'est-ce qu'elle a déjà établi ? De plus, l'invariant doit exprimer suffisamment de choses pour que, si l'exécution se termine (i.e. si le test renvoie la valeur vrai), il permette de déduire que les résultats fournis sont conformes à la spécification.

b. Un exemple simple

Considérons l'organigramme ci-dessous répondant à la spécification suivante: étant donné a et b, deux variables entières initialisées de telle sorte que $b \geqslant 0$, affecter à la variable p la valeur du produit de a et de b sans modifier leurs valeurs respectives.

(☆) Trouver l'invariant relève du raisonnement et de l'intuition: il n'y a pas de "recette miracle" pour ce faire.



Exprimons "l'état d'avancement" de l'exécution du programme, quel que soit le nombre de tours de boucle déjà effectués: $p = i * a$ et $i \geq 0$. De plus, la spécification du programme mentionne qu'on ne peut modifier les deux variables a et b. Pour pouvoir en déduire la correction partielle, complétons l'invariant :

$Inv_M \equiv$ a et b non modifiés et $i \geq 0$ et $p = i * a$.

c. Précisons quelque peu tout cela...

Considérons le programme P décrit page 9 . L'invariant de P sera une affirmation qui dépendra généralement des variables $\bar{x}$ et $\bar{d}$. Notons Inv $\langle \bar{d}, \bar{x} \rangle$ cette affirmation;elle est supposée avoir un sens quelles que soient les valeurs de ces variables.

La démonstration de correction partielle consiste à démontrer d'abord le lemme suivant:

Lemme: L'invariant est vérifié à chaque passage de l'exécution au point de bouclage.

Pour démontrer ce lemme, on raisonne par récurrence sur le nombre de ces passages:

a. On montre que l'invariant est vérifié au premier passage, ou encore que Inv $<\bar{d}, \text{init}(\bar{d})>$ est vrai pour tout $\bar{d} \in \bar{D}$.

   Exemple considéré: ($\bar{d} = (a,b)$ et $\bar{x} = (i,p)$)

   L'invariant était

   $$Inv_M \equiv a \text{ et } b \text{ non modifiés et } i \geqslant 0 \text{ et } p = i * a.$$

   Il est bien vérifié au premier passage au point de bouclage puisque les opérations d'initialisation consistent à affecter zéro aux variables i et p sans modifier les variables a et b.

b. On suppose qu'à un certain passage, l'invariant est vérifié et qu'il y a un passage suivant (i.e. le test test $(\bar{x})$ renvoie la valeur faux); il s'agit de montrer que l'invariant est encore vérifié en ce nouveau passage ou encore, de montrer que

   $$\forall\, \bar{x} \in \bar{X} \text{ et } \bar{d} \in \bar{D}: \left.\begin{array}{r}\text{Inv} <\bar{d},\bar{x}> \text{ et}\\ \text{test}(\bar{x}) = \text{faux}\end{array}\right] \Rightarrow \text{Inv}<\bar{d},\text{iter}(\bar{x})>$$

   Exemple: ($\text{iter}\,(i_o,p_o) = ((i_o+1),(p_o+a))$)

   Supposons donc, qu'à un certain passage, on ait

   $Inv_M <(a,b),(i_o,p_o)>$ et $(i_o \neq b)$ et voyons que

   $Inv_M <(a,b),(i_o+1,p_o+a)>$ est bien vérifié: en effet, aucune opération de la boucle ne modifie a ou b et, comme i_o est positif, (i_o+1) l'est aussi. De plus, comme

   $$p_o = i_o * a,$$

   on a

   $$p_o+a = i_o * a + a;$$

   d'où l'on déduit:

   $$p_o+a = (i_o+1) * a.$$

Une fois ce lemme démontré, on déduit la correction partielle du programme P en montrant que, si l'exécution de ce programme se termine (i.e. test $(\bar{x})$ = vrai), l'invariant Inv $<\bar{d},\bar{x}>$ entraîne que les résultats sont

conformes à la spécification ou encore, que

$$\forall\ \bar{x} \in \bar{X} \text{ et } \bar{d} \in \bar{D}: \left.\begin{array}{r}\text{Inv } <\bar{d},\bar{x}> \text{ et} \\ \text{test } (\bar{x}) \text{ est vrai}\end{array}\right] \Rightarrow f(\bar{d}) = res(\bar{x})$$

Exemple: On déduit facilement de l'invariant

Inv_M et de (i=b) (c'est-à-dire a,b non modifié et $i \geq 0$ et $p = i * a$ et $i = b$) qu'à la fin de l'exécution de ce programme (si fin il y a) p vaut le produit des valeurs des variables a et b, celles-ci n'ayant pas été modifiées.

Remarquons que, dans la démonstration de correction partielle de cet exemple, nous n'avons jamais utilisé le fait que b doit être positif. Cette même démonstration serait donc tout aussi valable pour montrer que ce même organigramme est partiellement correct face à la spécification où l'on change $(b \geq 0)$ par $(b < 0)$. Ce programme serait donc partiellement correct face à cette spécification. Remarquons qu'en fait, il bouclerait indéfiniment pour toutes valeurs permises pour a et b puisque l'invariant et le test vérifié sont contradictoires:

$Inv_M \equiv \ldots$ et $i \geq 0$ et... et $(i = b)$.

d. Remarque

Remarquons, enfin, que cette méthode de démonstration de la correction partielle est basée sur la structure de l'exécution. En effet, le raisonnement qu'il s'agit de faire est un raisonnement par récurrence sur les préfixes de l'exécution du programme (☆).

(☆) Elle sera qualifiée de méthode ascendante (car considère des préfixes de l'exécution) de raisonnement sur la structure de l'exécution [1].

2. La terminaison

a. Une première approche...

Pour démontrer la terminaison, on tentera de trouver une fonction entière des valeurs des variables qui soit bornée inférieurement par zéro et l'on montrera que la valeur de cette fonction décroît strictement à chaque tour de boucle. L'exécution se terminera donc.

Exemple: Considérons le programme Mult et remarquons que la valeur de la fonction f_M traduite par l'expression b-i va décroître à chaque tour de boucle d'une unité; reste à montrer qu'elle est bornée inférieurement par zéro.

b. Précisons quelque peu tout cela...

Considérons à nouveau le programme P; il s'agit de trouver une fonction

$$f : \bar{X}' \longrightarrow Z^+$$

où init $(\bar{D}) \subseteq \bar{X}' \subseteq \bar{X}$ et

Z^+ est l'ensemble des entiers positifs.

On montre ensuite que les valeurs successives de $\bar{x}$ au point de bouclage restent dans $\bar{X}'$ et que la valeur de la fonction f évaluée en ces valeurs, décroît strictement d'une itération à l'autre. Résumons cela en disant qu'il s'agit de montrer que

a. init $(\bar{D}) \subseteq \bar{X}'$,

b. $\forall\, \bar{x} \in \bar{X}' : f(\bar{x}) \geq 0$ et

c. $\forall\, \bar{x} \in \bar{X}' :$ test $(\bar{x})$ = faux $\Longrightarrow \left[\begin{array}{l} \text{iter}(\bar{x}) \in \bar{X}' \text{ et} \\ f(\text{iter}(\bar{x})) < f(\bar{x}) \end{array}\right.$

Exemple: Voyons que la fonction f_M décrite précédemment n'est pas bornée inférieurement par zéro lorsqu'on la considère sur $\bar{X}_M (\equiv Z \times Z)$ tout entier. Il s'agit donc de limiter son domaine. Considérons l'ensemble $\bar{X}'_M = \{(i,p) \in \bar{X}_M \mid 0 \leq i \leq b\}$ et remarquons que f_M restreinte

à cet ensemble est bien bornée inférieurement par zéro. De plus, init ((a,b)) $\in \bar{X}'_M$ pour toute valeur des données permises par la spécification. Montrer que les valeurs de (i,p) restent dans $\bar{X}'_M$ se fait facilement en raisonnant comme nous l'avons fait pour vérifier l'invariant. Très souvent d'ailleurs, on choisira un invariant plus complet en vue de faciliter la démonstration de la terminaison. Pour le programme Mult envisagé, on choisira, par exemple, l'invariant suivant:

$$Inv_M \equiv a \text{ et } b \text{ non modifiés et } 0 \leq i \leq b \quad \text{et} \quad p = i * a$$

c. D'une manière plus générale...

Pour démontrer la terminaison, nous avons indiqué qu'il s'agissait de trouver une fonction entière; de plus, nous avons fixé explicitement une borne inférieure de cette fonction en disant que les valeurs qu'elle renvoie sont positives. D'une manière plus générale, il suffit, en fait, de trouver une fonction à valeurs dans un ensemble muni d'une relation bien fondée. La présentation de cette méthode de démonstration dans le cas général se trouve dans l'ouvrage [1]. Par la suite, nous remarquerons que ces restrictions ne sont pas significatives pour la classe des problèmes envisagés. En fait, ce qui est plus contraignant n'est pas de se limiter à des fonctions entières mais bien de se limiter à des relations bien fondées particulières que sont les relations d'ordre total dont le plus petit élément est fixé explicitement. Nous verrons au chapitre II un exemple très simple où l'on sait que la fonction en question est bornée inférieurement mais pour laquelle il n'est guère trivial de donner explicitement cette borne inférieure.

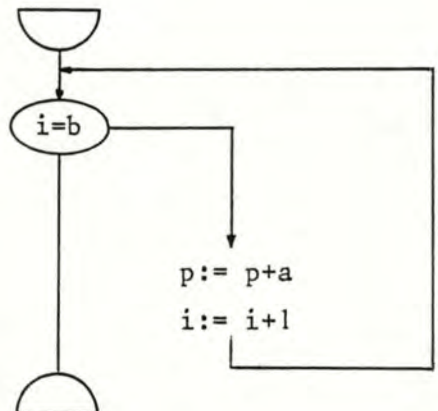
1.4. LA METHODE DU PROGRAMME AUXILIAIRE

a. Une première approche

Cette seconde méthode est basée sur la démarche suivante: il s'agit de considérer le programme auxiliaire associé au programme envisagé et de le spécifier en fonction des valeurs initiales de ses variables. On démontre ensuite la correction de ce programme auxiliaire par rapport à sa spécification en raisonnant par induction sur les valeurs initiales de ses variables. On déduit enfin, la correction du programme principal en particularisant la spécification du programme auxiliaire aux valeurs résultant des instructions d'initialisation du programme principal.

Etablir la spécification du programme auxiliaire est du même ordre de difficulté que de trouver l'invariant dans la première méthode. Cette spécification peut être suggérée par la réponse à la question suivante: se plaçant dans la situation générale atteinte après un certain nombre d'itérations du programme principal, que reste-t-il à faire pour fournir des résultats conformes à la spécification de ce dernier ?

Exemple: Considérons le programme auxiliaire du programme Mult:



Plaçons-nous dans la situation générale atteinte après quelques itérations du programme principal et notons p_o et i_o les valeurs respectives des variables p et i. La fin de l'exécution de Mult consistera à ajouter à la

variable p la quantité $(b - i_o) * a$. De ces constatations découle la spécification du programme auxiliaire:
Soient a et b deux variables initialisées de telle sorte que $b \geqslant 0$; soient encore p_o et i_o, $i_o \leqslant b$, les valeurs respectives des variables p et i à l'entrée du programme auxiliaire; son exécution aura pour effet d'affecter à la variable p, la valeur $p_o + (b - i_o) * a$ sans modifier ni a ni b.

<u>Précisons quelque peu tout cela...</u>

Soit PA, le programme auxiliaire du programme P décrit page 9.
Il s'agit de trouver une fonction totale

$$g : \bar{X}' \longrightarrow \bar{R}$$

où init $(\bar{D}) \subseteq \bar{X}' \subseteq \bar{X}$ et où
$\bar{X}'$ est muni d'une relation bien fondée notée $\leqslant$

et de montrer que le programme auxiliaire calcule bien cette fonction, ce qui est traduit par le lemme suivant:

<u>Lemme</u>: Pour tout $\bar{x}_o \in \bar{X}'$, l'exécution du programme auxiliaire avec $\bar{x} = \bar{x}_o$ se termine avec $\bar{r} = g(\bar{x}_o)$.

<u>Exemple</u>: Prenons comme domaine de la fonction g_M, l'ensemble $\bar{X}'_M = \{(i,p) \in \bar{X}_M \mid i \leqslant b\}$. Munissons-le de la relation bien fondée (☆) caractérisée par

$$(i_o,p_o) \leqslant (i_1,p_1) \quad \text{ssi} \quad i_o > i_1$$

et définissons g_M de la façon suivante:

$$g_M = \bar{X}' \longrightarrow \bar{R}$$
$$(i,p) \longrightarrow p + (b-i) * a$$

Pour démontrer ce lemme, on raisonnera par induction sur $(\bar{X}', \leqslant)$:

a. On montre que, si $\bar{x}_o \in \bar{X}'$ est tel que $test(\bar{x}_o)$ = vrai, le programme auxiliaire avec $\bar{x}_o$ comme valeur initiale calcule bien $g(\bar{x}_o)$ ou encore, que

$$g(\bar{x}_o) = res(\bar{x}_o) \quad \text{si} \quad b(\bar{x}_o) = \text{vrai}.$$

(☆) On constate facilement que cette relation sur $\bar{X}$ est une relation bien fondée puisque $(i,p) \in \bar{X}'_M \Longrightarrow i \leqslant b$.

b. On montre que si $\bar{x}_o \in \bar{X}'$ est tel que $test(\bar{x}_o)$ = faux, les propriétés ci-dessous sont vérifiées:

1. $iter(\bar{x}_o) \in \bar{X}'$ et $iter(\bar{x}_o) \lessdot \bar{x}_o$

2. $g(iter(\bar{x}_o)) = g(\bar{x}_o)$

On en déduit que si $test(\bar{x}_o)$ est faux, l'exécution du programme auxiliaire pour $\bar{x} = \bar{x}_o$ repassera au point d'entrée en $\bar{x} = iter(\bar{x}_o)$. A partir de là, l'exécution se déroulera comme si elle avait été déclenchée en $\bar{x} = iter(\bar{x}_o)$. Mais, comme $iter(\bar{x}_o) \in \bar{X}'$ et $iter(\bar{x}_o) \lessdot \bar{x}_o$, on peut supposer, par hypothèse d'induction, que le lemme est vrai pour $iter(\bar{x}_o)$. L'exécution du programme auxiliaire se terminera donc avec

$$\bar{r} = g(iter(\bar{x}_o))$$

Or, on vient de montrer que $g(iter(\bar{x}_o)) = g(\bar{x}_o)$.

Exemple: Prenons $(i_o,p_o) \in \bar{X}'$ tel que $i_o \neq b$. On a donc que $i_o < b$ d'où l'on déduit que

$$iter(i_o,p_o) = ((i_o+1),(p_o+a)) \in \bar{X}'.$$

On remarquera ensuite que

$$((i_o+1),(p_o+a)) \lessdot (i_o,p_o) \quad \text{puisque } i_o+1 > i_o.$$

Enfin,

$$\begin{aligned} g_M((i_o+1),(p_o+a)) &= p_o+a + (b-i_o-1) * a \\ &= p_o + (b-i_o) * a \\ &= g_M(i_o,p_o) \end{aligned}$$

ce qui termine la démonstration du lemme associé au programme Mult.

La démonstration de la correction du programme principal P se termine en montrant qu'on a l'égalité suivante:

$$f(\bar{d}_o) = g(init(\bar{d}_o)) \qquad \forall\, \bar{d}_o \in \bar{D}$$

On en déduit la correction du programme principal puisque son exécution revient à exécuter le programme auxiliaire PA en $\bar{x}_o = init(\bar{d}_o)$ et que ce dernier, d'après le lemme, renverra $\bar{r} = g(init(\bar{d}_o))$.

Exemple: Pour démontrer la correction totale de Mult, il reste à voir que $f_M(a,b) = a * b = g_M((0,0))$.

Remarque:

Contrairement à la méthode de l'invariant, cette méthode du programme auxiliaire est basée sur un raisonnement par induction sur les données; en effet, la démonstration du lemme revient à établir, par induction sur $(\bar{X}', \leqslant)$, que tout suffixe de l'exécution issu du point de bouclage avec $\bar{x} = \bar{x}_o$ se termine avec $\bar{r} = g(\bar{x}_o)$. (☆)

(☆) Elle sera qualifiée de méthode descendante (car considère des suffixes de l'exécution) de raisonnement par induction sur les données [1].

I.5. L'OUTIL PROPOSE : UN LANGAGE PERMETTANT L'EXPRESSION D'ASSERTIONS

Le cours intitulé "Séminaire de programmation" insiste sur la nécessité de résoudre un grand nombre d'exercices pour se familiariser avec les méthodes de démonstration de programmes que nous venons de présenter. Mais, comme nous l'avons déjà signalé, les démonstrations de programmes sont également sujettes à l'erreur. Le but de ce travail sera de fournir un outil permettant aux étudiants de s'assurer de la correction (☆) des démonstrations de programmes qu'ils ont faites. C'est plus précisément les démonstrations par la méthode de l'invariant que nous tenterons d'illustrer en proposant un langage permettant l'expression de spécifications et d'invariants (☆☆). A priori, ces deux objectifs ne semblent pas nécessiter une même approche. Remarquons cependant que la spécification d'un programme peut être présentée sous forme de pré/postcondition : la précondition exprime des conditions suffisantes sur les données du programme pour que son exécution soit déterminée, se termine et fournisse des résultats conformes à la postcondition (☆☆☆). Voyons ensuite que la précondition, la postcondition ou l'invariant d'un programme peuvent tous trois être considérés comme des assertions relatives à des moments caractéristiques de l'exécution, respectivement, le début, la fin et chaque passage au point de bouclage.

Ce sera donc un langage permettant l'expression d'assertions que nous proposerons ici; l'intérêt et la spécificité d'un tel langage n'apparaissent que lorsqu'on précise qu'il s'agit de vérifier ces assertions à l'exécution. Pour satisfaire cet objectif, il faudra donc "formaliser ces assertions" et c'est de cette formalisation que découlent les limites de ce type de travail: nous verrons qu'exprimer des assertions dans un langage formel est une activité

(☆) Puisqu'il s'agit de vérifications à l'exécution, il serait plus juste de dire "pour détecter un maximum d'erreurs".

(☆☆) Nous indiquerons au troisième chapitre quel type de modifications effectuer pour que ce langage permette d'illustrer la seconde méthode.

(☆☆☆) Précondition et postcondition doivent jouer, à elles deux, les deux rôles distincts d'une spécification mentionnés au paragraphe I.2.

fort proche de la programmation (☆). Tout au long des chapitres II et III, nous essayerons de souligner la pertinence de cette remarque lourde de conséquences, l'une d'entre elles étant la suivante: la formalisation des assertions et en particulier, des spécifications ([1] et [2])"dénature"leurs rôles respectifs. Une des raisons sous-jacentes à ces constatations est, à notre avis, la suivante: un langage formel a la lourde contrainte d'avoir une syntaxe et une sémantique fixée une fois pour toute. Or, il semble peu vraisemblable de pouvoir délimiter le domaine d'application (☆☆) d'un langage de programmation puisque ces limites sont fortement liées à celles de l'imagination et de l'intelligence humaines. C'est pourquoi, il nous semble tout aussi peu vraisemblable qu'un langage formel permette l'expression "directe" (i.e. "sans programmation") de toute propriété relative à un des programmes exprimables dans un langage normal. De plus, en admettant qu'un tel langage existe pour une classe restreinte de programmes, rien ne prouve que la vérification à l'exécution des assertions et des spécifications en particulier soit possible.

L'objectif de ce travail n'en est pas pour autant reduit à néant. En effet, nous avons insisté d'emblée sur les prétentions limitées d'un tel travail: fournir un outil d'illustration d'un cours et non un outil pour la construction de logiciels "utiles" (☆☆☆). C'est pourquoi nous allons tenter de cerner plus ou moins finement la classe des problèmes envisagés dans ce cours. Deux façons de faire sont possibles: une première serait de donner la liste exhaustive de tous ces problèmes. Si cette façon de faire a l'avantage d'être précise, elle a le gros désavantage suivant: un langage d'expression d'assertions basé sur l'expression d'une liste précise de problèmes risquerait d'être inadéquat face à des problèmes du même style mais en dehors de cette liste. Or, de nouveaux problèmes seront probablement envisagés dans ce cours, notamment lors des différents examens. Cette liste risque donc de s'étoffer d'année

---

(☆) C'est d'ailleurs pourquoi, par la suite, on utilisera souvent la périphrase "programmation d'assertions" plutôt que "expression d'assertions".

(☆☆) "Un bon langage de spécification, quel que soit son degré de formalisation, doit puiser ses concepts dans les applications elles-mêmes, et non dans une vue, aussi abstraite qu'on la suppose, de ce qu'il est possible de faire avec un langage de programmation". H. Leroy [2].

(☆☆☆) Ces remarques seront explicitées au chapitre III de ce mémoire.

en année. Remarquons ensuite qu'un tel langage ne pourrait se réduire à une liste de primitives exprimant les assertions correctes de chaque problème. En effet, le type de langage envisagé devrait permettre également l'expression d'assertions incorrectes (à la limite, n'ayant plus grand chose à voir avec le problème) puisqu'il a pour but de détecter des erreurs dans les assertions et non d'indiquer la correction de celles-ci (vérification à l'exécution). La seconde façon de cerner cette classe de problèmes est d'indiquer qu'on se limite à des concepts pouvant se représenter d'une manière relativement simple (avec peu de conventions de représentation) par les concepts élémentaires de tableaux et de variables simples. C'est la "véritable" limitation de cette classe, mais c'est assez vague. C 'est pourquoi on nuancera l'objectif de ce langage en disant qu'il s'agit de permettre "la programmation aussi aisée que possible" des assertions des problèmes de cette classe.

Chapitre II

ÉLABORATION D'UN LANGAGE PERMETTANT L'EXPRESSION D'ASSERTIONS

2.1. DEMARCHE SUIVIE

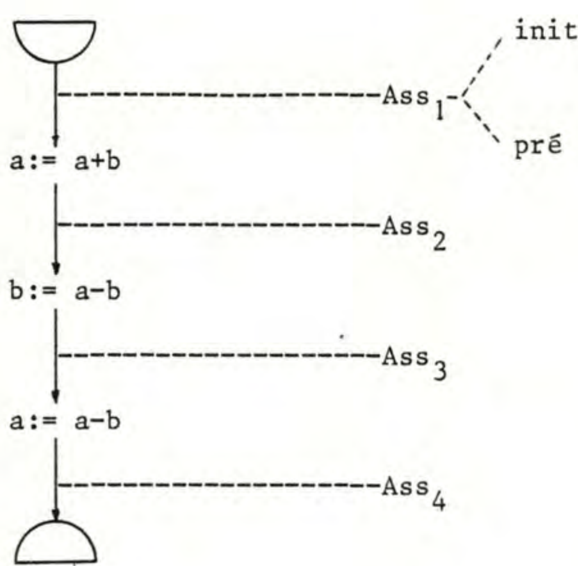
La fin du chapitre précédent a indiqué quel type d'outil nous allions proposer: un langage de programmation permettant l'expression d'assertions en vue de leur vérification à l'exécution. La définition d'un tel langage étant fastidieuse à lire, nous l'avons présentée en annexe de ce mémoire. Quant à ce chapitre, il se contentera de présenter une ébauche du langage en suivant une démarche particulière: nous considérerons progressivement des programmes de plus en plus compliqués et, au fur et à mesure, nous essayerons de suggérer l'utilité de différents concepts et opérations pour un langage du type considéré. Cette façon de faire me semble intéressante pour deux raisons: une première est qu'elle calque la démarche que nous avons suivie pour l'élaborer et la seconde est qu'elle souligne les prétentions limitées de ce langage en insistant sur la classe fort restreinte des problèmes envisagés.
La fin du chapitre présentera deux exemples plus compliqués pour indiquer que l'utilité de ces concepts et opérations ne se limite pas aux exemples présentés pour les introduire.

2.2. UN PROGRAMME SANS BOUCLES

Exemple: Permutation de deux valeurs entières.

Spécification: Si, initialement, les variables a et b ont une valeur entière, l'exécution du programme permute ces valeurs.

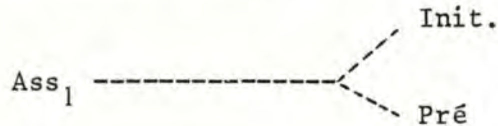
Organigramme:



Discussion...

a. Considérons la première partie de la spécification présentée ci-dessus; celle-ci est elle-même constituée de deux parties distinctes. Une première explicite le fait que les variables a et b "doivent avoir une valeur" et la seconde, que cette valeur doit être une valeur entière. On remarque que la partie d'une spécification mentionnant que certaines variables doivent avoir reçu une valeur est, bien qu'essentielle, souvent sous-entendue, faisant confiance au bon sens des personnes qui l'utilisent. Par contre, si l'on désire formaliser des spécifications en vue de leur vérification à l'exécution (par une machine), il est nécessaire de prévenir explicitement des situations n'ayant pas de sens telle l'évaluation d'assertions dont certaines variables n'ont pas reçu de valeur. Pour ce faire, l'outil proposé par notre formalisme est la fonction déf qui, étant donné une liste de variables ou tableaux, renvoie la valeur

de vérité vrai s'ils ont tous été initialisés et faux, sinon. L'emploi de la fonction <u>déf</u> peut être considéré comme la "préconidtion" universelle de la "programmation" d'une assertion (écrite en langage naturel, i.e. en dehors de tout formalisme) en une assertion écrite en un formalisme particulier. Ces constatations expliquent la raison pour laquelle nous dédoublerons systématiquement la première assertion de chaque programme en une première exprimant que certaines variables ont été initialisées, la seconde exprimant certaines propriétés que doivent vérifier les valeurs de ces variables.

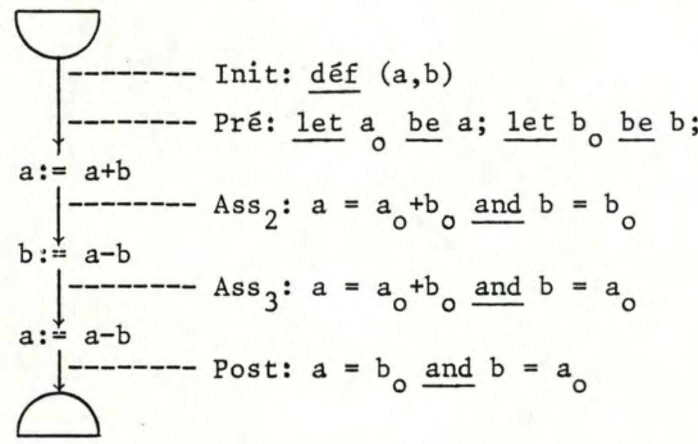


b. Il s'agit maintenant d'exprimer que les valeurs de a et de b doivent être entières; ce type de conditions est primordial: celles-ci sont notamment à la base du raisonnement fait pour construire le petit programme proposé. Leur place dans les assertions se justifierait donc. Cependant, comme le langage de programmation associé exige des déclarations typées de toutes les variables, exprimer ces conditions de type dans les assertions ferait double emploi avec ces dernières. C'est pourquoi les omettrons-nous volontairement par souci de concision.

c. Remarquons maintenant qu'une assertion quelconque, si elle se veut utile, doit souvent se référer à autre chose qu'aux valeurs courantes des variables; notamment aux valeurs qu'elles ont eues, au début ou en cours d'exécution. Ceci suggère l'utilité d'une "instruction" permettant leur mémorisation si nécessaire.

Outil proposé: le langage d'expression d'assertions permet une "instruction" dite de mémorisation. Les guillemets indiquent le caratère tout spécial de celle-ci. En effet, cette instruction n'a qu'un pouvoir limité; elle ne peut, bien sûr, pas modifier la valeur d'une variable du programme. Elle ne pourra donc porter que sur de nouvelles variables qui ne seront accessibles qu'à l'intérieur d'une assertion.

Ex: En vue d'établir la postcondition de l'exemple proposé, mémorisons, au début de l'exécution, les valeurs de a et de b. Nous sommes maintenant en mesure de compléter l'exemple:



Cet exemple très simple me semble indiquer que le langage d'expression d'assertions, s'il se veut utile, doit au moins permettre des expressions du même ordre de complexité que celles permises par le langage de programmation.

2.3. DEUX PROGRAMMES ELEMENTAIRES AVEC BOUCLES

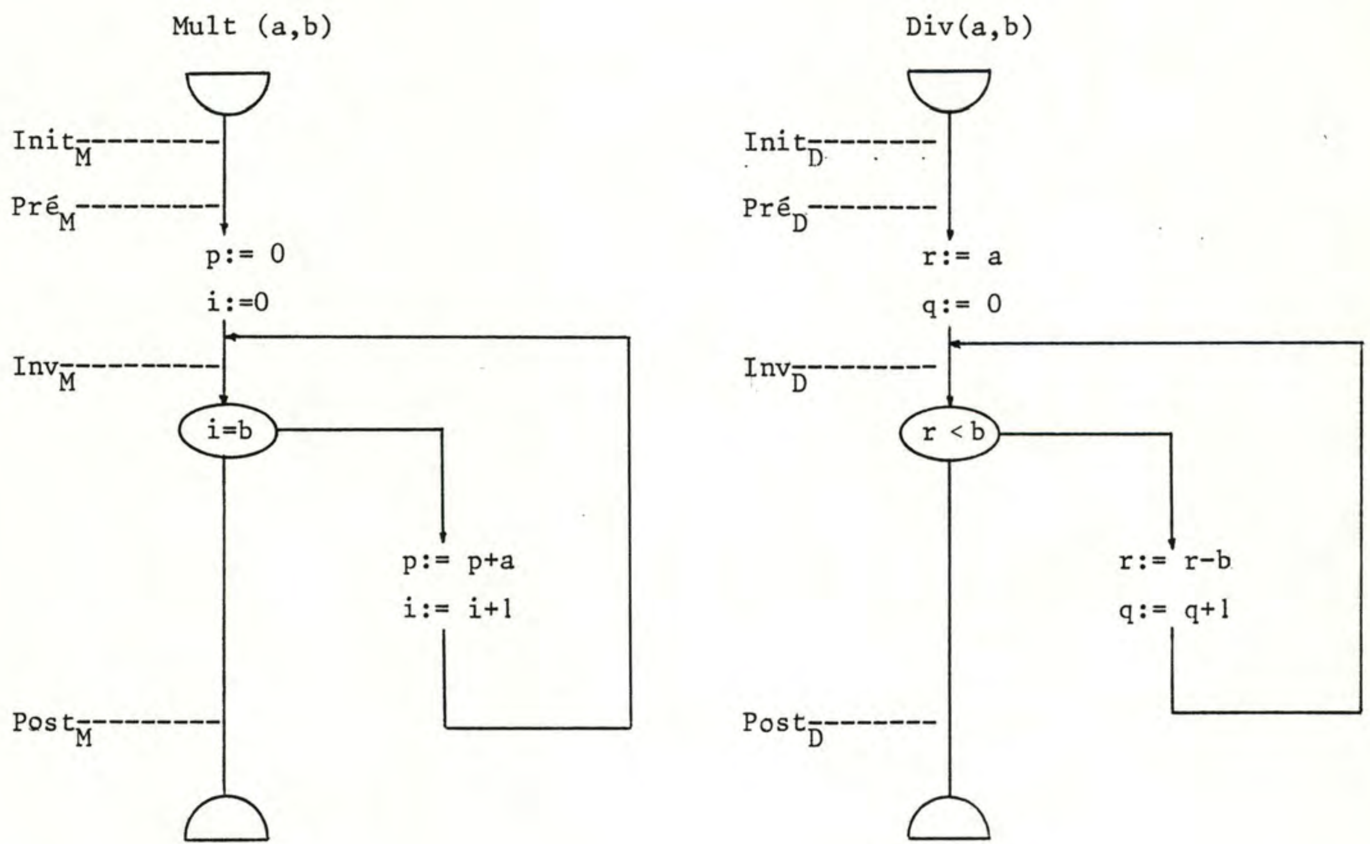
Exemples: Multiplication et division entière.

Spécifications :

Mult : Etant donné a et b, deux variables entières initialisées à des valeurs positives, affecter à la variable p la valeur du produit de a et b sans modifier leurs valeurs respectives.

Div : Etant donné deux variables entières, a, initialisée à une valeur positive et b, initialisée à une valeur strictement positive, affecter à la variable q la valeur de la division entière de a par b et à r, la valeur du reste de cette division, sans modifier ni a ni b.

Organigrammes :



Assertions

Mult	Div
$init_M$: Déf(a,b)	$init_D$: Déf(a,b)
$Pré_M$: Let a_o be a; Let b_o be b; $a_o \geqslant 0$ and $b_o \geqslant 0$	$Pré_D$: Let a_o be a; Let b_o be b; $a_o \geqslant 0$ and $b_o > 0$
Inv_M: $a = a_o$ and $b = b_o$ and $i \geqslant 0$ and $i \leqslant b$ and $p = i * a$	Inv_D: $a = a_o$ and $b = b_o$ and $r \geqslant 0$ and $a = q * b + r$
$Post_M$: $a = a_o$ and $b = b_o$ and $p = b * a$	$Post_D$: $a = a_o$ and $b = b_o$ and $q = a$ div b and $r = a$ mod b.

Discussion...

Si l'élaboration des assertions pour un programme sans boucle est un travail relativement facile en général, il n'en est plus de même lorsqu'il s'agit d'un programme avec boucles pour la simple et bonne raison qu'il nécessite un raisonnement par récurrence. Pour permettre l'expression aisée des assertions de tels programmes, nous allons voir que le formalisme envisagé se doit d'être plus riche que celui dans lequel est écrit le programme; il devrait, en fait, permettre un opérateur qui "englobe" ou "résume" la boucle. Dans les exemples proposés, remarquons que si les opérateurs *, div ou mod n'existaient pas comme opérateurs primitifs dans le langage de programmation (ce qui rendrait plus vraisemblable l'utilité de tels programmes), ils seraient bel et bien nécessaires dans celui permettant l'expresison des assertions de ces mêmes programmes. Pour s'en convaincre, il suffit de considérer le programme Mult en essayant d'exprimer ses assertions en ne permettant que les opérateurs + et - !... Certains rétorqueront, peut-être, que l'emploi d'un opérateur de sommation itérative, tel celui noté habituellement $\sum_{i=binf}^{bsup}$, permettrait de résoudre le problème: on formulerait sans doute l'invariant de la façon suivante:

$$Inv: a = a_o \ \underline{and} \ b = b_o \ \underline{and} \ i \geq 0 \ \underline{and} \ i \leq b$$

$$\underline{and} \quad p = \sum_{j=1}^{i} a$$

Mais, remarquons d'abord que l'opérateur utilisé est sûrement aussi complexe que l'opérateur $*$. De plus, le problème de la correction du programme serait, dès lors, reporté au problème de la correction de la formule

$$\sum_{j=1}^{i} a = i * a$$

qui est du même ordre de difficulté, bien qu'élémentaire, que le problème initial.

Il ressort de ce paragraphe que la définition d'un langage d'expression d'assertions doit se faire face aux primitives offertes par un langage de programmation fixé. Le langage de programmation choisi est un langage permettant la traduction aisée du formalisme des organigrammes du cours "Séminaire de programmation" en ce dernier. Il ressemble au Pascal tout en étant beaucoup moins riche. Il diffère principalement sur ces quelques points:

- restriction sur les types de base: seuls les entiers, les booléens et les caractères sont permis;
- seule structure permise: le tableau à une dimension;
- pas de récursivité;
- pas de variables globales;
- instructions de branchement différentes.

La définition d'un langage de programmation étant fastidieuse à lire, nous avons choisi de la présenter en annexe. Les quelques indications que nous venons de donner à son sujet suffiront pour la lecture de la suite de ce mémoire.

2.4. INTERET DES CONCEPTS DE SUITE ET D'ENSEMBLE

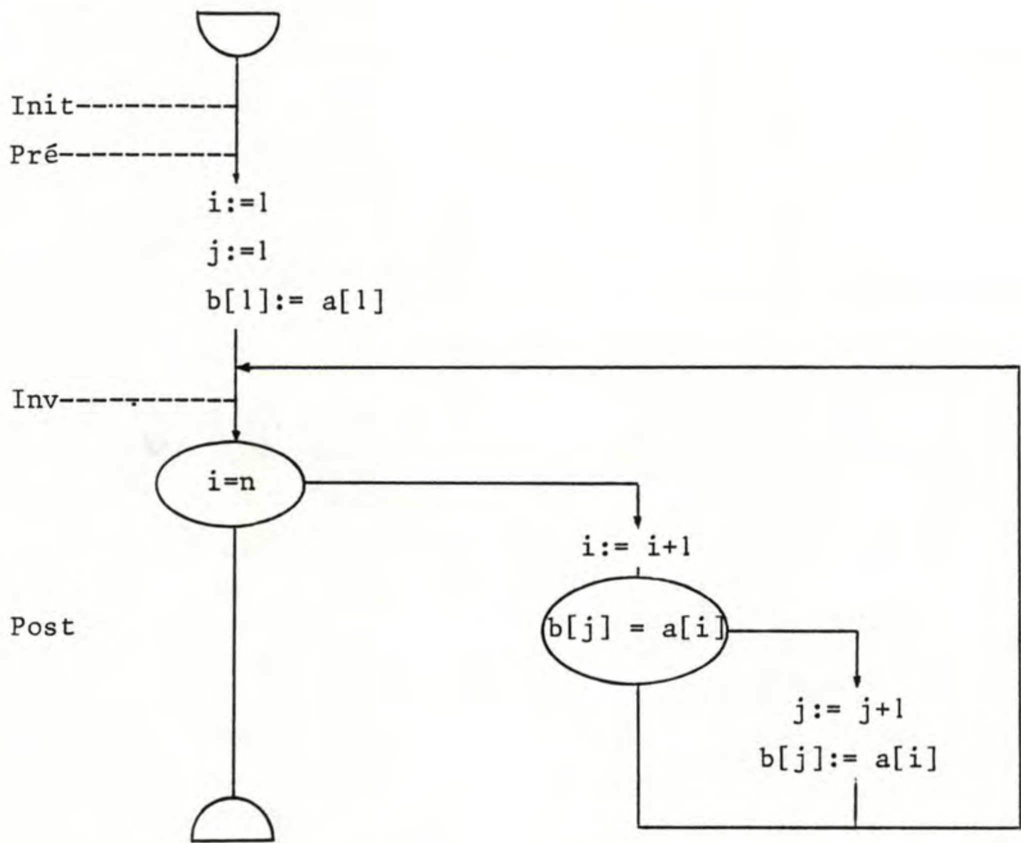
Le paragraphe précédent indique que le langage d'expression d'assertions devra permettre non seulement les opérations primitives du langage de programmation choisi mais aussi un ensemble d'opérations supplémentaires "bien choisies"; ces dernières devraient permettre d'une part, une "programmation" aisée de la majorité des assertions du cours concerné tout en restant, d'autre part, suffisamment générales pour être applicables à d'autres problèmes. Définir une primitive pour chaque problème aurait l'avantage considérable de permettre une programmation directe de certaines assertions, mais le langage résultant semblerait sans doute artificiel (☆) et risquerait d'être inadéquat si l'on devait résoudre des problèmes en dehors de cette classe. De plus, comme nous l'avons déjà indiqué, ce langage doit permettre la programmation d'assertions incorrectes, à la limite, n'ayant rien à voir avec le problème envisagé. En effet, le but de ce travail n'est pas de montrer la correction des assertions mais bien de détecter certaines erreurs !

Exemple: Contraction d'un vecteur d'entiers.

Spécification: Soit $a[1..n]$, $n \geq 1$, un vecteur d'entiers initialisé de sorte que $a[1] \leq ... a[i] \leq ... \leq a[n]$. Modifier le contenu de $b[1..n]$ de sorte que $b[1..j]$ où $1 \leq j \leq n$ contienne la suite des valeurs de $a[1..n]$ sans répétitions et soit trié par ordre croissant.

(☆) Exemple: permettre une opération calculant la longueur du plus long plateau d'un vecteur...

Organigramme



Assertions: (avec les notations du cours "Séminaire de programmation")

Init et Pré: $n \leqslant 1$ et a [1..n] vecteur initialisé tel que $a[1] \leqslant a[2] \leqslant \ldots \leqslant a[n]$.

Post: $1 \leqslant j \leqslant n$ et b [1..j] est tel que

a. $b[1] < b[2] \ldots < b[j]$ et que
b. l'ensemble des valeurs contenues dans b [1..j] soit égal à l'ensemble des valeurs contenues dans a [1..n] (non modifié).

Inv: $1 \leqslant j \leqslant i \leqslant n$ et b [1..j] est tel que

a. $b[1] < b[2] < \ldots < b[j]$ et que
b. l'ensemble des valeurs contenues dans b [1..j] soit égal à l'ensemble de celles contenues dans a [1..i] (a non modifié).

Discussion...

La "précondition" de l'exemple présenté ci-dessus est à nouveau divisible en deux parties, une première exprimant que le vecteur a est initialisé (dans notre langage, déf(a)) et une seconde disant que a est initialement trié par ordre croissant. Ce type de propriétés, qu'il s'agisse du tri croissant ou décroissant, strictement ou non, est très souvent utilisé. Elles peuvent être considérées comme des opérations à valeur booléenne dont l'argument serait, pour être suffisamment général, une suite de valeurs (appartenant à un type auquel est associé un ordre).

Considérons maintenant la partie de la postcondition exprimant que b[1..j] doit contenir toutes les valeurs de a et rien que celles-là. Cette propriété peut, à nouveau, être traduite par une opération à valeur booléenne qui aurait, pour arguments, deux ensembles de valeurs (celles du vecteur a et celles du vecteur b[1..j]) et qui renverrait la valeur vrai ssi ces deux ensembles sont égaux.

Ces deux remarques suggèrent l'utilité de nouveaux types de valeurs, générés à partir des types de base: il s'agit des suites et des ensembles d'entiers, de booléens ou de caractères.

Etant donné notre préoccupation, l'introduction de ces nouveaux types de valeurs comporte, à mon avis, certains avantages. Un premier serait qu'à force de manipuler si souvent ces concepts de suite et d'ensemble la majorité des informations les ont intériorisés ainsi que leurs opérations "primitives" habituelles. Cet avantage peut sembler négligeable mais c'est notamment parce qu'ils sont intériorisés qu'ils interviennent dans beaucoup de raisonnements (notamment parmi ceux par récurrence). Le second avantage, nous le percevrons progressivement tout au long de ce chapitre: nous verrons qu'on peut très souvent "résumer" les boucles des programmes de la classe envisagée par des opérations sur des suites ou des ensembles.

Ces constatations nous amènent à nous pencher sur les problèmes suivants:

- Dans ce langage d'expression d'assertions, comment permettre la construction de suites et d'ensembles à partir de valeurs, de variables ou de tableaux du programme ?
- Quels types d'opérations permettre ?

Comme précondition de l'exemple envisagé, on doit exprimer que la suite des valeurs du vecteur a (rangées par ordre croissant de l'indice, c'est-à-dire a[1], a[2], ..., a[n]) est triée par ordre croissant. Cette suite dans le langage d'expression d'assertions sera notée [a]. La postcondition, quant à elle, considère l'ensemble des valeurs du vecteur a. Ce dernier sera noté {a}. Les crochets seront utilisés dans les notations désignant des suites et les accolades, dans celles désignant des ensembles.
Remarquons qu'on ne considère pas toujours un vecteur dans son entier. L'invariant exprime, par exemple, la propriété de croissance restreinte à une partie du vecteur b, celle correspondant à la suite contiguë des indices 1, 2, ..., j. Cette suite d'entiers sera notée [1..j] tandis que la suite b[1], ..., b[j] sera notée b[1..j] . Parallèlement, l'ensemble des entiers 1, 2, ..., j sera notée {1..j} et l'ensemble des valeurs b[1], b[2], ..., b[j], {b[1..j]}.

Revenons aux propriétés rencontrées dans les assertions de l'exemple considéré; la propriété de croissance d'une suite de valeurs (☆) sera exprimée dans notre formalisme par la fonction <u>tricrois</u> qui a un argument de type suite renvoie une valeur booléenne (<u>tristrcrois</u> pour l'ordre strict).
Un autre type de propriété était l'égalité ensembliste. Elle sera traduite par la fonction <u>égal</u> à deux arguments de type ensemble et à valeur booléenne. La fonction de même nom mais à arguments de type suite exprimera, quant à elle, l'égalité entre suites.

Nous pouvons dès lors écrire les assertions du problème envisagé:

(☆) Valeurs d'un type auquel est associée une relation d'ordre; s'il s'agit d'entiers, c'est la relation d'ordre habituelle et s'il s'agit de caractères, c'est la relation d'ordre explicitée dans l'annexe pour le type caractère.

Init: déf (a)

Pré: Let a_o be [a] ; tricrois (a_o)

Post: 1 ≤ j and j ≤ n and tristrcrois ([b [1..j]]) and
égal ({b [1..j]}, {a}) and égal ([a] , a_o)

Inv: 1 ≤ j and j ≤ i and i ≤ n and
tristrcrois ([b [1..j]]) and egal ({b [1..j]}, {a [1..i]})
and egal ([a] , a_o)

Il est quelquefois intéressant de construire une suite de valeurs en explicitant tous ces éléments. Par exemple, si a_1, ..., a_n sont des expressions arithmétiques, on notera [a_1, ..., a_n] la suite d'entiers à n éléments où le $i^{ème}$ de cette suite est la valeur de l'expression a_i. Cela nous permet notamment d'alléger la formulation de l'assertion Inv; celle-ci devient:

Inv: tricrois ([1,j,i,n]) and ...

Nous venons de voir comment traduire la propriété de croissance de suites; parallèlement, les fonctions tristrdéc et tridec traduiront celles exprimant leur décroissance (stricte ou non).

De plus, la propriété d'égalité ensembliste suggère une propriété moins forte qui est l'inclusion. La fonction inclus qui la traduira est aussi une fonction qui, ayant deux ensembles comme arguments, renvoie une valeur booléenne.

2.5. CONSTRUCTION D'EXPRESSIONS UTILISANT DES SUITES OU DES ENSEMBLES

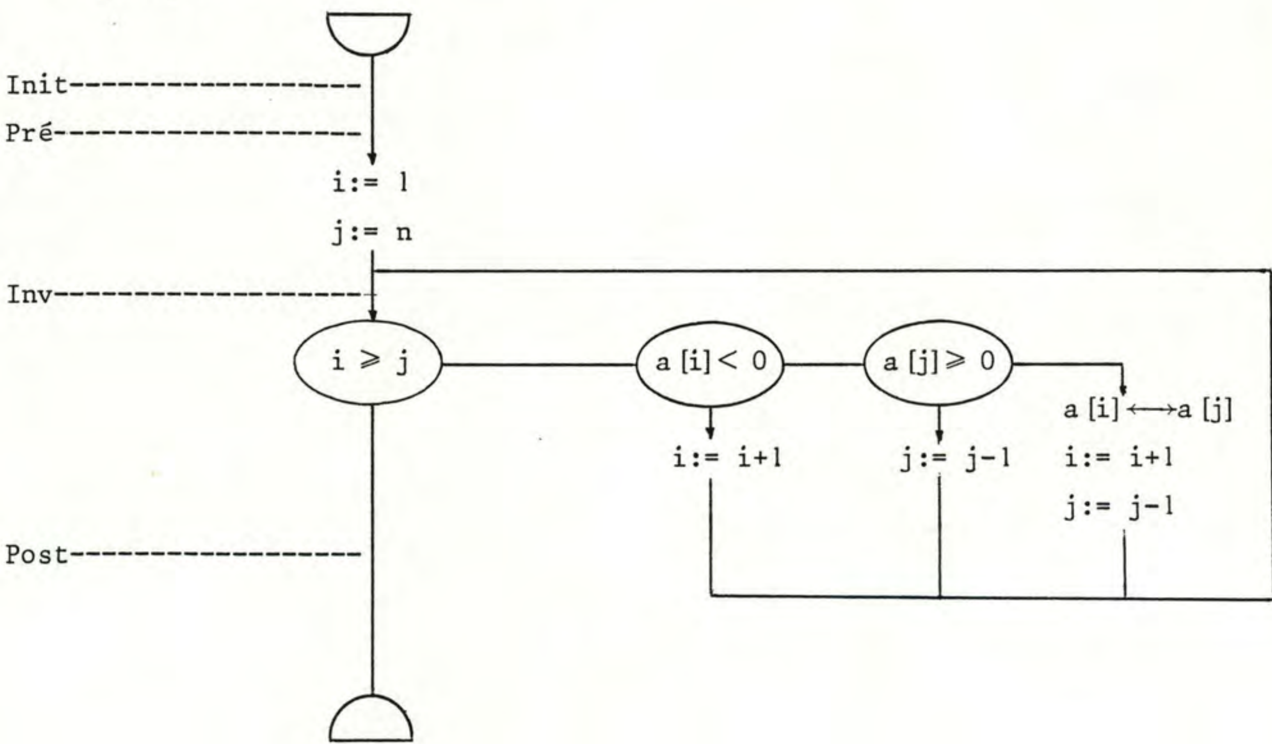
Comme nous l'avons signalé auparavant, une des qualités principales d'un langage du type considéré est de permettre une "programmation" aussi aisée que possible des assertions d'une classe de problèmes. Pour ce faire, nous avons d'abord introduit des types et des opérations n'existant pas dans le langage de programmation proprement dit. Le pas suivant consistera à permettre la construction d'expressions utilisant ces nouveaux types de valeurs.

2.5.1. Expressions booléennes avec "quantificateurs"

Exemple: Permutation d'un vecteur plaçant ses éléments strictement négatifs à gauche et les autres, à droite.

Spécification : Soit a [1..n] , un vecteur d'entiers initialisés. Permuter ce dernier de sorte que, à gauche se trouvent ses éléments négatifs et à droite ses éléments positifs ou nuls.

Organigramme



Assertions: (notations du cours "Séminaire de programmation").

Init et Pré: a [1..n] , vecteur initialisé.

Inv: $1 \leq i \leq j+1 \leq n+1$ et

a [1..n] permuté de sorte que

. $\forall$ k tel que $1 \leq k \leq i-1$: $a[k] < 0$

. $\forall$ q tel que $j+1 \leq q \leq n$: $a[q] \geq 0$

Post: a [1..n] permuté de sorte que

$\exists\, i_s$ tel que $1 \leq i_s \leq n$: ($\forall$ k tel que $1 \leq k \leq i_s-1$: $a[k] < 0$ $\land$
$\forall$ q tel que $i_s+1 \leq q \leq n$: $a[q] \geq 0$)

Discussion...

Jusqu'ici, les assertions envisagées ont été exprimées sous forme de proposition "atomique", négation, conjonction ou disjonction de propositions. Mais on remarque, notamment dans l'exemple envisagé, que les notations employées s'inspirent quelquefois des langages du premier ordre en introduisant des quantificateurs. Il serait donc intéressant que notre formalisme permette la construction d'expressions booléennes calquant ces notations: les quantificateurs, existentiel et universel, seront traduits respectivement par les périphrases

"There exists ... in ... :" et
"For all ... in ... :"

Par exemple, la dernière partie de l'invariant Inv s'écrira:

Inv: ... and for all q in {j+1 .. n} : (a [q] $\geq$ 0)

Nouvel identificateur — (expression de type) ensemble — expression booléenne qui peut contenir l'identificateur

Pour compléter l'invariant, il nous faut encore traduire "a permuté..." exprimant qu'on a modifié ce vecteur en n'effectuant que des permutations de l'ordre de ses éléments. Notons a_i le vecteur initial, a désignant le vecteur en cours d'exécution. Pour exprimer la propriété concernée, on serait tenté de dire que les deux vecteurs a_i et a doivent avoir même nombre

d'éléments et que toutes les valeurs de l'un doivent se retrouver dans l'autre et vice versa, c'est-à-dire, dans notre formalisme, _égal_ ({a},{a_i}). Remarquons que cette propriété est bien une condition nécessaire mais pas suffisante pour que a soit une permutation de a_i. En effet, le vecteur (1,1,3) n'est pas une permutation du vecteur (3,3,1) alors qu'ils ont même nombre d'éléments et que leurs ensembles de valeurs sont égaux. On pourrait cependant rajouter des conditions et trouver la formulation (☆) d'une propriété ne faisant appel qu'à des opérations sur des ensembles qui soit équivalente à celle exprimant qu'une suite est une permutation d'une autre. Mais il me semble qu'étant donné la complexité de cette formulation (☆) et la fréquence de ce style de propriété, il est intéressant d'introduire une opération qui la traduit. Celle-ci, notée _permut_, sera donc une opération qui, étant donné deux suites, renvoie la valeur de vérité vrai ssi ces deux suites sont permutation l'une de l'autre, c'est-à-dire ssi toutes les valeurs se trouvant dans l'une figurent autant de fois que dans l'autre et vice versa.

Nous sommes maintenant en mesure de compléter l'exemple:

Init: _déf_ (a)

Pré: _Let_ a_o _be_ [a] .

Inv: _Tricrois_ ([1,i,j+1,n+1]) _and_ _permut_ ([a] , a_o)
  and _for_ _all_ k _in_ {1..i-1} : (a [k] < 0)
  and _for_ _all_ q _in_ {j+1..n} : (a [q] ⩾ 0)

Post: _Permut_ ([a] , a_o) _and_
  There _exists_ i_s _in_ {1..n} : (_for_ _all_ k _in_ {1..i_s-1} : (a [k] < 0)
    and _for_ _all_ q _in_ {i_s+1..n} : (a [q] ⩾ 0))

Remarquons finalement que les quantificateurs introduits ci-dessus expriment, en fait, les opérateurs "itératifs" associés aux deux opérateurs de base ∧ et ∨ . Prenons, par exemple, la proposition

$$\forall k \in \{1..i\} : a[k] < 0.$$

(☆) Activité très proche de la programmation; tout autant sujette à l'erreur !

Elle peut être exprimée sous la forme :

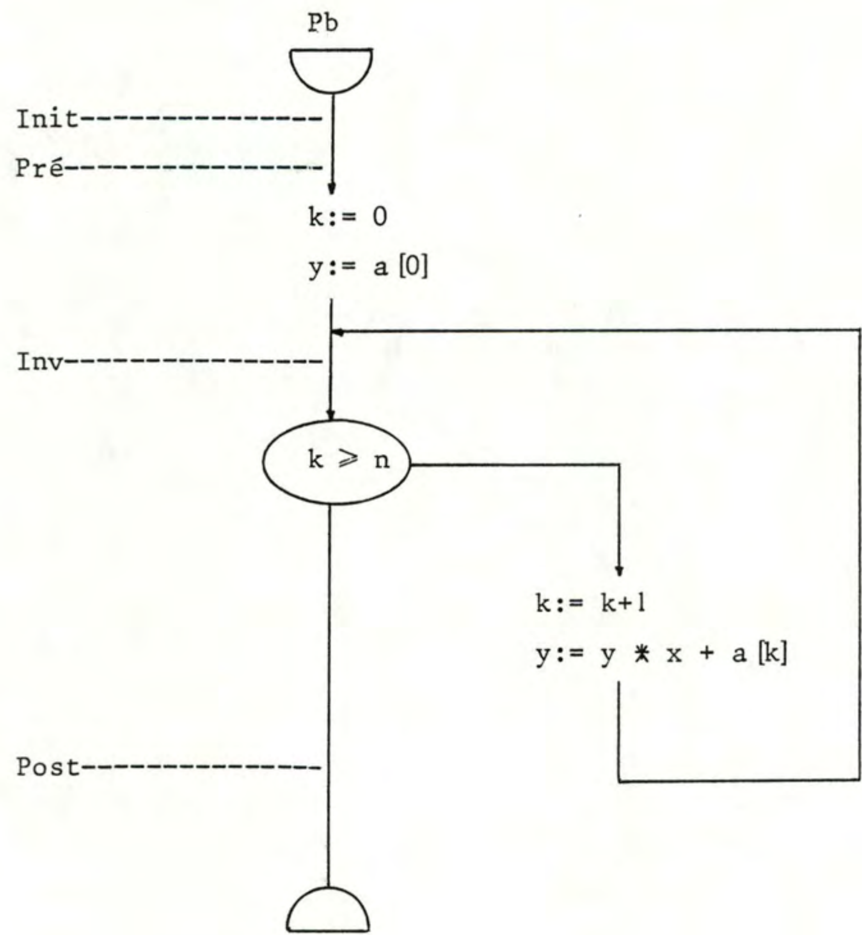
$$\bigwedge_{k=1}^{i} (a[k] < 0)$$

2.5.2. Opérateurs arithmétiques "itératifs"

Exemple : Calcul de la valeur d'un polynôme.

Spécification: Etant donné un vecteur d'entiers a[0..n] ($n \geq 0$) et une variable entière x, tous deux initialisés, affecter à la variable y, la valeur du polynôme de degré n (qui a pour coefficients les éléments de a) évalué en la valeur de x.

Organigramme



Assertions

Init et Pré: $n \geq 0$, a [0..n] et x initialisés

Inv: x,n,a [0..n] inchangés et

$$0 \leq k \leq n \quad \text{et} \quad y = \sum_{i=o}^{k} a[i]\, x^{k-i}$$

Post: x,n,a [0..n] inchangés et

$$y = \sum_{i=o}^{n} a[i]\, x^{n-i}$$

D'une manière semblable au paragraphe précédent, nous allons introduire des opérateurs "itératifs" associés aux opérateurs arithmétiques habituels. C'est ainsi que

$$\sum_{i=m}^{n} exp(i)$$ où exp(i) est une expression arithmétique dépendant de i

se notera

<u>Sum</u> <u>for</u> i <u>in</u> [m..n] of (exp(i))

(i : Nouvel identificateur ; [m..n] : (expression de type) suite)

De même,

$$\prod_{i=n}^{n} exp(i)$$ s'écrira ----------------→ <u>prod</u> <u>for</u> i <u>in</u> [m..n] <u>of</u> (exp(i))

$$\max_{m \leq i \leq n} exp(i)$$ ----------------→ <u>Max</u> <u>for</u> i <u>in</u> {m..n} <u>of</u> (exp(i))

$$\min_{m \leq i \leq n} exp(i)$$ ----------------→ <u>Min</u> <u>for</u> i <u>in</u> {m..n} <u>of</u> (exp(i))

L'invariant considéré devient:

Inv: ... <u>and</u> <u>tricrois</u> ([0,k,n]) <u>and</u>
y = <u>sum</u> <u>for</u> i <u>in</u> [0..k] <u>of</u> (a [i] * x^{k-i})

2.5.3. Le domaine de référence d'une expression

Le squelette général d'une expression avec un opérateur "itératif" en tête est le suivant:

Sum for Prod for Max for Min for There exists For all	<identificateur> in A...

où A, que ce soit une suite ou un ensemble, est appelé le domaine de référence de l'expression concernée.

Remarque: Les domaines de référence des expressions arithmétiques max (min) for ... in ... et des expressions booléennes there exists (for all) ... in ... sont des ensembles tandis que pour les deux autres types d'expressions arithmétiques, il s'agit de suites.

Pour expliquer les raisons de ce choix, prenons un exemple simple impliquant tour à tour le maximum et la somme de deux suites de valeurs: Max(2,2,1) est bien égal à max(2,1), tandis que la somme des nombres 2,2,1 est naturellement différente de celle des nombres 2,1. Cet exemple indique que le fait de considérer l'ensemble des valeurs d'une suite plutôt que la suite elle-même constitue une perte d'information qui n'est pas significative si l'on travaille avec l'opérateur max mais qui l'est si l'on travaille avec l'opérateur de sommation.

Considérons maintenant ces mêmes opérations en les "indiçant". Par exemple, il peut être intéressant de pouvoir exprimer le maximum et la somme des carrés des éléments d'une suite S. Si ce maximum peut être traduit par

max for x in {S} of (x * x)

(car le fait qu'un élément figure une ou plusieurs fois dans la suite n'influence pas le résultat), le fait de considérer la somme indicée sur l'ensemble des valeurs de la suite S aurait pour

conséquence d'ignorer le nombre de fois que figure chaque élément dans S et influencerait donc le résultat. C'est donc, finalement, pour faciliter l'expression de certaines opérations que nous avons choisi des suites et non des ensembles comme domaine de référence des expressions sum for ... in ... et prod for ... in ...

2.5.4. Conditions limites

L'introduction des opérateurs "itératifs" nous amène à nous poser la question suivante: Quel sens donner à des expressions dont le domaine de référence est vide ? Nous allons voir qu'il est intéressant de prendre l'élément neutre de l'opération dont l'opérateur en question traduit l'emploi itératif. Le tableau ci-dessous résume cette idée pour les différents opérateurs envisagés précédemment.

Opérateur "itératif"	Opération associée	Elément neutre
Sum for ...	+(.,.)	0
Prod for ...	*(.,.)	1
Min for ...	min(.,.)	$+\infty$
Max for ...	max(.,.)	$-\infty$
for all ...	$\wedge$ (.,.)	vrai
There exists ...	$\vee$ (.,.)	faux

Tâchons maintenant de voir les motivations sous-jacentes à ce choix. Considérons pour cela la somme et utilisons, par souci de concision, les notations " $\sum_{i=m}^{n}$ ..." habituelles plutôt que les périphrases du formalisme. Supposons que l'on ait comme invariant d'un programme:

$$m \leqslant k \leqslant n \wedge y = \sum_{i=m}^{k-1} exp(i)$$ où exp(i) est une expression arithmétique dépendant de i.

La propriété utilisée dans la démonstration (☆) d'un tel programme sera

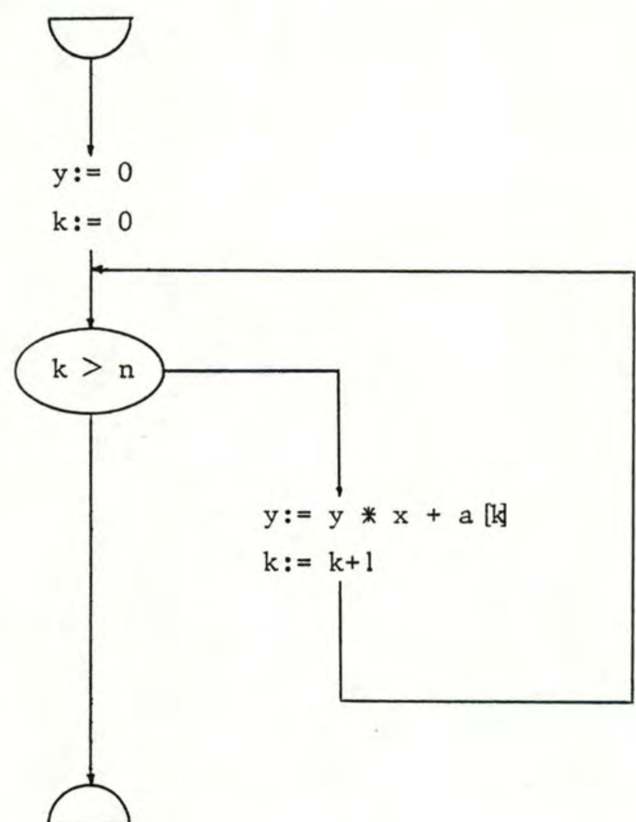
(☆) et par la même occasion, base de la construction du programme.

vraisemblablement

$$\sum_{i=m}^{k} exp(i) = \sum_{i=m}^{k-1} exp(i) + exp(k) \qquad \text{pour } m < k \leqslant n$$

Le fait de prendre comme convention que $\sum_{i=m}^{m-1} exp(i)$ vaut 0 est une généralisation "esthétique" de la sémantique de la notation " $\sum_{i=m}^{n} exp(i)$" puisqu'elle permet d'étendre cette propriété à $m \leqslant k \leqslant n$. A première vue, cela peut sembler négligeable face à notre préoccupation. Précisons cependant cette idée sur un exemple; considérons à nouveau le problème du calcul de la valeur d'un polynôme mais traitons-le différemment.

Organigramme



Si l'on prend les conventions explicitées plus haut, l'invariant peut s'écrire: $0 \leqslant k \leqslant n+1 \wedge y = \sum_{i=o}^{k-1} a(i)\, x^{k-1-i}$ sans distinguer le premier passage des suivants et la propriété utilisée dans sa démonstration sera vraisem-

blablement:

$$\sum_{i=o}^{k} a(i)\ x^{k-i} = (\sum_{i=o}^{k-1} a(i)\ x^{k-i-1}) * x + a(k)$$

$$= \sum_{i=o}^{k-1} a(i)\ x^{k-i} + a[k] * x^{k-k}$$

et sera également utilisée en k=0.

Remarquons finalement que ce dernier organigramme est plus général que le précédent puisqu'il permet d'avoir $n < 0$ (i.e. a vecteur vide) en renvoyant dans ce cas la valeur 0. On admettra facilement que cette convention (*) (à expliciter dans les spécifications du problème) est fortement liée à celle qui consiste à donner la valeur nulle à l'expression $\sum_{i=o}^{-1} exp(i)$. Ceci souligne à nouveau le caractère algorithmique de ce type de notations.

2.5.5. Traitement des infinis

Nous avons vu au paragraphe précédent qu'il est intéressant de donner les valeurs $+\infty$ et $-\infty$ à des expressions arithmétiques dont l'opérateur itératif est <u>min</u> ou <u>max</u> et dont le domaine de référence est vide. Pour simuler ces notions d'infini positif et négatif, nous allons introduire dans notre formalisme deux nouvelles valeurs, zmax et zmin, et généraliser certaines opérations habituelles en permettant à leurs arguments d'être l'une de ces deux valeurs. Voyons d'abord l'intérêt pratique de cette démarche.

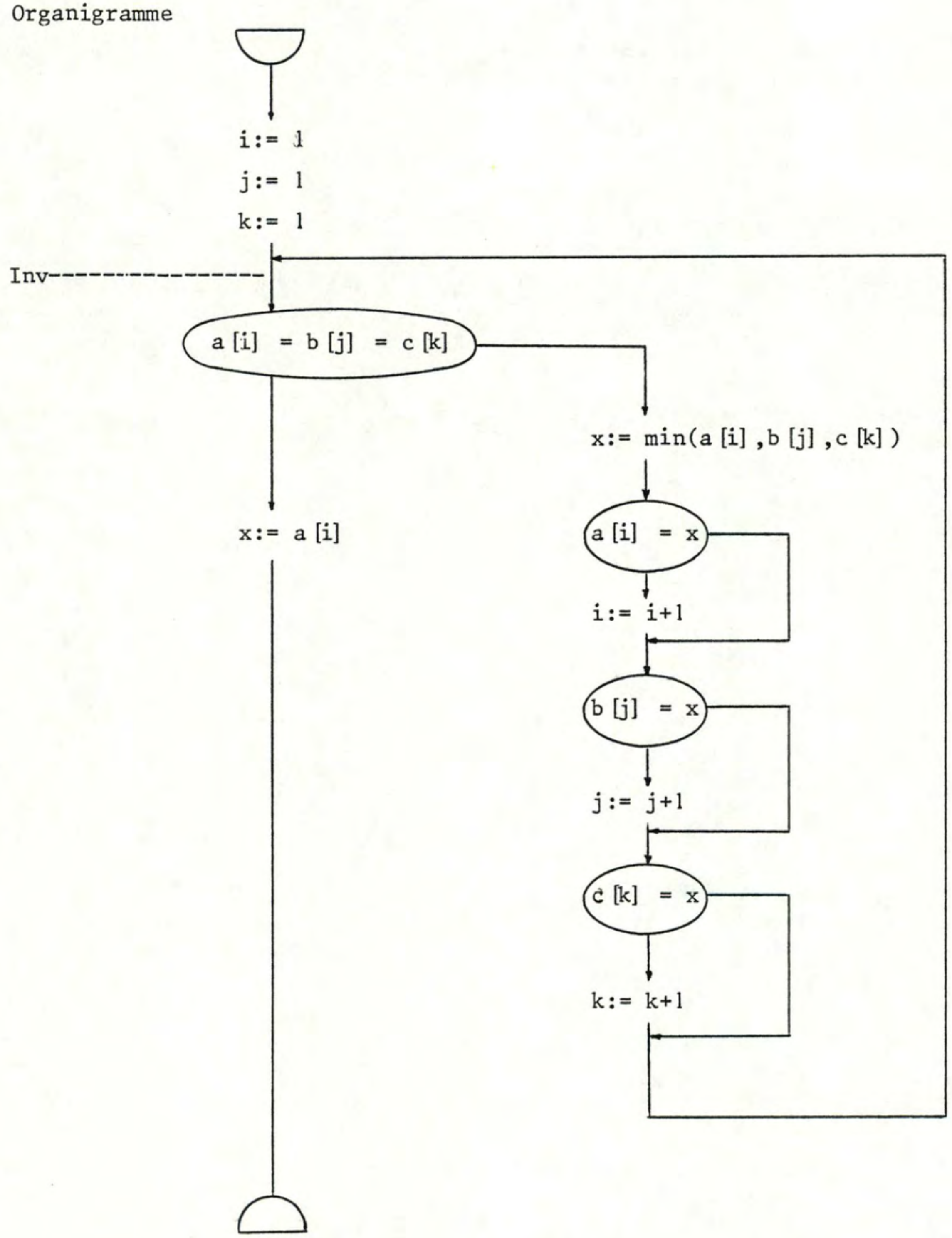
Exemple: Trouver la plus petite valeur commune à trois vecteurs d'entiers triés.

Spécification: Etant donné trois vecteurs d'entiers a, b et c initialement triés par ordre strictement croissant et qui ont au moins une valeur

(*) L'aspect "esthétique" apparaît ici sous une autre forme; en effet, une convention différente aurait impliqué l'ajout d'une "béquille" à l'organigramme, nuisant à son élégance relative...

commune, affecter à la variable x la plus petite d'entre elles sans modifier les trois vecteurs.

Organigramme



L'invariant Inv, avec les notations du cours, s'écrit:

$Inv \equiv 1 \leqslant i \leqslant m, \quad 1 \leqslant j \leqslant n, \quad 1 \leqslant k \leqslant p$ et

a. $a[1..i-1] \cap b[1..j-1] \cap c[1..k-1] = \emptyset$

b. $\max(a[1], ...a[i-1], b[1], ..b[j-1], c[1], ..c[k-1]) < \min(a[i], b[j], c[k])$

Considérons seulement la deuxième partie de cet invariant et exprimons-la dans notre formalisme:

Inv ≡ ... <u>and</u> <u>max</u> ({a [1..i-1] ,b [1..j-1] ,c [1..k-1] }) < <u>min</u>({a [i] ,b [j] ,c [k] })

Rem.: Soit A, un ensemble; <u>max</u>(A) est, dans notre formalisme, équivalent à l'expression <u>max</u> <u>for</u> p <u>in</u> A <u>of</u> (p).

Remarquons que cet invariant doit notamment être vérifié juste après les initialisations, c'est-à-dire lorsque les variables i, j et k valent 1. Dans ce cas, le domaine de référence du maximum est vide et donc, la valeur de cette expression arithmétique est zmin. Quant à l'évaluation de <u>min</u>({a [1] , b [1] , c [1] }), elle peut renvoyer n'importe quel entier suivant les valeurs initiales de a [1] , b [1] et c [1] . Nous devrions donc avoir (pour calquer la sémantique donnée aux notations) que

$$zmin < y \qquad \forall\, y, \text{ entier.}$$

Ceci confirme l'idée intuitive que l'on avait de zmin: "être l'élément neutre de max(.,.)" ou encore "être plus petit que tous les entiers".

D'une manière plus générale, rappelons-nous que les valeurs zmin et zmax ont été introduites pour pouvoir généraliser des propriétés telles

$$\begin{cases} \max(A) \leqslant \max(A \cup \{v\}) \\ \min(A) \geqslant \min(A \cup \{v\}) \\ \max(A \cup \{v\}) = \max(\max(A), v) \\ \min(A \cup \{v\}) = \min(\min(A), v) \end{cases}$$

au cas où A est l'ensemble vide.

Ces propriétés nous suggèrent la généralisation des opérations de comparaison de la manière suivante:

Table de l'inégalité $<$:

$<$	y	zmin	zmax
x	$(x < y)$	faux	vrai
zmin	vrai	///	vrai
zmax	faux	faux	///

Table de l'égalité :

=	y	zmin	zmax
x	$(x = y)$	faux	faux
zmin	faux	///	faux
zmax	faux	faux	///

où x et y ne sont ni zmin, ni zmax, et où les traits hachurés signifient que l'opération de comparaison n'est pas définie pour les valeurs correspondantes de la table.

Les tables des autres opérations de comparaison se construisent de façon analogue.

Nous avons choisi de ne pas définir les opérations de comparaison pour deux valeurs zmin ou deux valeurs zmax; le choix contraire, quelles que soient les conventions choisies, nous a semblé aller à l'encontre des notions intuitives traduites par ces valeurs. De plus, il serait malsain d'utiliser de telles conventions dans des raisonnements.

Venant de permettre les valeurs zmin et zmax comme arguments des opérations de comparaison, le pas suivant serait de permettre ces mêmes valeurs comme arguments des opérations habituelles, l'addition, la multiplication, le reste et la division entière, mais ce pas nous a semblé trop artificiel: remarquons d'abord qu'autant les opérations min et max sont directement liées aux opérations de comparaison, autant elles n'ont guère de liens immédiats avec les opérations considérées. De plus, il n'est pas évident de trouver un exemple

suggérant l'emploi de ce type de généralisation qui ne soit, lui-même, trop artificiel. Comme, enfin, un des objectifs de ce travail est de fournir un langage facile à maîtriser, nous avons finalement décidé de ne pas permettre ces valeurs comme arguments des opérations arithmétiques. La présentation du langage est cependant telle que la généralisation de ces opérations se ferait très facilement et de façon fort locale: il suffirait d'alléger les préconditions de ces opérations et de fixer leur table pour ces valeurs.

Un dernier choix relatif à l'introduction des valeurs zmin et zmax est de décider si ces valeurs peuvent ou non figurer comme élément d'une suite ou d'un ensemble. Certains exemples semblent indiquer qu'il serait intéressant de le permettre mais, en général, on peut très facilement détourner le problème en trouvant une autre formulation de leurs assertions, fort proche, et n'utilisant pas ces valeurs dans les suites ou les ensembles. C'est à nouveau par souci de simplicité que nous avons choisi de ne pas permettre ces valeurs dans les suites ou les ensembles: cela compliquerait significativement la sémantique du langage pour de maigres avantages. Par exemple, considérer zmin ou zmax comme un élément "comme un autre" d'un ensemble n'aurait pas été cohérent avec le choix que nous avons fait à leur sujet concernant les opérations de comparaison .

2.5.6. Construction de suites et d'ensembles

La précondition de l'exemple présenté au paragraphe précédent n'a pas été traduite dans notre formalisme. Elle mentionnait notamment que l'intersection des trois ensembles correspondant aux valeurs de chacun des trois vecteurs était non vide. Notre formalisme, pour ce genre de propriété, propose trois fonctions prédéfinies, inter, union et differ, qui traduisent respectivement l'intersection, l'union et la différence ensembliste. C'est ainsi que l'ensemble des valeurs communes aux trois vecteurs a, b et c peut s'écrire

inter ({a}, inter ({b}, {c}))

Dans une même optique, notre formalisme permet la construction de suites au moyen de quelques fonctions prédéfinies, la plus intéressante d'entre elles étant, sans doute, concat qui traduit la concaténation de deux suites.

Mais, ce type de construction d'ensembles ou de suites ne suffit pas toujours; comme l'indique l'exemple qui suit, il est quelquefois intéressant de pouvoir exprimer la partie d'un ensemble ou d'une suite qui vérifie telle ou telle propriété.

Exemple.

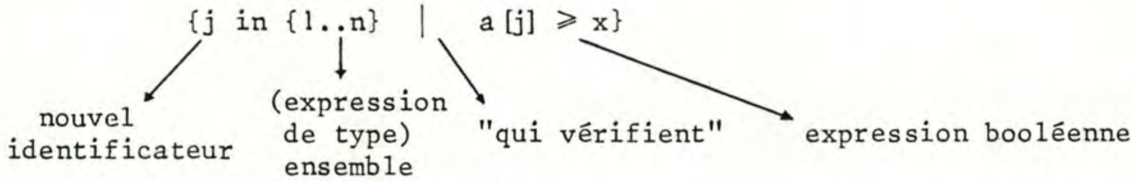
Spécification: Soient $a[1..n]$, un vecteur d'entiers trié par ordre croissant ($n \geqslant 0$) et

x, une variable entière possédant une valeur.

Il s'agit d'affecter à la variable i, le plus petit entier i_o tel que $1 \leqslant i_o \leqslant n$ et $a[i_o] \geqslant x$ s'il existe un tel entier et la valeur n+1 sinon.

Discussion...

Contentons-nous, pour cet exemple, d'exprimer les assertions notées habituellement "Pré" et "Post" correspondant, respectivement, à l'entrée et à la sortie de l'organigramme. Pour ce faire, remarquons d'abord que la spécification du problème distingue deux situations possibles. Pour les différencier, on peut, par exemple, considérer l'ensemble des indices du vecteur a qui sont tels que l'élément du vecteur correspondant soit supérieur à x et, ensuite, voir si cet ensemble est vide ou non. Cet ensemble s'exprime, dans notre formalisme, de la façon suivante:



De même, la sous-suite de la suite des indices vérifiant la même propriété s'écrira:

[j <u>in</u> [1..n] | a[j] ⩾ x]

↓

(expression de type) suite

Nous sommes maintenant en mesure d'écrire les assertions considérées:

(Init: déf(x,a)
Pré : Let données be [x,a] ;
tricrois ([a])
Post: Let ens be {j in {1..n} | a[j] ⩾ x};
égal (données, [x,a]) and
(vide (ens) ⟹ i = n+1) and (not vide (ens) ⟹ i = min(ens))

Remarques

a. La fonction vide est une fonction prédéfinie à valeur booléenne testant si un ensemble est vide.

b. Nous avons vu, au début de ce chapitre, l'intérêt de la pseudo-instruction let ... be ... . Nous l'utilisons ici dans une autre optique: rendre plus lisible l'expression d'une assertion.

c. Nous avons utilisé, dans cet exemple, le symbole "⟹" traduisant l'implication logique. Remarquons une des différences entre les opérateurs booléens de notre formalisme et ceux de la logique des propositions.
Dans la logique des propositions, la sémantique (table de vérité) de A ⇒ B est équivalente à celle de ¬ A ∨ B tandis que, dans notre formalisme, la sémantique de A ⇒ B est bel et bien différente de celle de not A or B. En effet, la sémantique de l'expression not A or B exige que l'évaluation de not A et de B soit déterminée; la sémantique de A ⇒ B exige aussi que l'évaluation de A soit déterminée, mais n'exige que celle de B le soit uniquement si l'évaluation de A renvoie la valeur de vérité vrai. Cette nuance peut sembler négligeable mais voyons sur un exemple qu'au contraire, elle peut être fort utile.

Exemple:

Les résultats du problème envisagé doivent notamment vérifier la propriété suivante: si la valeur de la variable i est un des indices du vecteur a, alors l'élément du vecteur correspondant est supérieur ou égal à la valeur de la variable x. Tâchons de l'exprimer dans notre formalisme; la remarque faite précédemment indique qu'on ne peut traduire cela par

<u>not</u> <u>inclus</u> ({i},{1..n}) <u>or</u> a[i] ⩾ x

car l'évaluation de a[i] ⩾ x pour i valant n+1 est indéterminée, mais bien par

<u>inclus</u> ({i},{1..n}) ⟹ a[i] ⩾ x

L'implication que nous venons d'introduire permet donc l'expression de propriétés "partielles", c'est-à-dire qui n'ont de sens que pour certaines valeurs des variables.

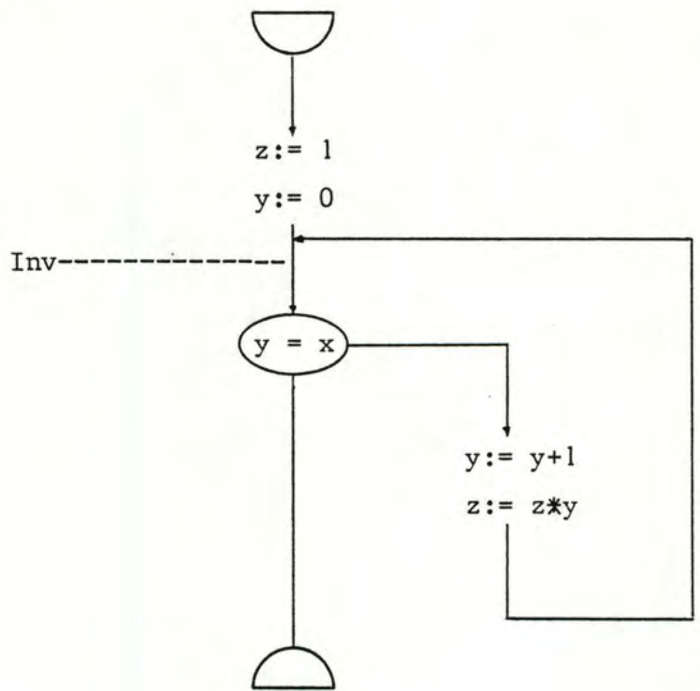
2.6. OPERATIONS DEFINIES PAR L'UTILISATEUR

Nous avons déjà indiqué que le travail consistant à exprimer des assertions dans un formalisme particulier nécessitait l'emploi d'opérations de "haut niveau" et de tous genres. En fait, ces opérations peuvent être aussi diverses que les problèmes envisagés. L'exemple présenté ci-dessous nous le confirme en montrant qu'il est quelquefois nécessaire d'utiliser, pour exprimer les assertions d'un programme, l'opération qui est justement l'objet du programme en question.

Exemple : La factorielle.

Spécification : Soit x, une variable entière ayant une valeur positive; affecter à la variable z la valeur x! sans modifier x.

Organigramme



Assertions

Inv: x non modifié et $0 \leqslant y \leqslant x$ et $z = y!$

Discussion...

Nous avons indiqué précédemment que nous ne suivrions pas la démarche consistant à permettre une opération prédéfinie pour chaque problème d'une classe énumérée exhaustivement. Il est donc nécessaire que l'utilisateur puisse programmer certaines opérations qu'il utilisera pour exprimer les assertions d'un programme. Certains se demanderont peut-être si l'objectif de ce mémoire n'est pas réduit à néant face à des programmes tels que celui de la factorielle, puisque l'utilisateur doit, pour exprimer ses assertions, programmer un problème de complexité égale à celle du problème initial et donc, tout autant sujet à l'erreur. Remarquons d'abord que ce type de problème est intrinsèque à l'objectif de notre travail. Ensuite, voyons que, même dans ce cas, si un des deux programmes contient une erreur, elle risque d'être détectée lors de la vérification des assertions à l'exécution. Sa localisation sera cependant plus difficile.

Il serait donc intéressant que notre formalisme d'expression d'assertions permette à l'utilisateur de programmer certaines opérations qu'il utilisera. Notre formalisme devient finalement un langage de programmation proprement dit, permettant notamment les mêmes instructions que le langage de programmation que l'on s'était fixé. Il est toutefois plus riche que ce dernier grâce à l'introduction des nouveaux types de valeur et grâce à la possibilité de construire des expressions plus complexes. Dans la suite de cet exposé, pour distinguer ces deux langages, nous dénommerons par L1 le langage de programmation initial et par L2, le langage d'expression d'assertions. La similitude entre ces deux noms souligne que, malgré l'objectif inhabituel du second, ils sont tous deux, avant tout, des langages de programmation.

En vue d'exprimer l'invariant du programme proposé, programmons, dans le langage L2, la fonction traduisant la factorielle:

```
function2 (*) fac (x: integer) : integer ;
  begin
    fac:= prod for i in [1..x] of i
  end;
```

(*) Le caractère "1" ou "2" apparaissant à la fin de certains mots indique s'il s'agit d'un concept respectivement, du langage L1 ou L2.

On peut dès lors exprimer l'invariant de la manière suivante (en supposant que x a été mémorisé, à l'entrée, dans la variable x_o):

Inv: $x = x_o$ and tricrois ([1,y,x])and z = fac(y)

Il est vrai qu'on aurait pu directement utiliser la formulation "prod for i in [1..x] of i" dans l'invariant mais ceci est dû au caractère simple du problème envisagé. Il n'en est plus de même s'il s'agit, par exemple, d'exprimer la longueur du plus petit plateau d'un vecteur...

Jusqu'à présent, le formalisme L2 permettait la construction d'ensembles ou de suites en explicitant leurs valeurs. Devenu un langage de programmation, il est intéressant d'y introduire les fonctions primitives "inverses" qui, à partir d'un ensemble ou d'une suite permettent d'accéder à leurs éléments: la fonction élt appliquée à un ensemble non vide renvoie un de ses éléments et les fonctions first et last, appliquées à une suite non vide, renvoient respectivement le premier et le dernier élément de cette suite. Supposons, par exemple, que l'on doive traduire la propriété exprimant que les éléments successifs d'une suite croissent d'une quantité fixée; on pourra programmer une fonction L2 la traduisant en considérant au fur et à mesure les éléments successifs de cette suite. Pour ce faire, on pourra, par exemple, utiliser itérativement les fonctions first et tail. Cette dernière, appliquée à une suite non vide, renvoie la suite suffixe qu' est la suite initiale tronquée de son premier élément.

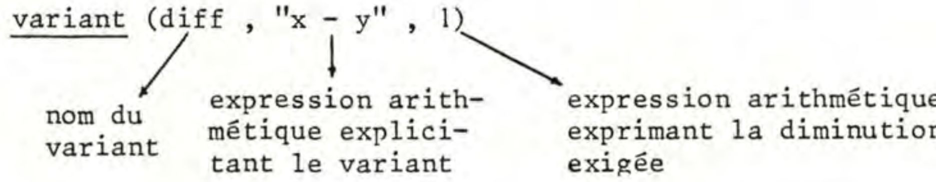
2.7. QUANT A LA TERMINAISON DES PROGRAMMES...

Le premier chapitre a exposé une méthode de démonstration de programmes appelée méthode de l'invariant. Rappelons-nous que cette méthode propose de démontrer la terminaison d'un programme en trouvant une fonction des variables qui soit bornée inférieurement et qui décroisse strictement à chaque itération. On a également indiqué dans ce premier chapitre qu'on pouvait se restreindre, sans perte de généralité, à des fonctions entières, strictement décroissantes et bornées inférieurement par zéro. Ce type de fonctions, utilisées dans la démonstration de terminaison d'une boucle, sera appelé le variant de cette boucle.

Prenons à nouveau l'exemple de la factorielle proposé au paragraphe précédent et voyons que la quantité x-y est bornée inférieurement par zéro et décroît, à chaque tour de boucle, d'au moins une unité. Elle peut donc faire office de variant. Le type de vérification que l'on pourrait faire à l'exécution est d'une part, vérifier que cette quantité x-y n'est jamais négative et d'autre part, qu'elle diminue bien, à chaque itération, de la quantité fixée, en l'occurrence 1.

Pour permettre cette vérification, nous proposons à l'utilisateur de notre langage de procéder comme suit:

a. Déclarer avant la boucle, le variant de celle-ci en lui donnant un nom, en l'explicitant (par une expression arithmétique) et en indiquant la quantité dont il est censé décroître à chaque itération. Cette déclaration se fait au moyen de la pseudo-instruction <u>variant</u>. Par exemple, la déclaration du variant du problème de la factorielle s'écrit:

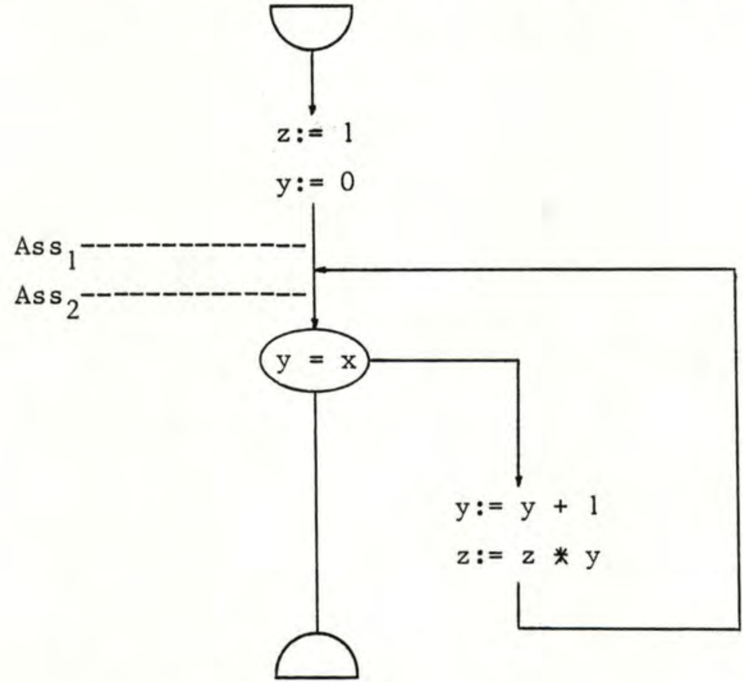


b. Indiquer, à un endroit dans la boucle, que l'évaluation de ce variant va décroître à chaque itération. Pour ce faire, nous utiliserons la fonction <u>term</u> en indiquant le variant auquel elle se réfère. Dans

notre exemple, nous l'exprimerons par

<u>term</u> (diff)

Cette fonction renvoie la valeur booléenne vrai si son évaluation suit une "exécution" de la pseudo-instruction déclarant ce variant (sans autre évaluation de <u>term</u> (diff) entre) (☆) ou si elle suit une autre évaluation de <u>term</u> (diff) et que la valeur du variant a diminué d'au moins la quantité fixée dans la pseudo-instruction en question, en l'occurrence 1. Considérons, à nouveau, l'exemple proposé et exprimons, partiellement, les assertions qui nous intéressent:



Ass_1: ... <u>variant</u> (diff, "x - y", 1); ...

Ass_2: ... <u>and</u> <u>term</u> (diff) <u>and</u> ...

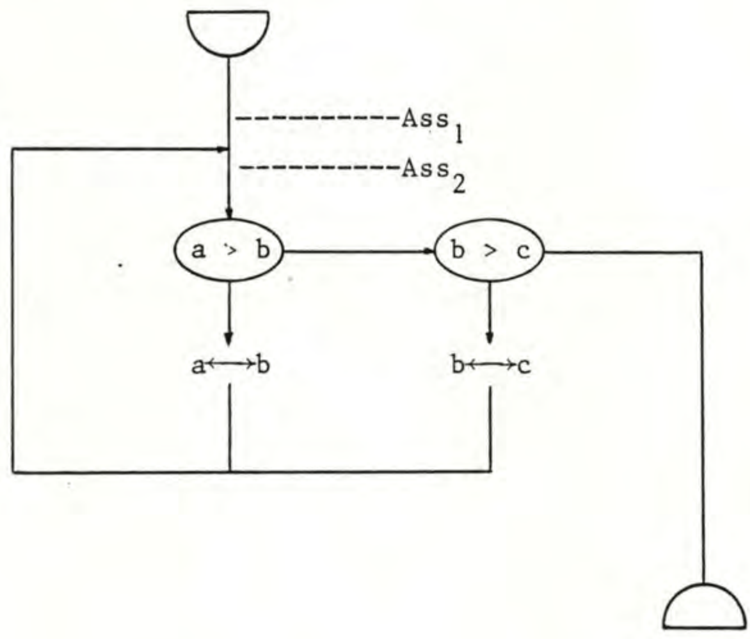
A chaque fois que Ass_2 sera évaluée, on évaluera donc la quantité x - y en vérifiant qu'elle est positive et qu'elle décroît d'au moins une unité d'une évaluation à l'autre.

La démonstration de correction partielle est souvent beaucoup plus compliquée que celle concernant la terminaison. Le variant est en fait,

---

(☆) La pseudo-instruction <u>variant</u> (...) joue un double rôle: la déclaration du variant <u>et</u> son "initialisation".

souvent fort simple. Ce n'est cependant pas toujours le cas: prenons, par exemple, le problème consistant à permuter les valeurs de trois variables entières, a, b et c, de sorte que a ⩽ b ⩽ c, et qui est résolu par l'organigramme suivant:



Pour ce problème, la correction partielle se démontre facilement: en effet, pour sortir de la boucle, il faut nécessairement que a ⩽ b et que b ⩽ c, les opérations internes à la boucle se limitant à des permutations de valeurs. La démonstration de la terminaison est moins évidente; voyons cependant que la fonction f(a,b,c) = 3 a + 2 b + c décroît strictement à chaque tour de la boucle. Remarquons ensuite qu'elle ne peut prendre qu'un nombre fini de valeurs puisqu'elle ne peut être évaluée qu'en 3! combinaisons des valeurs des variables. Elle est donc bornée inférieurement. La démonstration est ainsi terminée, mais, pour "formaliser" ce raisonnement dans notre langage, on est forcé de trouver explicitement une borne inférieure. On constate aisément que 6 min{a,b,c} peut convenir à cet effet, min {a,b,c} étant constant puisque, seules des opérations de permutation des valeurs des variables ne sont exécutées dans la boucle. Nous pouvons dès lors prendre comme

variant de la boucle

$$3\ a + 2\ b + c - 6 \min \{a,b,c\}$$

qui est bien borné inférieurement par zéro. De plus, min{a,b,c} étant constant, il aura la même propriété de décroissance que la fonction considérée initialement.
Traduisons cela dans notre formalisme:

Ass_1: ... <u>variant</u>(x, "3*a + 2*b + c - 6 *<u>min</u>({a,b,c})", 1); ...

Ass_2: ... <u>term</u>(x) ...

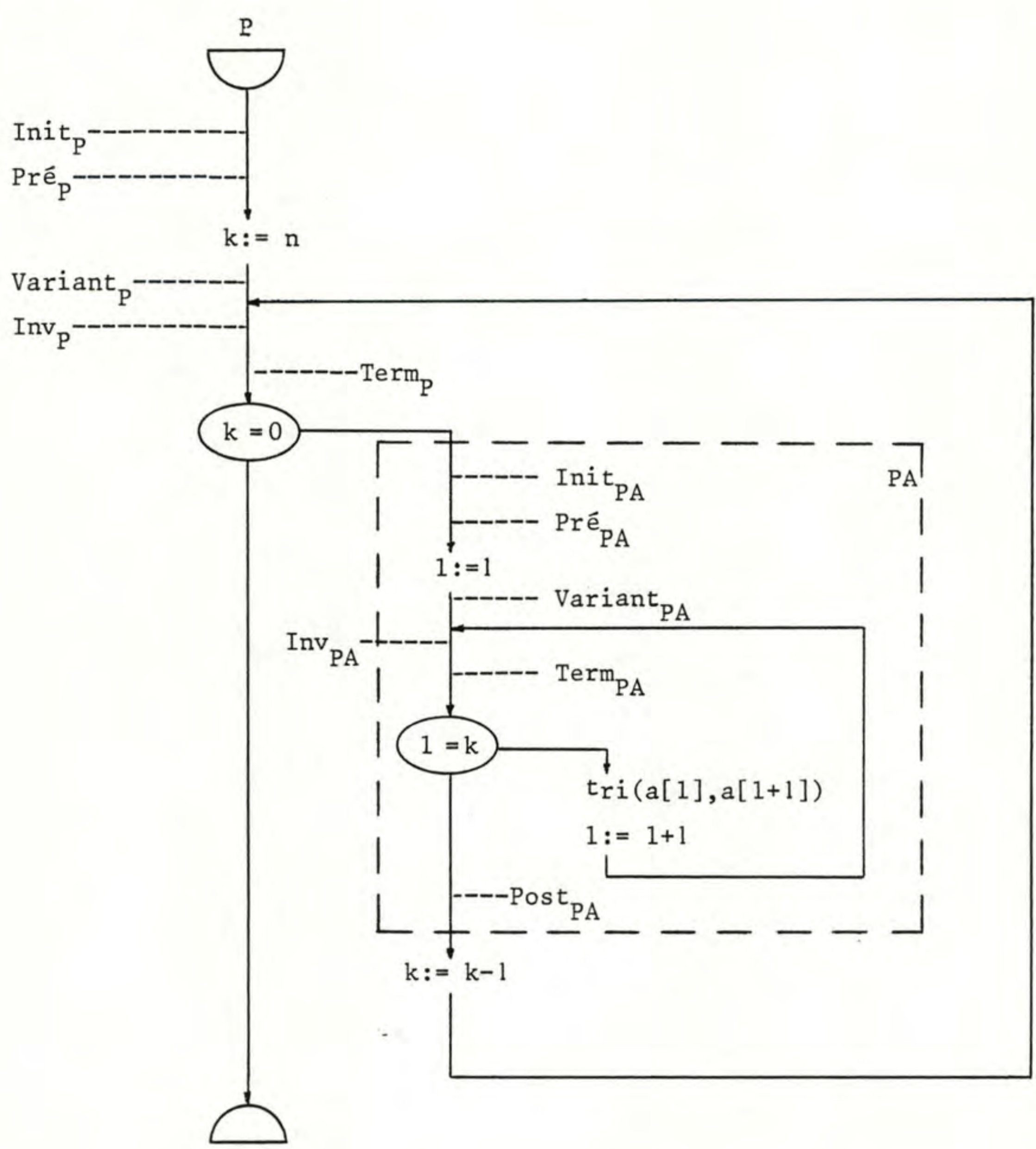
2.8. DEUX PROGRAMMES PLUS COMPLIQUES

En guise de conclusion de ce chapitre, il nous a semblé intéressant de présenter deux programmes légèrement plus compliqués. Le premier d'entre eux est fort représentatif de la classe envisagée: il s'agit du tri d'un vecteur.

Exemple: tri d'un vecteur d'entiers.

Spécification: L'exécution du programme a pour effet de trier par ordre croissant le vecteur d'entiers initialisé a[1..n].

Organigramme



où PA, la partie du programme délimitée par des pointillés, est considéré comme un programme auxiliaire et répond à la spécification suivante:

Spécification de PA: Etant donné une variable entière k comprise entre 1 et n et un vecteur d'entiers a[1..n] initialisé, l'exécution de PA consiste à effectuer une permutation du vecteur a de sorte que a[k] soit le maximum des valeurs de a[1..k] sans modifier ni k ni a [k+1..n] .

Invariants (avec les notations du cours "Séminaire de programmation").

Inv_P: $0 \leqslant k \leqslant n$ et a est permuté de sorte que

1. a[k+1..n] est trié et
2. $\forall$ i,j : $1 \leqslant i \leqslant k < j \leqslant n$: a[i] $\leqslant$ a[j]

Inv_{PA}: $1 \leqslant l \leqslant k$ et a est une permutation du contenu initial (i.e. au début de l'exécution de PA) de sorte que

1. a[l] = max a[1..l] et
2. a[l+1..n] et k n'ont pas été modifiés depuis le début de l'exécution de PA.

Terminaison

Programme P : k diminue d'une unité à chaque tour de boucle et est borné inférieurement par zéro.

Programme PA: k-l diminue d'une unité à chaque tour de boucle et est borné inférieurement par zéro.

Dans notre formalisme...

Programme P

> $Init_P$: déf(a)
> $Pré_P$: Let a_o be [a] ;
> $Post_P$: permut ([a] ,a_o) and tricrois ([a])
> $Variant_P$: variant(x,"k",1);
> Inv_P: tricrois([0,k,n]) and permut([a] ,a_o) and
> tricrois([a [k+1..n]]) and
> For all i in {1..k}: (for all j in {k+1..n}:(a [i] $\leqslant$ a [j]))
> $Term_P$:term(x)

Programme PA

> $Init_{PA}$: def(a,k)
> Pre_{PA} : Let [a_1] be a ; Let k_1 be k; tricrois([1,k,n])
> $Post_{PA}$: permut([a] ,a_1) and a [k] = max({a [1..k] }) and
> suffixe([a [k+1..n]] ,a_1) and k = k_1
> $Variant_{PA}$:variant(y,"k-1",1);
> Inv_{PA} : tricrois(1,1,k) and permut([a] ,a_1) and
> a [l] = max({a [l..l] }) and suffixe([a[l+1..n]] ,a_1) and k = k_1
> $Term_{PA}$: term(y)

Le second exemple est basé sur la notion de plateau d'une suite d'entiers. Cet exemple est intéressant car il nécessite la programmation, en L2, d'opérations qui seront utilisées dans les assertions. Nous verrons que la programmation de ces opérations reste difficile malgré les possibilités offertes par le langage L2. L'énoncé de cet exemple a été tiré de l'ouvrage [1].

Exemple: Le problème des plateaux.

Définition: Soit $S = (s_1,s_2,...,s_n)$ une suite finie, non vide, d'entiers ($n \geqslant 1$).

On appelle plateau de S tout intervalle [i:j] (☆) tel que

(☆) La notion d'intervalle est supposée connue.

(1) $1 \leq i \leq j \leq n$,

(2) $s_i = s_{i+1} = \ldots = s_j$

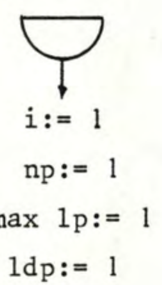
(3) [i:j] n'est pas strictement inclus dans un intervalle possédant les propriétés (1) et (2).

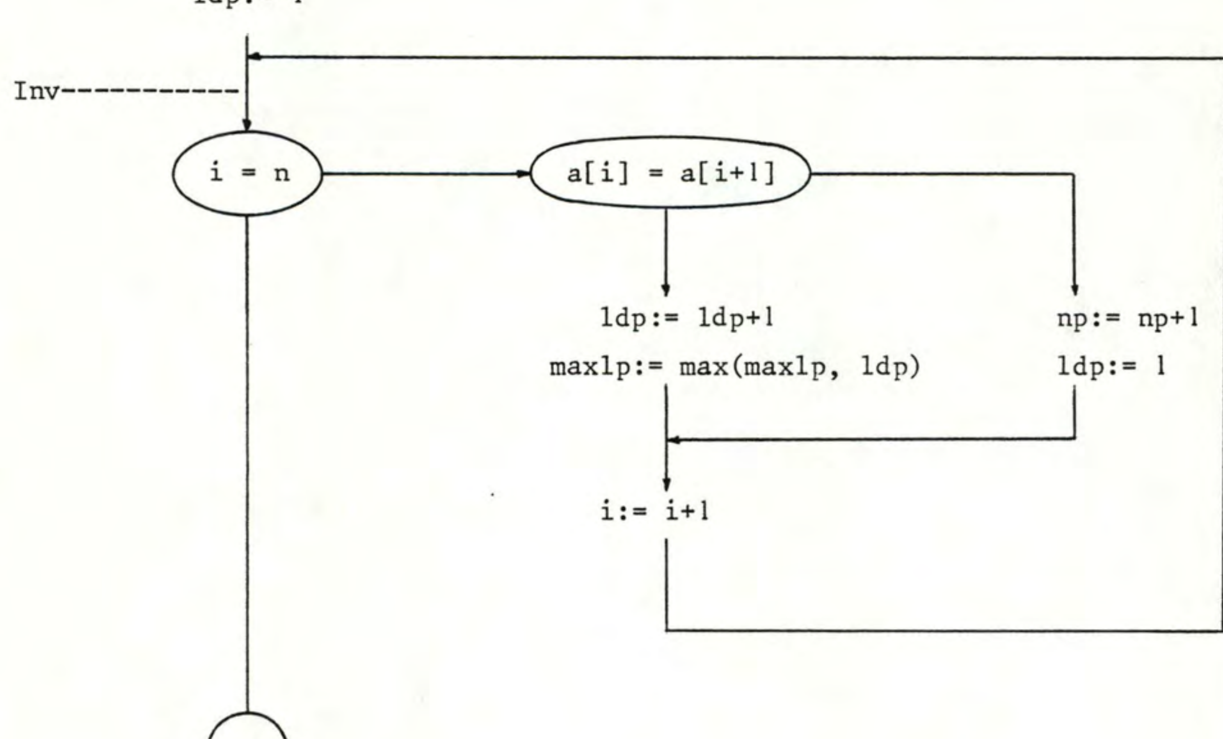
Spécification: On donne: a [1..n] , un tableau initialisé de type entier ($n \geq 1$) et

np et maxlp, deux variables de type entier.

L'exécution du programme a pour effet d'affecter à la variable np le nombre de plateaux de la suite (a [1] ,a [2] , ... à [n]) et à maxlp le maximum de leurs longueurs (la longueur d'un plateau étant le nombre de ses éléments).

Organigramme





Invariant (avec les notations du cours "Séminaire de programmation").

Inv: $1 \leqslant i \leqslant n$ et

np = nbplat (a[1..i]) (nombre de plateaux de a[1..i])

maxlp = maxlplat (a[1..i]) (longueur du plus grand plateau de a[1..i])

ldp = ldplat (a[1..i]) (longueur du dernier plateau de a[1..i])

Dans notre formalisme...

Pour programmer cet invariant, il s'agit d'abord de programmer les trois opérations nbplat, maxplat et ldplat.

a. Fonction L2 calculant le nombre de plateaux de a[1..i].

Considérons les premiers éléments de chaque plateau de a[1..i]. On vérifie aisément qu'ils sont caractérisés par la propriété suivante:

$j \in \{1..i\} \mid j \neq 1 \Longrightarrow a[j-1] \neq a[j]$

L'ensemble des premiers éléments des plateaux de a[1..i] peut donc s'écrire:

{j in {1..i} | j <> 1 ⟹ a[j-1] <> a[j]}

et le nombre de plateaux de a[1..i] sera le cardinal de cet ensemble.

Programmons la fonction L2 traduisant cette opération:

```
function2 nbplat (var a:array[1..n] of integer; i:integer): integer;
    var ens : set of integer;
    begin
        ens:= {j in {1..i} | j <> 1 ⟹ a[j-1] <> a[j]};
        nbplat:= card (ens)
    end;
```

b. Fonction L2 calculant la longueur du plus grand plateau de a[1..i].

Soit j, le premier élément d'un plateau de a[1..i]; le dernier élément de ce même plateau sera le plus grand entier k_j vérifiant

a. $j \leqslant k_j \leqslant i$ et

b. $\forall\, l \in \{j..k_j\}$: $a[l] = a[j]$

Exprimons cet entier dans notre formalisme:

$$k_j = \max \{k \text{ in } \{j..i\} \mid \text{for all } l \text{ in } \{j..k\}: a[l] = a[j]\}$$

Considérons enfin le maximum des longueurs des différents plateaux

$$\max_{j \in ens} (k_j - j + 1)$$

où ens est l'ensemble des premiers éléments des plateaux de a [1..i] .

Programmons cette opération en L2.

```
function2 maxplat (var a:array [1..n] of integer; i: integer): integer;
    var ens : set of integer;
    begin
        ens:= {j in {1..i} | j <> 1 ==> a[j-1] <> a[j]};
        maxplat:= max for j in ens of
                      (max ({k in {j..i} | for all l in {j..k}:a[l] = a[j]})
                                                                 - j+1)
    end;
```

c. Fonction L2 calculant la longueur du dernier plateau de a [1..i] .

Il suffit, pour cela, de considérer le premier élément du dernier plateau de a [1..i] . Celui-ci peut s'exprimer de la façon suivante:

max (ens)

où ens est l'ensemble des premiers éléments des plateaux de a [1..i] .

La fonction L2 programmant l'opération considérée s'écrit:

```
function2 ldplat (var a:array [1..n] of integer; i: integer): integer;
    var ens : set of integer;
    begin
        ens:= {j in {1..i} | j <> 1 ==> a[j-1] <> a[j]};
        ldplat:= i - max(ens) + 1

    end;
```

L'invariant Inv du programme considéré s'exprime au moyen de ces fonctions de la manière suivante:

Inv: tricrois ([1,i,n]) and
np = nbplat (a,i) and
maxlp = maxlplat (a,i) and
ldp = ldplat (a,i)

Quelques remarques en guise de conclusion

Le premier exemple présenté dans ce paragraphe indique que l'utilité des concepts et opérations du langage L2 ne se réduit pas aux exemples qui les ont introduits. De plus, la "traduction" des assertions de cet exemple dans notre langage est assez immédiate. Il en est de même pour la majorité des problèmes de la classe envisagée. Il en existe cependant certains tels que celui que nous venons de présenter, pour lesquels cette "traduction" est nettement moins directe. Ceci tend à confirmer le fait que, même face à une classe de problèmes assez restreinte, l'expression des assertions dans un formalisme est une activité fort proche de la programmation.

Chapitre III

EXTENSIONS POSSIBLES ? ...

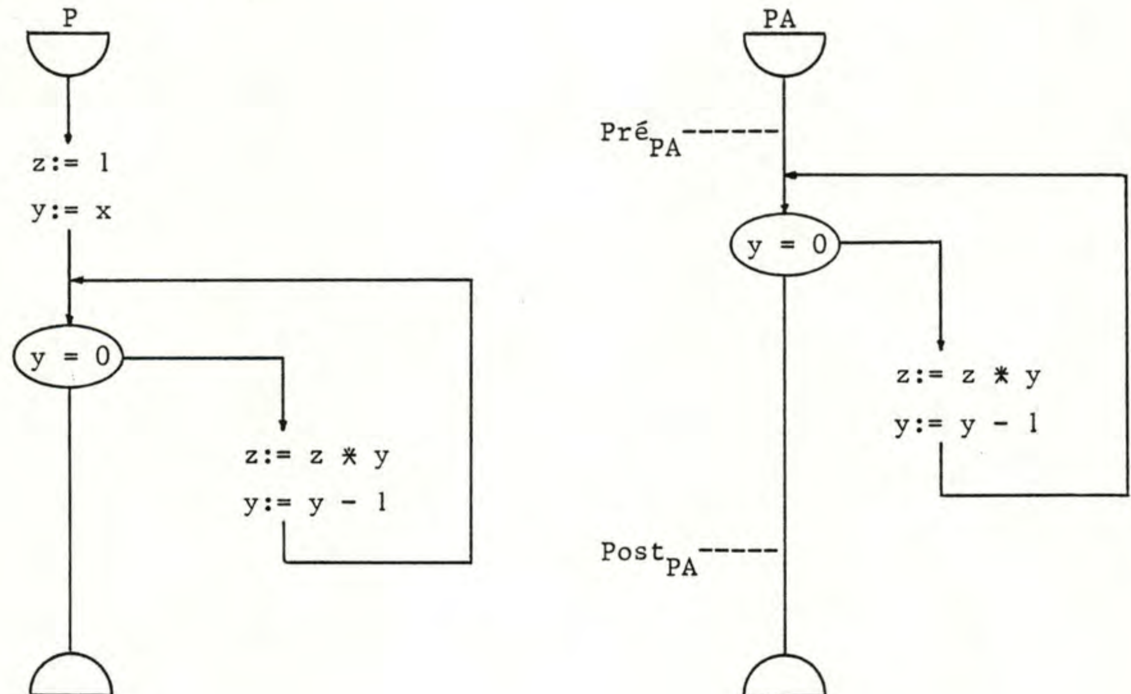
3.1. POUR ILLUSTRER LA METHODE DU PROGRAMME AUXILIAIRE...

Prenons un exemple simple et voyons quels types de vérification à l'exécution il serait intéressant de fournir pour "confirmer la correction" (☆) d'une démonstration par la méthode du programme auxiliaire.

Exemple: La factorielle

Spécification: Etant donné une variable entière x initialisée à une valeur positive, affecter à la variable z la valeur x! sans modifier x.

Organigrammes (☆☆)



(☆) Il serait plus juste de dire "pour détecter un maximum d'erreurs!", cfr.1.1.

(☆☆) Cet organigramme est différent de celui présenté au paragraphe 2.6 répondant à la même spécification: le parcours des valeurs entières entre 0 et x se fait dans l'autre sens. Cela facilite l'expression de la spécification du programme auxiliaire associé. Par contre, l'invariant pour cet organigramme aurait été plus compliqué. Ceci provient du fait que la méthode de l'invariant se base sur des préfixes de l'exécution tandis que la méthode du programme auxiliaire se base sur des suffixes.

où le programme auxiliaire PA associé à P répond à la spécification suivante: si initialement $(y,z) = (y_o,z_o)$ avec $y_o \geq 0$, l'exécution de PA a pour effet d'affecter à la variable z la valeur $z_o * y_o!$

Discussion...

On pourrait exprimer dans notre formalisme $Pré_{PA}$ et $Post_{PA}$ de la façon suivante:

$Pré_{PA}$: let y_o be y; let z_o be z; $y \geq 0$

$Post_{PA}$: $z = z_o * fac(y_o)$

où fac(.) serait une fonction définie dans le langage L2 traduisant la factorielle (cfr. § 2.6).

"Exécutons" maintenant le programme principal en x = 5. Pour illustrer la méthode du programme auxiliaire, il serait intéressant de vérifier les préconditions et postconditions suivantes:

préconditions: $5 \geq 0$, $4 \geq 0$, $3 \geq 0$, $2 \geq 0$, $1 \geq 0$, $0 \geq 0$

postconditions: $z = 1 * 5! = 5 * 4! = 20 * 3! = 60 * 2! = 120 * 1! = 120 * 0!$

Celles-ci traduisent en fait la démarche suivie dans la démonstration (avec les notations du paragraphe 1.4.).

si $\bar{x}_o \in \bar{X}'$ et si test $(\bar{x}_o)$ = faux

on a iter $(\bar{x}_o) \in \bar{X}'$ (préconditions)

g(iter $(\bar{x}_o)$) = g$(\bar{x}_o)$ (postconditions)

Pour être plus complet, il faudrait également vérifier à l'exécution que

iter$(\bar{x}_o) \leqslant \bar{x}_o$ (concerne, en fait, la terminaison)

Dans notre exemple, il s'agirait de vérifier que

$(0,120) \leqslant (1,120) \leqslant (2,60) \leqslant (3,20) \leqslant (4,5) \leqslant (5,1)$

où l'on définit la relation bien fondée $\leqslant$ par:

$(y_o,z_o) \leqslant (y_1,z_1)$ ssi $y_o < y_1$.

Remarquons ensuite qu'en formulant $Pré_{PA}$ et $Post_{PA}$ comme nous l'avons fait, la seule vérification effectuée à l'exécution sera:

$5 \geq 0$ et $z = 1 * 5!$

Une modification possible serait de permettre la récursivité dans le langage L1 et de réaliser les programmes auxiliaires, comme des procédures récursives. Leurs préconditions et postconditions seraient dès lors vérifiées à chaque appel. Quant à la vérification que leurs données, d'un appel récursif à l'autre, respectent la relation bien fondée, elle pourrait s'effectuer d'une manière analogue à la façon dont nous avons traité la terminaison par la méthode de l'invariant.

Remarquons enfin qu'il est possible de transformer les spécifications des programmes auxiliaires en assertions comme l'indique B. Le Charlier dans sa thèse (réf.[1]). Laisser le soin à l'utilisateur de le faire n'est pas une solution adéquate puisque ce travail est avant tout destiné à des étudiants cherchant à se familiariser avec ces méthodes.

3.2. UN OUTIL POUR LE DEVELOPPEMENT DE LOGICIEL ?...

Nous avons indiqué au paragraphe 1.5 que les limites inhérentes à ce type de travail proviennent de la nécessité de formaliser les assertions. Le chapitre II nous semble confirmer la remarque faite dans ce même paragraphe: l'expression des assertions dans un langage formel est une activité fort proche de la programmation. C'est d'ailleurs pourquoi nous avons d'emblée limité les prétentions du travail en restreignant d'abord la classe des problèmes envisagés et en indiquant, ensuite, qu'il se voulait un outil pédagogique et non un outil pour le développement de logiciels "utiles".

a. Adapter ce type d'outil à des programmes en vraie grandeur poserait de sérieux problèmes de coûts

Considérons le problème du tri d'un vecteur présenté au paragraphe 2.8 et remarquons qu'en se limitant à la vérification des invariants, on testera de l'ordre de $(\frac{n^2}{2})$ fois que le vecteur en cours d'exécution est une permutation de son contenu initial (n étant la longueur de ce vecteur). On nous rétorquera sans doute que, dans l'optique du développement de logiciels, un programme de tri aussi peu performant ne serait pas acceptable. Pour pallier à cette remarque, supposons l'existence d'un programme de tri (exagérément) performant ne nécessitant que de l'ordre de n fois cette même vérification: tester n fois qu'un vecteur de longueur n est une permutation d'un autre durant l'exécution d'un programme de tri d'un tel vecteur devient très vite disproportionné par rapport à cette même exécution sans vérification, pour peu que n dépasse la dizaine. On admettra aisément que le problème soulevé par cet exemple s'accentuera suivant le nombre de boucles imbriquées du programme envisagé.

b. Un langage de programmation d'assertions ne pourrait être "complet" (☆) qu'en limitant précisément les applications que l'on envisagera et non en restreignant les primitives du langage de programmation initial.

Prenons l'exemple très simple qu'est celui de la multiplication présenté au paragraphe 2.3 et supposons que, seul, l'opérateur arithmétique habituel + soit permis dans le langage de programmation initial. Nous avons déjà montré que l'opérateur * devrait être fourni par le langage de programmation d'assertions. Utilisons itérativement l'opération * traduite par l'organigramme en question; un langage de programmation d'assertions complet devrait dès lors permettre des opérations telles que l'exponentiation ou la factorielle... et ainsi de suite. C'est ainsi que, partant d'un langage de programmation d'assertions, il suffit de choisir une de ses primitives "de plus haut niveau", de la programmer et de "lui trouver une application en l'utilisant itérativement" pour que les assertions du programme traduisant cette application ne puissent plus s'exprimer directement, "sans programmation", mais nécessite l'emploi de notations algorithmiques telles que

$$\sum_{i=n}^{m} \ldots .$$

c. Même en limitant le domaine d'application d'un tel outil, la vérification à l'exécution des assertions peut être très compliquée, voire impossible.

Prenons deux exemples simples illustrant l'impossibilité de vérifier certaines assertions à l'exécution. Considérons tout d'abord le domaine d'application concernant la minimisation de fonctions entières. La spécification (simplifiée) d'un programme de minimisation peut avoir la forme suivante: étant donné un sous-programme (fonction) calculant une fonction totale convexe, l'exécution du programme renvoie le minimum de cette fonction. Le théorème

---

(☆) Par complet, nous entendons permettre l'expression directe de toutes les assertions d'une classe de problèmes.

de Rice (☆) nous indique que la vérification de la précondition d'une telle spécification n'est pas calculable. Remarquons qu'en considérant d'autres domaines d'application tel celui des compilateurs et interpréteurs, nous serions confrontés au même problème qu'est la non-calculabilité de certaines vérifications [3].

Considérons un tout autre domaine: celui des prévisions météorologiques. La spécification d'un programme dans ce domaine pourrait être la suivante: son exécution renvoie la valeur vrai s'il fera beau demain. Dans ce cas, il paraît même insensé de vouloir vérifier à l'exécution une telle spécification: rappelons-nous qu'un des rôles d'une spécification d'un programme est d'indiquer quelle information on peut tirer de son exécution, où le terme information est employé au sens commun et non au sens qu'on lui donne dans le jargon informatique [3].

d. Si un "bon" langage d'expression de spécifications ou d'assertions existait, il serait plus intéressant (☆☆) de l'utiliser comme langage de programmation.

Considérons à nouveau l'exemple de la multiplication présenté au paragraphe 2.3. Il est bien évident que l'organigramme présenté perdrait son intérêt si l'on pouvait utiliser l'opérateur * qui, comme nous l'avons indiqué, devrait être proposé par le langage de programmation d'assertions. Cet exemple ainsi que ceux présentés au deuxième chapitre semblent confirmer la remarque suivante: en supposant l'existence d'un tel outil pour le <u>développement</u> d'une classe de logiciels, il est fort probable qu'il serait beaucoup plus intéressant (☆☆) de programmer ces logiciels dans le langage de programmation

---

(☆) Théorème de Rice: supposons qu'on ait défini une propriété des fonctions partielles. Alors, s'il existe un algorithme permettant de déterminer si un programme quelconque calcule une fonction partielle avec cette propriété, <u>toutes</u> les fonctions partielles calculables ont cette propriété, ou <u>aucune</u> ne l'a.

(☆☆) du point de vue de la fiabilité par rapport aux coût.

d'assertions plutôt que dans le langage de programmation initial. En effet, un "bon" langage de programmation d'assertions se doit de permettre l'expression (programmation) aisée des diverses applications envisagées et de leurs propriétés.

c. Quant aux démonstrateurs automatiques...

Remarquons que les démonstrateurs automatiques seront sans doute confrontés au même type de limites puisqu'ils nécessitent entre autres la formalisation des spécifications. De plus, leur but est plus ambitieux puisqu'il s'agit non plus de vérifier, à l'exécution, que la "spécification" (☆) d'un programme est respectée mais bien de décider si la fonction partielle calculée par celui-ci vérifie ou non les propriétés traduites par la spécification (☆☆).

La raison principale pour laquelle le style d'outil proposé ne peut être utilisé (en tant que tel) dans l'optique <u>développement</u> de logiciels me semble être résumée par le professeur H. Leroy dans la conclusion de son ouvrage [2]: "*Un bon langage de spécification, quel que soit son degré de formalisation, doit puiser ses concepts dans les <u>applications</u> elles-mêmes et non dans une vue, aussi abstraite qu'on la suppose, de ce qu'il est possible de faire avec un langage de programmation*". C'est pourquoi, si l'objectif est de fournir un outil pour le développement d'une classe de logiciels, il nous semble plus intéressant de se pencher sur le problème consistant à trouver un "bon" (☆☆☆) langage de programmation plutôt que sur celui consistant à essayer de formaliser les spécifications ou les assertions en vue de leur vérification.

(☆) Les guillemets soulignent que la formalisation des spécifications dénature leur rôle.

(☆☆) A ce sujet, le théorème de Rice déjà cité émet de sérieux doutes quant à la faisabilité de cet objectif!... [3].

(☆☆☆) Par "bon" langage de programmation, nous entendons un langage de programmation qui, étant donné la classe des problèmes envisagés <u>et</u> les personnes auxquelles il est destiné, facilite le raisonnement visant à se persuader de la correction des programmes écrits dans ce langage.

CONCLUSION

Le but premier de ce travail était de proposer un langage de programmation permettant l'expression d'assertions en vue de leur vérification à l'exécution. Dans le temps qui nous était imparti, nous n'avons pas pu envisager l'implémentation, même dans son principe. Toutefois, nous avons constamment gardé à l'esprit qu'elle devait être possible. Le choix d'un mode performant d'implémentation, notamment des suites et des ensembles, nous semble cependant nécessiter un travail considérable.

Si ce langage est effectivement implémenté pour, ensuite, motiver quelque peu la résolution d'exercices de démonstration de programmes, l'objectif de ce travail sera atteint, mais il le sera déjà amplement s'il illustre d'une manière pertinente la constatation suivante: l'expression de spécifications et d'assertions dans un langage formel est une activité fort proche de la programmation. Une des conséquences de cette remarque est que c'est essentiellement des compétences du programmeur que dépendra, vraisemblablement toujours, la fiabilité d'un programme. En effet, en supposant l'existence d'outils sophistiqués tels des démonstrateurs automatiques, la formalisation de la spécification des programmes, exigée par ces outils, restera le travail du programmeur et nécessitera des raisonnements comparables à ceux utilisés en programmation.

RÉFÉRENCES

[1] B. Le Charlier, *Réflexions sur le problème de la correction des programmes*, thèse de doctorat, Institut d'Informatique, janvier 1985.

[2] H. Leroy, *La fiabilité des programmes*, Presses Universitaires de Namur, 1975.

[3] H. Leroy, *Notes sur la calculabilité*, février 1984 (manuscrit).

[4] R.L. London, *Experience with inductive assertions for proving programs correct*, Symposium on Semantics of Algorithmic Languages, p. 236.

[5] H. Leroy, *La fiabilité des programmes*, Ecole d'été de l'AFCET, juillet 1978.

[6] P. Naur, *Proof of Algorithms by General Snapshots*, BIT 6, 1966.

[7] R.W. Floyd, *Assigning Meanings to Programs*, Proceedings of a Symposium in Applied Mathematics, 1967.

[8] Z. Manna - R.J. Waldinger, *Studies in Automatic Programming logic*, Artificial Intelligence series, The Computer Science library.

[9] R.M. Burstall, *Program proving as hand simulation with a little induction*, North-Holland Publishing Company, 1974.

[10] J.H. Morris - B. Wegbreit, *Subgoal Induction*, Xerox Palo Alto Research Center, 1977.

Facultés Universitaires Notre-Dame de la Paix - NAMUR

INSTITUT D'INFORMATIQUE

DEFINITION D'UN LANGAGE DE PROGRAMMATION PERMETTANT L'EXPRESSION D'ASSERTIONS

(ANNEXE)

Denis FISETTE

Année académique 1984-1985

TABLE DES MATIÈRES

Page

Introduction 1

Chapitre I CONSIDERATIONS GENERALES 3

1.1. Classe des problèmes envisagés 3

1.2. Structure d'un programme avec assertions 5

a. Diagramme 5

b. Traduction B.N.F. 6

1.3. Remarques quant à la présentation 7

Chapitre II QUELQUES NOTIONS FONDAMENTALES 8

2.1. Les symboles de base 8

2.2. Mémoires, variables et tableaux 9

2.3. A propos des déclarations de fonctions et de procédures 10

2.4. Les fichiers d'entrée et de sortie 11

2.5. Notion de contexte 12

2.6. Les expressions de désignation 13

Chapitre III LE LANGAGE L1 15

3.1. Introduction, parties concernées et symboles spéciaux 15

3.2. Types de valeurs et opérations primitives associées 16

a. Le type entier 1 17

b. Le type caractère 17

c. Le type booléen 18

3.3. Les expressions (sauf les appels de fonctions) 19

a. Expressions de type caractère 19

b. Expressions arithmétiques 20

c. Expressions booléennes 22

3.4. Déclarations de variables et tableaux 25

3.5. Instructions (introduction aux deux types d'exécution) 27

a. Instructions d'affectation 28

b. Instructions conditionnelles 29

c. Instructions répétitives 30
d. Instructions de lecture 30
e. Instructions d'écriture 31
3.6. Instructions composées, interfaces pour assertions, exécution avec ou sans vérification d'assertions 33
3.7. Instructions de branchement 36
3.8. Déclarations et appels de fonctions ou de procédures 37
a. Déclarations 37
b. Appels 38

Chapitre IV . LE LANGAGE L2 42

4.1. Introduction, parties concernées et symboles spéciaux 42
4.2. Types de valeurs et opérations primitives associées 45
a. Types de base 45
b. Types générés à partir des types de base 47
4.3. Expressions (sauf les appels de fonctions) 50
a. Expressions arithmétiques 50
b. Expressions de type caractère 53
c. Expressions booléennes 54
d. Expressions de type ensemble et suite d'entiers 57
e. Expressions de type ensemble de caractères et suite de caractères 62
4.4. Déclarations de variables et tableaux 63
4.5. Instructions (sauf les appels de procédures) 64
4.6. Instructions L1/Assertions L2: Interface sémantique 67
4.7. Déclarations et appels de fonctions ou de procédures 68
a. Déclarations 68
b. Appels 69
4.8. Fonctions prédéfinies 70
a. Fonctions prédéfinies à arguments de type suite 70
b. Fonctions prédéfinies à arguments de type ensemble 72
4.9. Mémorisation des valeurs intermédiaires 74
4.10. "Fonctions" prédéfinies supplémentaires 76
4.11. Les énoncés d'assertion et de précondition 79
a. Enoncés d'assertion 79
b. Enoncés de précondition 79

Chapitre V DEUX EXEMPLES EN GUISE DE CONCLUSION 81

5.1. Choix et présentation des exemples 81

5.2. La factorielle 82

5.3. Tri d'un vecteur 84

a. Spécification 84

b. Programme 84

Référence 87

INTRODUCTION

Comme l'indique son titre, l'objet de cette annexe est la présentation d'un langage de programmation permettant l'expression d'assertions. Précisons d'emblée ce titre en ajoutant qu'il s'agit de pouvoir vérifier ces assertions à l'exécution. La première partie de ce mémoire indique, en détail, les objectifs d'un tel travail. Contentons-nous ici de les résumer en ne mentionnant que l'objectif principal: fournir un langage permettant d'illustrer le cours, intitulé "Séminaire de programmation", qui se donne, actuellement, en première licence en informatique. Cet objectif suggère les qualités que l'on est en droit d'attendre d'un tel langage. Une première est qu'il puisse être facilement maîtrisé par les étudiants concernés. Pour ce faire, nous mettrons l'accent sur sa présentation en tâchant de la rendre agréable et compréhensible (☆). Ces qualités sont, à notre avis, non moins importantes que d'autres telles la complétude ou la non-ambiguïté.

Cette présentation est divisée en quatre parties: une première indique le type de notations utilisées dans le cours en question et dresse, ensuite, l'architecture grossière de ce que sera, dans notre formalisme, un programme avec assertions. Un choix relatif à la présentation a été de scinder ce langage en deux sous-langages, l'un se préoccupant de l'expression des programmes proprement dits et l'autre, de leurs assertions. Pour être plus concis, nous les dénommerons, respectivement, L1 et L2 (☆☆). Le chapitre II se soucie des concepts communs aux deux langages et les chapitres III et IV, des concepts propres à chacun d'eux.

En vue de favoriser sa maîtrise, le langage L1 ressemblera au langage Pascal, supposé connu des étudiants concernés. Il sera, cependant,

---

(☆) présentation s'inspirant de celle du langage LSD80 réf.[1].
(☆☆) appellations hautement suggestives!...

moins riche. Quant au langage L2, son but est donc l'expression d'assertions. La première partie de ce mémoire indique qu'il est plus judicieux de parler de "programmation d'assertions". Pour satisfaire l'objectif que l'on s'est fixé, il serait intéressant que cette "programmation" soit rendue aussi aisée que possible. Ceci sera notre constante préoccupation lors de l'élaboration du langage L2 (☆).

Un dernier chapitre présente, en guise de conclusion, la résolution de quelques problèmes; ceux-ci se veulent représentatifs parmi la classe de ceux qui sont envisagés dans le cadre du cours mentionné précédemment.

(☆) Les raisons des choix faits lors de l'élaboration du langage L2 sont explicitées, en détail, dans les chapitres I et II de la première partie de ce mémoire.

Chapitre I

CONSIDÉRATIONS GÉNÉRALES

1.1. CLASSE DES PROBLEMES ENVISAGES

Le premier but de tout langage de programmation est de pouvoir exprimer une classe d'algorithmes. Cette classe est principalement déterminée par deux critères:

1. les types de valeurs permises et leurs opérations primitives,
2. les mécanismes de composition algorithmique permis.

La classe des algorithmes envisagés étant celle correspondant aux problèmes posés au cours "Séminaire de programmation" (1ère lic.), la restriction aux objets de type entier, caractère et booléen semble convenir. En effet, si quelquefois le type réel intervient, son remplacmeent systématique par des entiers ne change rien au but pédagogique poursuivi (i.e.même complexité de résolution).

Dans ce même cours, les algorithmes sont représentés sous la forme d'organigrammes. Leurs assertions sont situées sur l'organigramme par un trait discontinu indiquant l'arc de ce dernier auquel elle se rapporte (cfr fig. 1).

Les mécanismes de composition algorithmique du langage L1 ont été choisis pour permettre un passage simple et systématique de cette représentation sous forme d'organigrammes en une représentation dans ce langage.

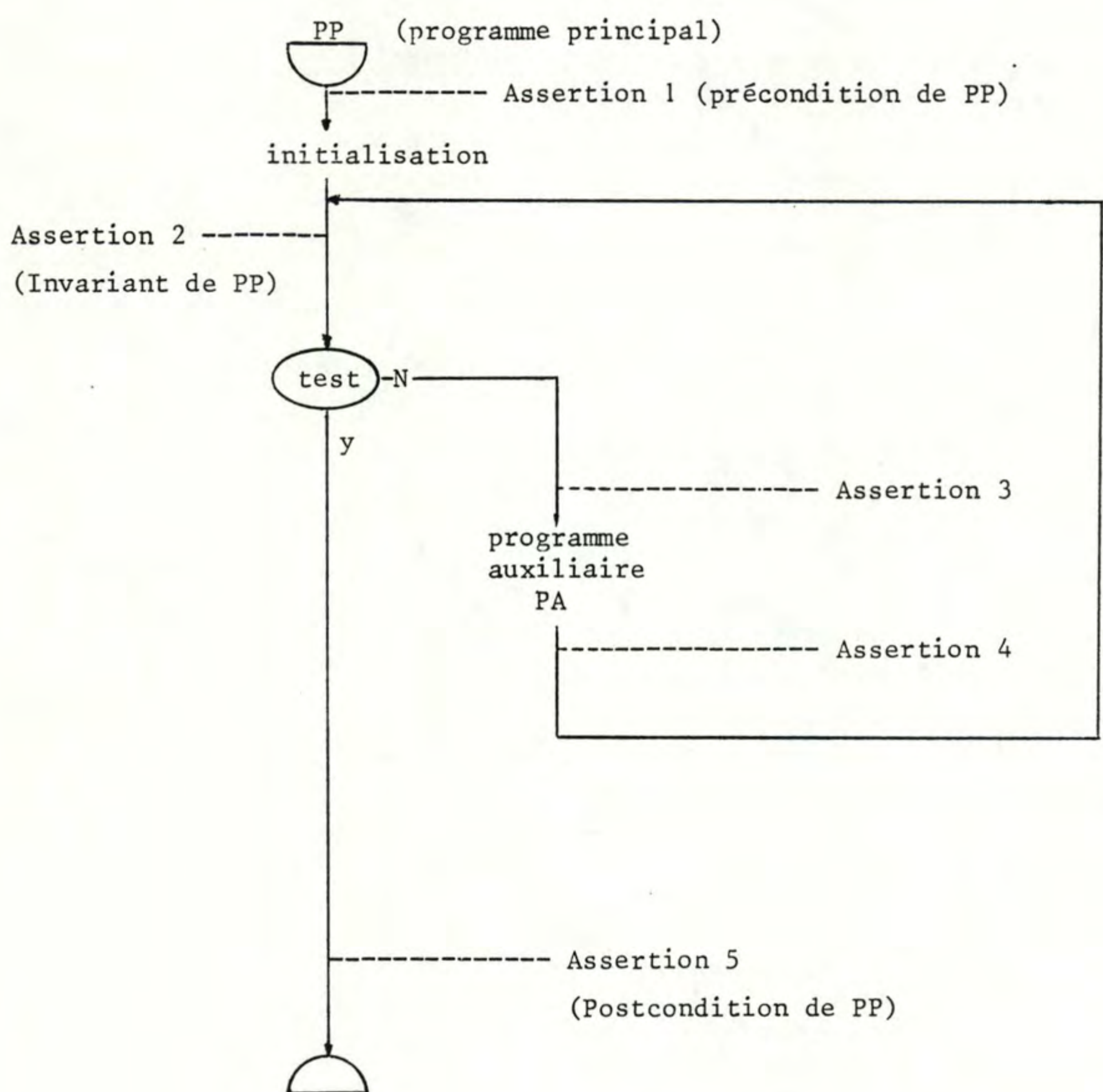
Formalisme des organigrammes ——————→ Formalisme L1
traduction

Par contre, les assertions étant exprimées dans ce cours au moyen de notations propres à chaque problème (ie en dehors de tout formalisme), on ne pourra espérer construire un langage, L2, permettant la traduction

systématique de ces énoncés, en énoncés écrits dans ce langage. Il s'agira dès lors de construire un langage permettant leur "programmation" d'une façon aussi aisée que possible.

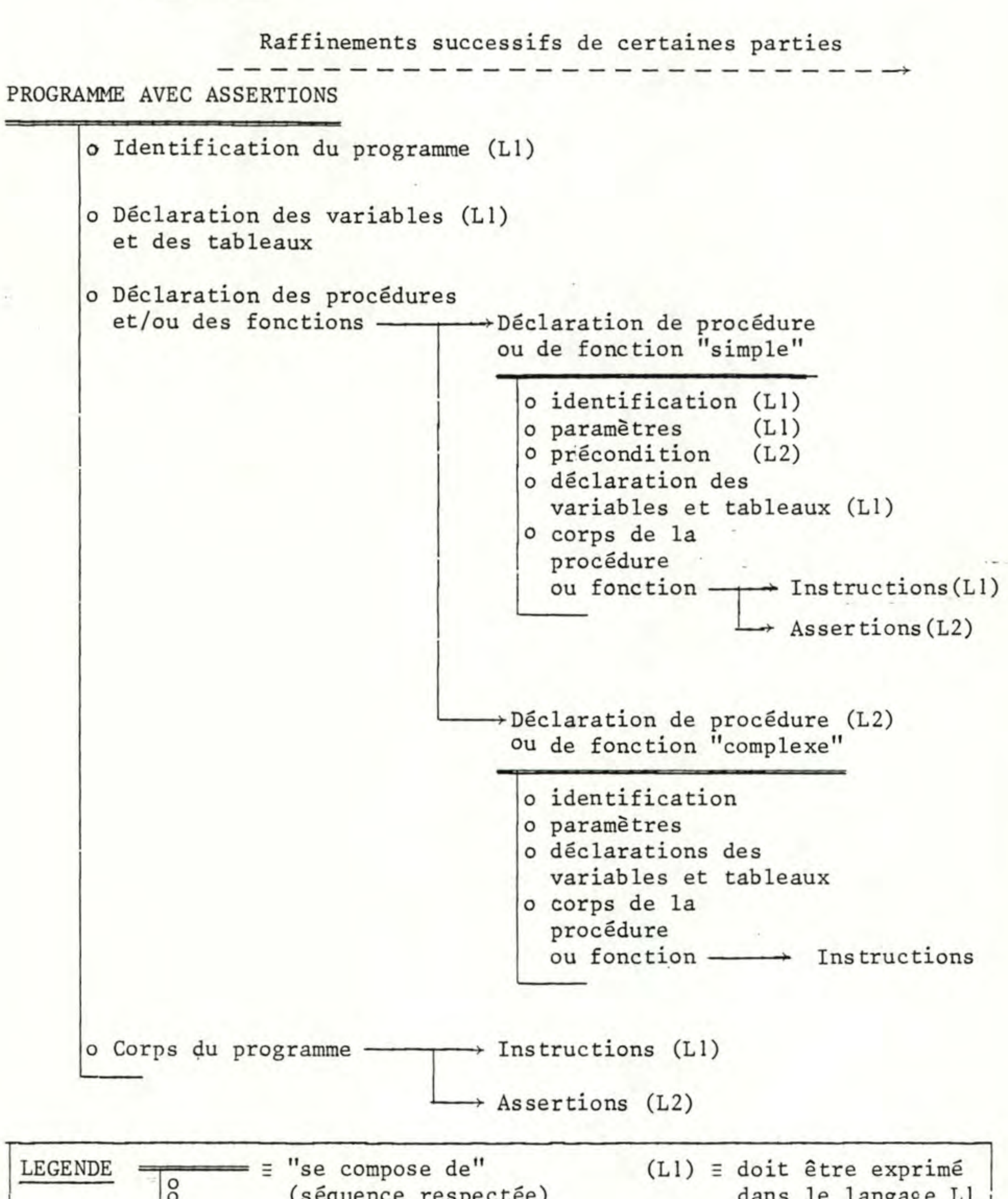
Notation des assertions ⟿ Formalisme L2
programmation

Fig. 1



1.2. STRUCTURE D'UN PROGRAMME AVEC ASSERTIONS

a. Diagramme



LEGENDE		
≡ "se compose de" (séquence respectée)	(L1) ≡ doit être exprimé dans le langage L1
≡ "peut être composé de" (séquence non nécessairement respectée)	(L2) ≡ doit être exprimé dans le langage L2

b. Traduction B.N.F.

Précisons ce diagramme en utilisant la grammaire B.N.F. ...

Remarque : Les caractères "1" et "2" apparaissant à la fin d'un mot indiquent si ce dernier désigne un concept du langage L1 ou du langage L2.

<programme avec assertions>:: = program <nom de programme>; <déclaration des variables et des tableaux><déclaration des fonctions et/ou des procédures><corps 1>.

<déclaration des fonctions et/ou des procédures>:: = <vide>| <déclaration des fonctions et/ou des procédures><déclaration de fonction ou de procédure L1>; |<déclaration des fonctions et/ou des procédures><déclaration de fonction ou de procédure L2>;

<déclaration de fonction ou de procédure L1>:: = <déclaration de fonction 1>| <déclaration de procédure1 >

<déclaration de fonction ou de procédure L2>:: = <déclaration de fonction 2>| <déclaration de procédure2 >.

<déclaration de fonction 1>:: = <identification fonction 1><précondition <déclarations des variables et des tableaux><corps1 >
<déclaration de procédure1 >:: = <identification procédure1 ><précondition> <déclarations des variables et des tableaux><corps1 >

<déclaration de fonction 2>:: = <identification fonction2 ><déclarations des variables et des tableaux><corps2 >
<déclararion de procédure 2>:: = <identification procédure 2><déclarations des variables et des tableaux><corps 2>

<corps 1>:: = <instruction composée (d'instructions et d'assertions)>

<corps 2>:: = <instruction composée (d'instructions)>

1.3. REMARQUES QUANT A LA PRESENTATION...

La présentation des différents concepts sera généralement divisée en deux rubriques; une rubrique "syntaxe" et une autre, "sémantique". Sous la première seront reprises les règles syntaxiques qui doivent être vérifiées par un texte écrit dans le langage. Ces règles sont explicitées partiellement au moyen des notations BNF. La rubrique "sémantique", quant à elle, se préoccupera essentiellement de deux choses; d'une part, de tout ce qui est du domaine de l'exécution (étant donné un texte syntaxiquement correct, que se passera-t-il à l'exécution...) mais également de la sémantique des déclarations (de variables, tableaux, procédures, ...) qui permet l'analyse syntaxique. Sous cette seconde rubrique, les règles seront quelquefois explicitées en se basant sur les conventions suivantes. Elles seront réparties en différentes étapes notées # 1, # 2, ..., # N-1, # N pour indiquer leur agencement: d'une manière générale, cela exprime qu'il faut d'abord suivre les règles de l'étape # 1, puis de l'étape # 2, ... et ainsi de suite. Mais cet ordre peut être modifié par l'emploi de la périphrase "aller en # i" qui, figurant à l'étape # j, indique qu'on doit arrêter de suivre les règles de l'étape # j pour suivre celles de l'étape # i, # i+1, ... jusqu'à l'apparition d'une autre périphrase, rompant à nouveau l'ordre de base. La fin de la séquence des règles à appliquer sera indiquée explicitement.

Chapitre II

QUELQUES NOTIONS FONDAMENTALES

2.1. LES SYMBOLES DE BASE

La syntaxe des langages L1 et L2 est, en fait, un ensemble de règles d'ordonnancement de symboles de base considérés comme indécomposables. Le texte d'un programme avec assertions est donc réduit à une suite syntaxiquement correcte de symboles de base. L'ensemble des symboles de base est défini ci-dessous par une grammaire B.N.F.

<symbole de base>:: = <identificateur>|<constante>|<symbole spécial>

<identificateur>:: = <lettre minuscule>|<identificateur><lettre minuscule>|
<identificateur><chiffre>

<lettre minuscule>:: = a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z

<chiffre>:: = 1|2|3|4|5|6|7|8|9|0

<constante>:: = <entier>|'<caractère>'|<valeur de vérité>

<entier>:: = <chiffre>|<entier><chiffre>

<caractère>:: = <lettre majuscule>|<chiffre>|<caractère spécial>

<lettre majuscule>:: = A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z

<caractère spécial>:: = ' '|+|-|*|/|(|)|=|>|<|,|.|"|'|?|:|!|;

<valeur de vérité>:: = <u>true</u>|<u>false</u>

<symbole spécial>: Ils diffèrent selon le langage (⇒ Voir chapitres III et IV).

*

*   *

2.2. MEMOIRES, VARIABLES ET TABLEAUX

Désignons par mémoire tout support physique vérifiant les quatre propriétés suivantes :

- il est possible d'y enregistrer de l'information;
- il est possible d'y lire l'information contenue;
- l'information contenue ne change pas tant qu'il n'y a pas nouvel enregistrement;
- une mémoire est caractérisée par le type d'information qu'elle peut contenir.

On appelle variable simple, l'association d'un identificateur et d'une mémoire. L'identificateur est appelé le nom de la variable, la valeur (si elle existe) dont la représentation est enregistrée dans la mémoire à un instant donné, est la valeur courante de la variable. Enfin, on appelle type de la variable, le type de sa mémoire.

On appelle tableau, l'association d'un identificateur (le nom du tableau) et d'une bijection d'un intervalle fini de nombres entiers (les indices du tableau) dans un ensemble de mémoires (les éléments du tableau) de même type (le type du tableau).

On appelle variable indicée, l'association d'un tableau et d'un indice de ce tableau.

Soit x, une variable, indicée ou non, et v, une valeur du même type. Affecter à x la valeur v exprime le fait d'enregistrer une représentation de v dans la mémoire de x. Dès cet instant, la valeur courante de x sera v et le restera jusqu'à ce qu'on lui affecte une valeur au moins une fois depuis sa création jusqu'à l'instant t.

*

*   *

2.3. A PROPOS DES DECLARATIONS DE FONCTIONS ET DE PROCEDURES

Le mécanisme algorithmique sous-jacent à ces notions de procédures et fonctions permet notamment de traduire l'emploi de programmes auxiliaires utilisés comme une primitive dans les organigrammes (cfr § 1.1). En effet, leurs déclarations sont formées d'une suite d'opérations sur des paramètres suffisamment générale pour qu'en plusieurs endroits du programme avec assertions, il soit intéressant d'utiliser en bloc ces opérations en particularisant les paramètres à certaines valeurs, variables ou tableaux. De plus, cette suite d'opérations est censée former un tout, de sorte qu'une spécification générale (en fonction des paramètres) permette leurs utilisations en différents endroits tout comme l'emploi d'opérations primitives.

Les déclarations de fonctions et de procédures sont des constructions du langage dont la sémantique peut être décrite (très) approximativement, comme suit :

- Une déclaration de fonction spécifie un procédé de calcul d'une valeur v (valeur de la fonction), à partir d'un certain nombre d'autres valeurs $v_1, v_2, \ldots, v_n$ (arguments de la fonction) si ces valeurs vérifient la précondition de la fonction.
- Une déclaration de procédure spécifie une suite d'opérations à effectuer sur un ensemble de variables et modifiant les valeurs de celles-ci si initialement, les valeurs de ces variables vérifient la précondition de la procédure.

*

* *

2.4. LES FICHIERS D'ENTREE ET DE SORTIE

Le fichier d'entrée d'un programme avec assertions est un support physique sur lequel est enregistrée, à un moment donné de l'exécution du programme, la représentation d'une suite de caractères α. Cette suite constitue "la suite des données restant à lire par le programme", à cet instant t. On l'appelle également l'état du support d'entrée.

Le fichier de sortie du même programme, est un autre support physique, contenant au même moment de son exécution, la représentation d'une autre suite de caractères β. Celle-ci constitue "la suite des résultats déjà imprimés, par le programme". On l'appelle aussi l'état du fichier de sortie, à cet instant.

L'état initial α_0 du fichier d'entrée constitue l'ensemble des données du programme. L'état initial du fichier de sortie est la suite vide.

La lecture d'un caractère et l'écriture d'un caractère ou d'un entier sont les seules opérations primitives pouvant être effectuées, respectivement, sur le fichier d'entrée et sur le fichier de sortie, durant l'exécution du programme. L'effet exact de ces opérations sera précisé dans la sémantique des instructions de lecture et d'écriture.

*

*   *

2.5. NOTIONS DE CONTEXTE

On définit un contexte C en se donnant

- un fichier d'entrée et un de sortie,
- un ensemble de variables simples et de tableaux de noms distincts,
- un ensemble de déclarations de fonctions et de procédures de noms distincts et
- un ensemble d'étiquettes de noms distincts.

Une variable simple et un tableau ne peuvent avoir le même nom dans un même contexte mais, par contre, ils peuvent avoir le même nom qu'une déclaration de fonction ou de procédure ou qu'une étiquette.

A tout instant t, les fichiers d'entrée et de sortie du contexte C sont dans un certain état et les mémoires de ses variables et tableaux possèdent des valeurs bien déterminées, ou n'ont pas encore été initialisées. L'ensemble de ces renseignements constitue <u>l'état du contexte C</u> à l'instant t.

Remarque 1: Notations: Pour faciliter la suite de l'exposé, on utilisera la notation C pour désigner la partie figée d'un contexte (ie sans considérer son état) et la notation E(C) pour désigner ce même contexte mais, cette fois, en considérant son état courant.

Remarque 2:
- Les règles syntaxiques seront explicitées en fonction d'un contexte figé (partie figée) C. (rubrique "syntaxe").
- La rubrique "sémantique" se préoccupera, d'une part, de l'élaboration d'un contexte C face à des déclarations (de variables, tableaux, procédures, ...) bien qu'il puisse être fixé une fois pour toutes (en dehors d'une exécution particulière) et, d'autre part, de tout ce qui concerne l'exécution d'un texte syntaxiquement correct en fonction d'un contexte et de son état courant E(C).

2.6. EXPRESSIONS DE DESIGNATION

Une expression de désignation de variable (de tableau) est une construction qui, étant donné un contexte et son état, identifie, sous certaines conditions, une et une seule variable (tableau).

Syntaxe

<expression de désignation de tableau>:: = <nom de tableau>
<expression de désignation de variable>:: = <nom de variable>|<expression de désignation de tableau>[<expression arithmétique>]
<nom de variable>:: = <identificateur>
<nom de tableau> ::= <identificateur>

Pour qu'une expression de désignation de variable (de tableau) soit syntaxiquement correcte dans le contexte C, l'identificateur concerné par celle-ci doit référencier une variable (tableau) de C.

De plus, on dit qu'une expression de désignation de tableau a un type, le type du tableau désigné. De même, pour les expressions de désignation de variable, s'il s'agit d'une variable simple, c'est son type, et il s'agit d'une variable indicée, c'est le type du tableau désigné.

Sémantique

Soient C, un contexte, x et t, des identificateurs. Si x est un nom de variable de C, alors l'expression de désignation x désigne cette variable. Si, dans C, t identifie un tableau, l'expression de désignation de tableau t identifie ce tableau et l'expression de désignation (de variable (☆)) t[exp] où exp est une expression arithmétique, est évaluée de la manière suivante: on évalue l'expression arithmétique exp dans E(C). Si cette évaluation est indéterminée, c'est aussi le cas de l'évaluation de

(☆) Lorsque l'on parlera d'expresison de désignation, on sous-entendra expression de désignation de variable et lorsqu'il s'agira d'une expression de désignation de tableau, on l'indiquera explicitement.

l'expression de désignation t[exp], sinon notons i, la valeur de l'expression arithmétique exp; si i est un des indices du tableau, l'expression de désignation en question est déterminée et désigne l'unique variable indicée t[i]; dans l'alternative, son évaluation est dite indéterminée.

Exemples: Si t est un nom de tableau et x, i, j des noms de variables, t est une expression de désignation de tableau et x, t[i+j], t[23] sont des expressions de désignation.

Chapitre III

LE LANGAGE L1

3.1. INTRODUCTION, PARTIES CONCERNEES ET SYMBOLES SPECIAUX

Ce chapitre présente un langage, L1, qui ressemble au Pascal tout en étant beaucoup moins riche. Cette présentation est découpée en plusieurs parties qui sont agencées de la manière suivante: ce premier paragraphe, après cette brève introduction, rappelle les parties d'un programme qui doivent être exprimées dans le langage L1 et se termine en explicitant les symboles spéciaux de ce langage. Le paragraphe suivant introduit les différents types de valeurs et leurs opérations primitives. Le troisième paragraphe, quant à lui, se préoccupe des différentes expressions possibles et le quatrième, des déclarations des variables et des tableaux. Une caractéristique importante du langage considéré est de permettre deux types d'exécution, avec ou sans vérification d'assertions. Les particularités de ces deux types d'exécution sont introduites aux paragraphes 5 et 6, le premier se souciant des instructions dont la sémantique ne change pas suivant le type d'exécution choisi et le second, de l'instruction composée dont la sémantique est exposée pour les exécutions avec et sans vérification d'assertions. Les instructions de branchement font l'objet du paragraphe 7. Nous terminerons finalement ce chapitre en nous penchant sur les déclarations et les appels de fonction ou de procédure.

Parties concernées : Comme l'indique le schéma général d'un programme avec assertions, la totalité du programme avec assertions doit être exprimée dans le langage L1 à l'exception

1. des déclarations des fonctions et procédures L2,
2. des préconditions des fonctions et procédures L1 et
3. des énoncés d'assertions.

Symboles spéciaux : Le chapitre II a déjà défini partiellement les symboles de base du langage L1. Restait à expliciter ses symboles spéciaux :

<symbole spécial>:: = +|-|*|(|)|=|,|.|<|>|:|;|]|[|{|}|
and|array|begin|boolean|char|div|do|else|end|exit|
false|function1|goto|if|integer|mod|not|of|or|
precondition|procedure1|program|read|then|true|var|
while|write

*

*   *

3.2. TYPES DE VALEURS ET OPERATIONS PRIMITIVES ASSOCIEES

a. Le type entierl

Soit $\mathbb{Z}$, l'ensemble des entiers

a. Ensemble des objets de type entierl = Z (☆)

b. Opérations primitives :

$$Z \times Z \longrightarrow Z$$

addition	$(a,b) \rightsquigarrow a + b$	
soustraction	$(a,b) \rightsquigarrow a - b$	
multiplication	$(a,b) \rightsquigarrow a * b$	
division entière	$(a,b) \rightsquigarrow a \text{ div } b$	Précondition:
reste	$(a,b) \rightsquigarrow a \text{ mod } b$	$b \neq 0$

$$Z \times Z \longrightarrow B = \{\text{vrai, faux}\}$$

égalité $(a,b) \rightsquigarrow (a = b)$

où (a=b) vaut la valeur de vérité vrai si l'entier a égale l'entier b et faux dans le cas contraire.

De même pour les autres types de comparaison: <, >, < = , > = et < >.

b. Le type caractère

Soit l'ensemble C = {' ',A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T, U,V,W,X,Y,Z,0,1,2,3,4,5,6,7,8,9,+,-,*,/,(,), = ,",",.,"ʺ",!, ',?, , ,:} auquel on associe la relation d'ordre définie par l'énumération de ces éléments.

a. Ensemble des valeurs des objets de type caractère : C

(☆) On ignore pour l'instant tous les problèmes provenant des dépassements des plus petits et plus grands nombres entiers représentables dans le système.

b. Opérations primitives

$$C \times C \longrightarrow B = \{vrai, faux\}$$

$$\text{égalité } (a,b) \rightsquigarrow (a = b)$$

où (a=b) vaut la valeur de vérité vrai si le caractère a égale le caractère b et faux dans le cas contraire. De même pour les autres types de comparaison.

c. Le type booléen

a. Ensemble des valeurs des objets de type booléen : B = {vrai,faux}

b. Opérations primitives

$$B \times B \longrightarrow B$$

$$\text{conjonction } (a,b) \rightsquigarrow a \text{ and } b$$

$$\text{disjonction } (a,b) \rightsquigarrow a \text{ or } b$$

$$B \longrightarrow B$$

$$\text{négation } \quad a \rightsquigarrow \text{not } a$$

*

*   *

3.3. EXPRESSIONS (sauf les appels de fonctions)

Une expression est une construction du langage qui, étant donné l'état de son contexte E(C), spécifie une suite d'opérations à effectuer pour calculer une valeur v. Cette suite d'opérations est appelée l'évaluation de l'expression dans E(C) et v, sa valeur. On appelle également type de l'expression, le type des valeurs qu'elle calcule. Il est toutefois possible que, pour certains contextes ou pour certains états d'un même contexte, l'évaluation de l'expression soit indéterminée ou ne se termine pas. Dans ce cas, la valeur de l'expression n'existe pas.

Syntaxe générale

<expressionl>:: = <expl arithmétique>|<expl de type caractère>|
<expl booléenne>

a. Expression de type caractère

Syntaxe :

<expl de type caractère>:: = <expression de désignation de type caractère>|
'<caractère>'|<appel de fonction de type
caractère>|[<expl de type caractère>]

Sémantique (sauf les appels de fonctions) :

Si l'expression de type caractère se résume à une expression de désignation, on évalue cette dernière dans E(C), état du contexte de l'expression. Si cette évaluation est déterminée et si la variable qu'elle identifie a été initialisée, alors l'expression est déterminée et sa valeur est la valeur de la variable identifiée. Dans le cas contraire, l'évaluation de l'expression de type caractère est indéterminée.

Si l'expression de type caractère a la forme "'C'", sa valeur est le caractère C.

Si expr est une expression de type caractère quelconque, la valeur de l'expression de type caractère (expr), dans le contexte C et son état E(C), est la même que celle de expr (si elle existe) et les évaluations de ces deux expressions se déroulent de la même manière.

Exemples: Soient tabcar, un tableau de caractères et
i, une variable entière de C.

tabcar [i], 'A', (((tabcar [i]))) sont des expressions de type caractère

b. Expressions arithmétiques

Syntaxe :

<facteurl>:: = <expression de désignation de type entierl>|
<entier>|<appel de fonction de type entierl>|
(<expl arithmétique>)

<terme l>:: = <facteurl>|<termel><opérateur multiplicatif><facteurl>

<opérateur multiplicatif>:: = *|div|mod.

<expl arithmétique>:: = <termel>|
<expl arithmétique><opérateur additif><termel>|
- <termel>

<opérateur additif>:: = +|-

Sémantique

Soient C un contexte et E(C) son état,
fact , un facteur,
term , un terme et
expr , une expression arithmétique.

Soient encore, w_1,un opérateur multiplicatif et w_2, un opérateur additif.
Les expressions ci-dessous sont évaluées dans E(C) de la façon suivante:

fact: Si fact se résume - à une expression de désignation, on évalue cette dernière dans E(C).
Si cette évaluation est déterminée et si la variable qu'elle désigne a été initialisée, alors l'évaluation de l'expression fact est déterminée et sa valeur est la valeur de la variable identifiée.
Dans le cas contraire, l'évaluation de fact est indéterminée.
- à une constante formée d'une suite de chiffres, sa valeur est l'entier dont la suite est la représentation décimale.

Si expr est une expression arithmétique quelconque, la valeur de l'expression arithmétique (expr) dans E(C) est la même que celle de expr si elle existe) et les évaluations de ces deux expressions se déroulent de la même manière.

term w_1 fact: On évalue term et fact dans E(C). Si l'évaluation de l'une ou de l'autre est indéterminée, c'est aussi le cas pour l'expression term w_1 fact, sinon, notons v_1 et v_2 les valeurs (de type entierl) obtenues. Si le couple (v_1,v_2) vérifie la précondition de l'opération primitive w_1, la valeur de l'expression est v_1 w_1 v_2 (i.e. la valeur de l'opération primitive w_1 appliquée à (v_1,v_2).) sinon, l'évaluation de l'expression term w_1 fact est indéterminée.

expr w_2 term: On évalue expr et term dans E(C). Si l'évaluation de l'une ou l'autre est indéterminée, c'est aussi le cas pour l'expression expr w_2 term, sinon, notons v_1 et v_2 les valeurs (de type entierl) obtenues. Si le couple (v_1,v_2) vérifie la précondition de l'opération primitive w_2, la valeur de l'expression est $v_1w_2v_2$, sinon, l'évaluation de expr w_2 term est indéterminée.

- term: On évalue dans E(C) term. Si cette évaluation est indéterminée, c'est aussi le cas pour -term, sinon notons v la valeur ontenue (de type entierl); (-v) est la valeur de l'expression -term.

Exemples :

Soient x, i, t deux variables et un tableau d'entiers de C.
facteurs: x, 212, (x + t[i] div i), t[i]
termes; x, x * 212, x * 212 div (9 + t[i])
expressions arithmétiques: x, x * 212, x + x * 212 + 3, -x + t[i]

c. Expressions booléennes

Syntaxe

<expl booléenne simple>:: =<expression de désignation de type booléen>|
<valeur de vérité>|<appel de fonction de type bool>|
(<expl booléenne>)

<propositionl atomique>:: = <expl booléenne simple>|
<expl arithmétique><opérateur relationnel>
< expl arithmétique>|
<expl de type caractère><opérateur relationnel>
<expl de type caractère>

<opérateur relationnel>:: = =|<=|>=|<>|>|<

<négationl>:: = <propositionl atomique>|not <propositionl atomique>

<conjonctionl>:: = <négationl>|<négation l> and <conjonctionl>

<expl booléenne>:: = <conjonctionl>|< conjonctionl> or <expl booléenne>

Sémantique

Soient C un contexte et E(C) son état,
Sexpr, une expression booléenne simple,
aexpr1, aexpr2, deux expressions arithmétiques ou
deux de type caractère,
aprop, une proposition atomique,
neg, une négation,
conj, une conjonction,
bexpl, une expression booléenne et
w, un opérateur relationnel.

L'évaluation des expressions booléennes ci-dessous s'effectue, dans E(C), de la façon suivante:

Sexpr : Si sexpr se résume - en une expression de désignation, on évalue cette dernière dans E(C). Si cette évaluation est déterminée et si la variable qu'elle désigne a été initialisée, alors l'évaluation de l'expression sexpr est déterminée et sa valeur est la valeur de la variable désignée. Dans le cas contraire, l'évaluation de sexpr est indéterminée.

- en un des deux symboles spéciaux true ou false, sa valeur est la valeur de vérité correspondante, respectivement vrai ou faux.

Si expr est une expression booléenne quelconque, la valeur de l'expression booléenne (expr) dans E(C) est la même que celle de expr (si elle existe) et les évaluations de ces deux expressions se déroulent de la même manière.

aexpr1 w aexpr2 : On évalue aexpr1 et aexpr2 dans E(C). Si l'évaluation de l'une ou l'autre est indéterminée, alors l'évaluation de l'expression booléenne aexpr1 w aexpr2 l'est aussi, sinon, notons v_1 et v_2 les valeurs obtenues. Celles-ci sont de même type (entiers ou caractère). Si ces deux valeurs, v_1 et v_2 vérifient la précondition de l'opération primitive w, v_1 w v_2 est la valeur de l'expression aexpr1 w aexpr2 (i.e. la valeur de l'opération primitive w appliquée à (v_1,v_2)),sinon, l'évaluation de aexpr1 w aexpr2 est indéterminée.

not aprop: On évalue aprop dans E(C). Si cette évaluation est indéterminée, c'est aussi le cas de l'évaluation de not aprop, sinon notons v sa valeur. not v est la valeur de l'expression not aprop.

neg and conj: On évalue neg et conj dans E(C). Si l'évaluation de l'une ou l'autre est indéterminée, c'est aussi le cas de l'évaluation de neg and conj, sinon, notons v_1 et v_2 les valeurs obtenues (de type booléen). v_1 and v_2 est la valeur de l'expression neg and conj.

conj _or_ bexpr: On évalue conj et bexpr dans E(C). Si l'évaluation de l'une ou l'autre est indéterminée, c'est aussi le cas de l'évaluation de conj _or_ bexpr, sinon, notons v_1 et v_2 les valeurs obtenues (de type booléen). v_1 or v_2 est la valeur de l'expression conj _or_ bexpr.

Exemples

Soient x_1 et x_2 deux variables entières et bool une variable de type booléen de C.

propositions atomiques: _true_, bool, (bool _or_ $x_1 \leq x_2$), $x_1 \leq x_2$.

négations: bool, _not_ bool, $x_1 \leq x_2$, _not_ (bool _or_ $x_1 \leq x_2$)

conjonctions: bool, _not_ bool, $x_1 \leq x_2$ _and_ _not_ bool.

expressions booléennes: bool, _not_ bool _or_ $x_1 \leq x_2$ _and_ bool.

*

*   *

3.4. DECLARATIONS DE VARIABLES ET TABLEAUX

La déclaration des variables et tableaux du programme, d'une procédure ou d'une fonction, consiste en la création de ces variables et tableaux; création propre au corps du programme, de la procédure ou de la fonction en question. Ces déclarations contribuent à l'élaboration du contexte.

Syntaxe

<décl1 des variables et tableaux>:: = <vide>|var <liste1 de décl. de variables ou de tableaux>

<liste 1de décl de variables ou de tableaux>:: =
<décl1 de variables ou de tableaux>;|
<liste 1de décl de variables ou de tableaux><décl1 de variables ou de tableaux>;

<décl1 de variables ou de tableaux>:: = <décl1 de variables simples>|
<décl1 de tableaux>

<décl1 de variables simples>:: = <nom de variable>:<type1>|
<nom de variable>,<décl1 de variables simples>

<décl1 de tableaux>:: = <nom de tableau>: array [<entier>..<entier>]
of <type1>|<nom de tableau>, <décl1 de tableaux>

<nom de variable>:: = <identificateur>
<nom de tableau>:: = <identificateur>

<type1>:: = integer|boolean|char

De plus, pour que la déclaration de variable ou de tableau de nom x soit syntaxiquement correcte dans le contexte C, il faut que ce contexte ne contienne pas encore de variables ou tableaux de nom x.

Sémantique

La déclaration d'une variable x entraîne la modification de C par l'ajout de cette nouvelle variable et la mise à jour de E(C) en renseignant que la variable n'est pas initialisée. Cela se déroule d'une façon analogue s'il s'agit d'un tableau.

Exemples

Soient x, t_1, t_2, t_3 des identificateurs n'étant pas des noms de variables ou tableaux de C.

Déclaration de variables et tableaux:

var x : <u>integer</u>;

t_1,t_2 : <u>array</u> [1..10] <u>of</u> <u>char</u>;

t_3 : <u>array</u> [12..123] <u>of</u> <u>boolean</u>;

*

*   *

3.5. INSTRUCTIONS (INTRODUCTION AUX DEUX TYPES D'EXECUTION)

Une instruction est une construction du langage qui, étant donné un contexte C et son état, spécifie une suite d'opérations à effectuer et certains points de contrôle d'exécution appelés assertions. Cette suite d'opérations est susceptible de modifier l'état du contexte C.

L'exécution sans vérification des assertions de cette instruction consiste à exécuter cette suite d'opérations sans considérer les assertions. L'exécution avec vérification d'assertions consiste à exécuter cette même suite d'opérations mais en évaluant à chaque point de contrôle la valeur de l'assertion qui doit être la valeur de vérité vrai. Dans ces deux types d'exécution, l'évaluation d'une des opérations peut ne pas être déterminée si certaines conditions imposées par la sémantique ne sont pas vérifiées. On dit, dans ce cas, que l'exécution de l'instruction est indéterminée. De plus, une exécution déterminée peut ne pas se terminer. On dit alors que l'exécution ne se termine pas. Enfin, lorsque l'exécution est déterminée et se termine, elle modifie de manière bien déterminée l'état du contexte C. Ce nouvel état est le résultat de l'exécution de l'instruction.

Syntaxe générale

<instruction1>:: = <instr1 d'affectation>|
<instr1 conditionnelle>|
<instr1 répétitive>|
<instruction de lecture>|
<instruction d'écriture>|
<instr1 de branchement>|
<appel1 de procédure>|
<instr1 composée>

Remarque: Les cinq premières classes d'instructions seront exposées dans ce paragraphe; à leur niveau n'interviennent pas les différences dues au type d'exécution choisi. C'est d'ailleurs pourquoi, dans ce

paragraphe, on parlera simplement d'exécution sans plus préciser son type. Au contraire, c'est dans les instructions composées que se marque cette différence; elles feront l'objet du paragraphe suivant. Enfin, c'est au paragraphe 3.7 que seront exposées les instructions de branchement.

a. Instructions d'affectation

Syntaxe

<instr1 d'affectation>:: = <expression de désignation>: = <expression 1>

De plus, l'expression de désignation et l'expression à droite du symbole := doivent être de même type.

Sémantique

Soit C, un contexte.
Soient encore dexpr, une expression de désignation
et expr une expression de même type.

L'exécution dans E(C) de l'instruction d'affectation despr := expr se déroule comme suit:

On évalue l'expression de désignation dexpr et l'expression expr dans E(C). Si l'évaluation de l'une ou l'autre est indéterminée, l'exécution de l'instruction dexpr := expr l'est également. Dans le cas contraire, notons x la variable désignée par la première et v la valeur calculée par la seconde. L'exécution de l'instruction s'achève en affectant à x la valeur v.

Exemple

Soient t, x, y, z un tableau et des variables entières de C.

instructions d'affectation: t[x] := y * (z+3)
y := y-1

b. Instructions conditionnelles

Syntaxe

<instrl conditionnelle>::=

if <exprl booléenne> then <instructionl>| (1ère forme)
if <exprl booléenne> then <instructionl>
else <instructionl> (2ème forme)

Sémantique

Soient C un contexte,
bexpr, une expression booléenne et
ins1, ins2, deux instructions.

Considérons les deux expressions conditionnelles suivantes:
if bexpr then ins1 (1ère forme)
if bexpr then ins1 else ins2 (2ème forme)

L'exécution de ces instructions conditionnelles s'effectue comme suit: On évalue bexpr dans E(C). Si cette évaluation est indéterminée, c'est aussi le cas de l'instruction conditionnelle considérée. Dans le cas contraire, notons v la valeur de type booléen qu'elle renvoie. Si v est la valeur de vérité true, exécuter ins1 dans E(C) sinon
(1ère forme) $\Longrightarrow$ L'exécution de l'instruction conditionnelle est terminée.
(2ème forme) $\Longrightarrow$ Exécution ins2 dans E(C).

Remarque: Une instruction conditionnelle de la forme if bexpr1 then if bexpr2 then ins1 else ins2 est considérée comme étant de la première forme (comme en Pascal).

Exemples: Soient t et x, un tableau et une variable entière de C.
Instructions conditionnelles: if x $\geqslant$ 0 then t[x]:= t[x] * 2
if x $\geqslant$ 0 then t[x]:= 1 else x := x+1

c. Instructions répétitives

Syntaxe

<instrl répétitive>::= while <exprl booléenne> do <instructionl>

Sémantique

Soient C, un contexte,
bexpr, une expression booléenne et
ins, une instruction.

L'exécution de l'instruction while bexpr do ins dans E(C) se déroule comme suit: on évalue bexpr dans E(C). Si cette évaluation est indéterminée, c'est aussi le cas de l'instruction répétitive. Dans le cas contraire, si elle se termine, notons v la valeur booléenne qu'elle renvoie. Si v est la valeur de vérité false, l'exécution de l'instruction répétitive se termine, sinon, on exécute, dans E(C), insl puis l'instruction répétitive while bexpr do ins.

Exemple:

Soient t et x, un tableau et une variable entière de C;
Instruction répétitive: while t[x] <> 0 do x := x+1

d. Instructions de lecture

Syntaxe

<instruction de lecture> ::= read(<expression de désignation>)

De plus, l'expression de désignation doit être de type caractère.

Sémantique

Soient C, un contexte,
dexpr, une expression de désignation de type caractère et
α, l'état du fichier d'entrée.

L'exécution de l'instruction read (dexpr) dans E(C) se déroule comme suit: on évalue l'expression de désignation dexpr dans E(C). Si cette évaluation est indéterminée, c'est aussi le cas de l'exécution de l'ins-

truction de lecture, sinon, notons x la variable qu'elle désigne. Si α est vide, l'exécution de l'instruction est indéterminée, sinon, soient c, le premier caractère de α et β, la suite des caractères suivants. Le caractère c est "lu" sur le fichier d'entrée. La suite β devient le nouvel état du fichier d'entrée de C. La valeur c est affectée à la variable x.

Exemple:

Soient x, une variable entière et
t, un tableau de caractères de C.

Instruction de lecture : read (t[2 * x+1])

L'instruction de lecture du langage L1 est assez pauvre comparée à celles fournies par un langage comme Pascal. Pour combler cette lacune, le langage L1 propose une procédure prédéfinie permettant la lecture d'entiers:

read (var x : integer ; var bool : boolean) (☆)

dont la spécification est la suivante: Lecture (de caractères) sur le fichier d'entrée jusqu'au moment où le caractère lu est le caractère ' '. Notons n, le nombre des caractères lus. Si la suite des (n-1) premiers caractères lus est la représentation décimale d'un entier, on affecte à bool la valeur de vérité true et à x, cet entier. Dans le cas contraire, on affecte à bool, la valeur de vérité false.

e. Instructions d'écriture

Syntaxe

<instruction d'écriture>::= write (<expl de type caractère>) |
write (<expl arithmétique>)

Sémantique

Soient C, un contexte
expr, une expression de type caractère (arithmétique)
et α, l'état du fichier de sortie de C.

(☆) Ces notations seront explicitées au paragraphe 3.8.

L'exécution de l'instruction <u>write</u> (expr) dans C se déroule comme suit: on évalue, dans E(C), l'expression de type caractère (arithmétique) expr. Si cette évaluation est indéterminée, c'est aussi le cas de l'exécution de l'instruction d'écriture, sinon, notons v la valeur qu'elle renvoie. On "écrit" le caractère (la représentation décimale de) v sur le fichier de sortie. La suite α c devient donc le nouvel état du fichier de sortie, où c est le caractère (la représentation décimale) en question.

<u>Exemples</u>: Soient x, une variable entière et
t, un tableau de caractères de C.

Instructions d'écriture: write (3 * x+2)
write (t[x])

*

* *

3.6. INSTRUCTIONS COMPOSEES, INTERFACES POUR ASSERTIONS, EXECUTION AVEC ET SANS VERIFICATION D'ASSERTIONS

Par instruction composée, nous entendons instruction composée d'instructions et d'assertions. L'instruction composée est un concept primordial du langage L1 puisque, d'une part, le corps d'un programme avec assertions, d'une fonction ou d'une procédure L1, est une instruction composée et, d'autre part, c'est dans ce type d'instructions qu'interviennent les assertions et c'est donc dans ce paragraphe que seront spécifiés les deux types d'exécutions permises par ce langage.

Syntaxe

<instr1 composée> ::= begin <liste d'instructions et d'assertions> end|
    begin : <identificateur><liste d'instructions et d'assertions> end
<liste d'instructions et d'assertions> ::= <instruction1><suite d'assertions>|<instruction1>;<liste d'instructions et d'assertions>|
    <assertion><liste d'instructions et d'assertions>
<suite d'assertions> ::= <vide>|<assertion><suite d'assertions>
<assertion> ::= {*<énoncé d'assertion> *}
<énoncé d'assertion> : exprimé en L2 (cfr chapitre IV)

Pour que l'instruction composée begin : x ... end soit syntaxiquement correcte dans le contexte C, l'identificateur x ne peut être le nom d'une étiquette de C.

Sémantique

Soient C, un contexte,
ins1, ... insn des instructions, (n > 0) et
suite 1, ... suite n+1, des suites d'assertions.

L'exécution sans vérification d'assertions de l'instruction begin suite1 ins1; ... insn suiten+1 end dans E(C) consiste à exécuter sans vérification d'assertions successivement chacune des instructions (☆) ins1,

(☆) Voir remarque page suivante.

ins2, ... insn, dans E(C). Si l'exécution de l'une d'entre elles est indéterminée, c'est aussi le cas de l'exécution de l'instruction composée.

L'exécution avec vérification d'assertions de cette même instruction composée se déroule comme suit:

\# 1 : faire i ← 1

\# 2 : si la suite d'assertions suite i est vide, aller en # 3, sinon évaluer successivement dans E(C') (où C' est le contexte des assertions de cette instruction) les assertions de la suite suite i (☆). Si l'évaluation de l'une d'entre elles est indéterminée ou renvoie la valeur faux, aller en # 4.

\# 3 : si i = n+1, l'exécution avec vérification d'assertions de l'instruction composée se termine, et est déterminée, sinon, exécuter avec vérification d'assertions insi dans E(C). Si cette exécution est indéterminée, aller en # 4 sinon, faire i ← i+1 et aller en # 2.

\# 4 : L'exécution avec vérification d'assertions de l'instruction composée est indéterminée.

Soit, de plus, x, un identificateur n'étant pas le nom d'une étiquette du contexte C. L'exécution dans E(C) (avec ou sans vérification d'assertions) de l'instruction composée <u>begin</u> : x suite1 ins1; ... <u>end</u> se déroule de la manière suivante:

\# 1 : considérer le contexte C' construit à partir de C en lui ajoutant l'étiquette de nom x en l'associant à l'instruction composée concernée.

\# 2 : Exécuter, dans E(C'), l'instruction composée <u>begin</u> suite1 ins1; ... <u>end</u> (avec ou sans vérification d'assertions suivant le cas).

\# 3 : Si celle-ci se termine et est déterminée, l'exécution de l'instruction composée considérée l'est également et se termine en supprimant du contexte C', l'étiquette de nom x.

<u>Remarque importante</u>

La sémantique de l'exécution d'une instruction composée est présentée ci-dessus comme étant la séquence des exécutions de toutes les instructions

---

(☆) Cfr chapitre IV.

ins1, ... insn qui la compose (avec ou sans vérification d'assertions entre deux d'entre elles). Mais ce n'est pas toujours le cas ! En effet, cette séquence peut être interrompue à cause de l'exécution de l'une de ces instructions. Il s'agit des instructions de branchement qui font l'objet du paragraphe suivant.

Exemple: Cfr instructions debranchement, page suivante.

*

*   *

3.7. INSTRUCTIONS DE BRANCHEMENT

Syntaxe

<instrl de branchement> ::= goto <identificateur>| exit <identificateur>

De plus, pour que l'instruction de branchement soit syntaxiquement correcte dans le contexte C, l'identificateur concerné doit être le nom d'une étiquette de C.

Sémantique

Soient C, un contexte et x, un nom d'étiquette de C.
Comme x est un nom d'étiquette de C, il est associé à une instruction composée qu'on est en train d'exécuter. Notons cette instruction ins. L'exécution de l'instruction exit x entraîne la fin de l'exécution de ins en retirant du contexte C l'étiquette de nom x et toutes les étiquettes associées à des instructions internes à ins. L'exécution de l'instruction goto x se déroule de la façon suivante:

1 : exécuter exit x; notons C le contexte résultant;
2 : (Nouvelle) exécution de ins dans E(C)

Exemple: Soient x et t, une variable et un tableau d'entiers de C.

Instruction composée (avec instructions de branchement)

```
begin : étiq
      if x > longtab then exit étiq;
      t[x] := 2 * t[x];
      x := x+1;
     goto étiq
end
```

*

*   *

3.8. DECLARATIONS ET APPELS DE FONCTIONS OU DE PROCEDURES

a. Déclarations de fonctions ou de procédures

Syntaxe

<déclaration de fonction 1>::= <en-tête de fonction1><déclaration des variables et tableaux><corps1>

<déclaration de procédure 1>::= <en-tête de procédure 1><déclaration des variables et tableaux><corps1>

<en-tête de fonction 1> ::= function 1 <nom de fonction>(<spécification des arguments formels>): <type 1><précondition>;

<nom de fonction> ::= <identificateur>

<spécification des arguments formels> ::= <groupe d'arguments formels>| <spécification des arguments formels>; <groupe d'arguments formels>

<groupe d'arguments formels> ::= <argument formel>:<type 1>| <argument formel>,<groupe d'arguments formels>

<argument formel> ::= <nom de variable>

<en-tête de procédure 1> ::= procédure 1 <nom de procédure>;| procédure 1 <nom de procédure>(<spécification des paramètres formels>) <précondition>;

<nom de procédure> ::= <identificateur>

<spécification des paramètres formels> ::= <groupe de paramètres formels>| <spécification des paramètres formels>;<groupe de paramètres formels>

<groupe de paramètres formels> ::= <liste de paramètres formels>:<type 1>| var <liste de paramètres formels>:<type 1>
var <liste de paramètres formels>: array [<entier>..<entier>] of <type 1>

<liste de paramètres formels> ::= <paramètre formel>| <liste de paramètres formels>, <paramètre formel>

<paramètre formel> ::= <nom de variable>|<nom de tableau>

<corps 1> ::= <instr1 composée>

<type 1> ::= integer|boolean|char

<précondition> ::= <vide>|précondition: <énoncé de précondition>

<énoncé de précondition>: cfr chapitre IV car doit être eprimée en L2.

De plus, pour qu'une déclaration de fonction ou de procédure soit syntaxiquement correcte dans le contexte C, il faut que son nom ne soit pas encore le nom d'une fonction ou d'une procédure de C.

Sémantique

Une déclaration de fonction ou de procédure entraîne l'ajout de celle-ci au contexte concerné.

b. Appels de fonctions ou de procédures

Un appel de fonction est une expression et son évaluation produit une valeur.

Un appel de procédure est une instruction et son exécution, dans un certain contexte, modifie l'état de ce contexte.

Syntaxe

a. <appel de fonction1> ::= <nom de fonction>(<liste d'arguments effectifs>).
<liste d'arguments effectifs> ::= <argument effectif>|
<liste d'arguments effectifs>,<argument effectif>
<argument effectif> ::= <expression1>
<appel de procédure 1> ::= <nom de procédure>|<nom de procédure>
(<liste de paramètres effectifs>)
<liste de paramètres effectifs> ::= <paramètre effectif>|
<liste de paramètres effectifs>,<paramètre effectif>
<paramètre effectif> ::= <expression>|<expression de désignation>|
<expression de désignation de tableau>

b. Remarques syntaxiques supplémentaires.

Soient C, un contexte,
p, un identificateur,
eff_1..,eff_n des paramètres (ou arguments) effectifs ($n \geq 0$)

Pour être syntaxiquement correct dans le contexte C, l'appel de procédure (fonction) p(eff_1, ... eff_n) (ou p si n = 0) doit vérifier quelques règles

supplémentaires: l'identificateur p doit être le nom d'une procédure (fonction) du contexte C. Soit dp, la déclaration de procédure (fonction) en question et soient forl, ;.. form la suite des identificateurs figurant, dans cet ordre, comme paramètres (arguments) formels dans la déclaration dp. Pour que cet appel soit syntaxiquement correct, il faut, de plus,

a. que cette suite soit de même longueur que la suite eff1...effn (i.e. n=m);
b. que les identificateurs forl, ... forn soient distincts et même qu'ils soient distincts de p si dp est une déclaration de fonction;
c. si for i figure dans un groupe de paramètres (arguments formels de dp qui a la forme ..., fori, ...: ty, le paramètre effectif effi doit être une expression de type ty (on dit alors qu'il s'agit d'un appel par valeur);
d. si fori figure dans un groupe de paramètres formels de dp qui a la forme <u>var</u> ... fori, ...: ty, le paramètre effectif effi doit être une expression de désignation de type ty (on dit alors qu'il s'agit d'un appel par adresse);
e. si fori figure dans un groupe de paramètres formels de dp, qui a la forme <u>var</u> ... fori, ...: <u>array</u> [k..l] <u>of</u> ty, le paramètre effectif effi doit être le nom d'un tableau du contexte C de type ty et d'indice [k..l]. (On dit alors qu'il s'agit d'un tableau appelé par adresse.)

Notations: La notation v : m caractérise une variable de nom v et de mémoire m. De même, la notation t : M[k..l] caractérise un tableau de nom t dont M est la bijection de ses indices [k..l] dans l'ensemble de ses éléments.

<u>Sémantique</u>

Sous l'hypothèse d'un appel de procédure (fonction) syntaxiquement correct et en utilisant les notations de la rubrique syntaxe ci-dessus, l'exécution (l'évaluation) de l'appel de procédure (fonction) p(effi,... effn) dans le contexte C, se déroule comme suit:

\# 1 : C' ← contexte constitué de toutes les fonctions et procédures L1 déclarées dans le contexte C.

Si dp est une déclaration de fonction, on crée une mémoire m du type de la fonction (spécifié dans dp) et on ajoute au contexte C', la variable p:m. Faire i ← 1.

\# 2 : Si i > n, aller en # 7.

Considérer les $i^{èmes}$ paramètres (arguments), formel et effectif, des suites for1, ... forn et eff1, ... effn. S'il s'agit d'un appel par valeur, aller en # 3, d'un appel par adresse, aller en # 4 ou d'un tableau appelé par adresse, aller en # 5.

\# 3 : Evaluation de l'expression effi dans le contexte C. Si celle-ci est indéterminée, aller en # 8, sinon, notons v sa valeur de type ty; on crée une nouvelle mémoire m de type ty et on ajoute la variable fori : m ayant la valeur v, au contexte C'. Aller en # 6.

\# 4 : Evaluation de l'expression de désignation effi dans le contexte C. Si cette évaluation est indéterminée, aller en # 8, sinon, notons m, la mémoire de la variable identifiée. Ajouter ensuite à C' la variable fori : m. Aller en # 6.

\# 5 : Evaluation de l'expression de désignation de tableau effi.

Soit M[k..l], la bijection de l'ensemble des indices du tableau dans l'ensemble de ses éléments; on ajoute à C' le tableau fori: M[k..l]. Aller en # 6.

\# 6 : Faire i ← i+1 et aller en # 2.

\# 7 : a. S'il s'agit d'une exécution avec vérification d'assertions et si la déclaration de la procédure (fonction) mentionne une précondition, on évalue cette dernière; si cette évaluation est indéterminée ou renvoie la valeur faux, aller en # 8.

b. Analyse dans C' des déclarations de variables et tableaux de dp qui a pour conséquence l'ajout de ceux-ci au contexte C' (cfr paragraphe 3.3).

c. Exécution de l'instruction composée qui est le corps de la procédure (fonction) dans le contexte C'. Si elle est indéterminée, aller en # 8, sinon, si elle se termine, considérons les deux cas envisagés en parallèle:

- S'il s'agit d'un appel de procédure, l'instruction qui consiste en cet appel est déterminée et se termine.

- S'il s'agit d'un appel de fonction, considérer la valeur de la variable p. Si celle-ci est indéterminée, aller en # 8, sinon, l'expression qui consiste en cet appel est déterminée, se termine et renvoie cette valeur.

8 : L'exécution (l'évaluation) de l'appel de procédure (fonction) est indéterminée.

Remarque : LE LANGAGE L1 NE PERMET PAS D'APPELS RECURSIFS !

Exemples : Soient x et t, une variable entière et un tableau d'entiers d'indices [1..5];

a. déclaration de fonction:

```
functionl cube (i : integer) : integer;
begin
  cube := i * i * i
end
```

b. déclaration de procédure:

```
procedurel carré (var tab: array [1..5] of integer; i: integer)
  précondition: 1 ≤ i and i ≤ 5;
begin
  tab [i]:= tab[i] * tab[i]
end
```

c. appel de fonction:

```
x := cube(x) + 3 x + 1
```

d. appel de procédure:

```
carré (t, 3 x + 2)
```

Chapitre IV

LE LANGAGE L2

4.1. INTRODUCTION, PARTIES CONCERNEES ET SYMBOLES SPECIAUX

Comme l'indique la première partie de ce mémoire, le langage de programmation d'assertions L2 se doit d'être plus riche que le langage L1. Ceci est essentiellement réalisé de deux façons:

1. Possibilité d'exprimer des expressions plus complexes. Le niveau de complexité sera choisi en fonction des constatations suivantes; remarquons notamment qu'au plus élevé sera ce niveau, au plus vaste sera la classe des problèmes réalisables dans ce langage. Par contre, étant trop élevé, il rendrait difficile la maîtrise de celui-ci. Pour faciliter cette maîtrise, tout en permettant l'expression d'une classe assez vaste de problèmes, le langage L2 sera basé sur le formalisme de la logique des prédicats du premier ordre.

2. Possibilité d'utiliser des fonctions prédéfinies. Celles-ci doivent être suffisamment générales pour être utilisables dans une grande partie des problèmes à traiter. On doit, bien sûr, se rendre compte qu'il ne sera pas possible d'exprimer, d'une façon simple, tous les problèmes de la classe prévue (à moins de ne permettre une fonction prédéfinie particulière pour chaque problème rencontré, ce qui fausserait totalement le but de ce travail !).

En vue de satisfaire ces exigences, il nous a semblé intéressant d'introduire de nouveaux types de valeurs, générés à partir des types du langage L1. Ces types et leurs opérations primitives font l'objet du paragraphe suivant. Le troisième paragraphe indique comment, grâce à ceux-ci, construire des expressions plus compliquées.

Les langages L1 et L2 ont beaucoup de similitudes; c'est notamment le cas en ce qui concerne les déclarations de variables et tableaux et les déclarations et appels de fonctions ou procédures. Celles-ci sont respectivement envisagées aux paragraphes 4 et 7 qui n'explicitent que les différences face au langage L1.

Nous indiquons ensuite qu'il est intéressant que l'utilisateur puisse définir certaines fonctions qu'il utiliserait dans les assertions. Le langage L2 devient dès lors un langage de programmation, avec, notamment, des mécanismes permettant différentes instructions; ces dernières font l'objet du cinquième paragraphe.

Comme les langages L1 et L2 sont destinés à être utilisés ensemble, un interface est nécessaire; la partie syntaxe de ce dernier a déjà été définie au chapitre III; reste sa partie sémantique qui sera envisagée au paragraphe 6 de ce chapitre.

Le paragraphe 8, quant à lui, tente de satisfaire la seconde exigence en proposant un ensemble de fonctions prédéfinies.

Nous montrons ensuite que, pour être utile, une assertion doit être exprimée en fonction des valeurs qu'ont eues certaines variables, d'où l'intérêt de pouvoir les mémoriser.

Nous terminons enfin ce chapitre en montrant l'utilité de deux autres fonctions prédéfinies dont le statut diffère de celles présentées précédemment. Ces deux dernières fonctions clôturent l'ensemble des outils proposés par le langage L2 pour permettre l'expression d'assertions.

<u>Parties concernées par le langage L2</u>

1. Les énoncés d'assertions,
2. les énoncés de préconditions et
3. les déclarations des fonctions et procédures L2.

Symboles spéciaux du langage L2

<symbole spécial> ::= +|-| |(|)|=|,|.|<|≥|:|;| | |}|{|||
all|and|array|be|begin|boolean|char|def|div|do|
else|end|exists|exit|false|for|function2|goto|if|in|
variant|integer|let|max|min|mod|not|of|or|
procedure2|prod|read|sum|term|then|there|true|var|
while|write

*

* *

4.2. TYPES DE VALEURS ET OPERATIONS PRIMITIVES ASSOCIEES

a. Types de base

Type entier2

Soit Z, l'ensemble des entiers;

a. Ensemble des objets de type entier2 ≡ Z'

où $Z' = Z \cup \{zmin, zmax\}$;

zmin et zmax symbolisent respectivement l'infini négatif et positif.

b. Opérations primitives.

Addition : $Z' \times Z' \longrightarrow Z'$

$(a,b) \leadsto a+b$

précondition: a et b ∉ {zmin, zmax}

Soustraction : $Z' \times Z' \longrightarrow Z'$

$(a,b) \leadsto a-b$

précondition: a et b ∉ {zmin, zmax}

Multiplication: $Z' \times Z' \longrightarrow Z'$

$(a,b) \leadsto a*b$

précondition: a et b ∉ {zmin, zmax}

Division entière et reste: $Z' \times Z' \longrightarrow Z'$

$(a,b) \leadsto$ a div b

$(a,b) \leadsto$ a mod b

précondition: a et b ∉ {zmin, zmax} et b ≠ 0

Egalité: $Z' \times Z' \longrightarrow B = \{vrai, faux\}$

$(a,b) \leadsto c$

précondition: (a,b) ∉ {(zmin, zmin), (zmax, zmax)}

Table de l'égalité

=	y	zmin	zmax
x	(x=y)	faux	faux
zmin	faux	////	faux
zmax	faux	faux	////

où x et y ne sont ni zmin, ni zmax et où (x=y) vaut la valeur de vérité vrai si l'entier x égale l'entier y et faux dans le cas contraire.

Inégalité < : $Z' \times Z' \longrightarrow B = \{vrai, faux\}$

$(a,b) \rightsquigarrow c$

précondition: $(a,b) \notin \{(zmin,zmin),(zmax,zmax)\}$

Table de l'inégalité <

<	y	zmin	zmax
x	(x<y)	faux	vrai
zmin	vrai	////	vrai
zmax	faux	faux	////

où x et y ne sont ni zmin, ni zmax et où (x y) vaut la valeur de vérité vrai si l'entier x est plus petit strictement que l'entier y et faux dans le cas contraire.

D'une manière analogue pour les autres types de comparaisons.

<u>zmin</u> et <u>zmax</u> : $Z' \longrightarrow B$

a $\rightsquigarrow$ <u>zmin</u>(a)

a $\rightsquigarrow$ <u>zmax</u>(a)

où <u>zmin</u>(a) est la valeur de vérité vrai ssi a est la valeur zmin et où <u>zmax</u>(a) est la valeur de vérité vrai ssi a est la valeur zmax.

<u>Remarque</u>: Le type <u>entier2</u> est une généralisation du type <u>entier1</u> puisque

a. $Z \subset Z'$

b. les opérations primitives du type <u>entier1</u> sont les restrictions de celles du type <u>entier2</u> à l'ensemble Z.

D'où, un objet de type <u>entier1</u> peut être considéré comme de type <u>entier2</u>.

<u>Type caractère et booléen</u> : cfr chapitre III.

b. Types générés à partir des types de base

Ces types sont basés sur les notions de suite et d'ensemble. Celles-ci sont fréquemment utilisées par tout informaticien et sont, généralement, bien intériorisées. Elles interviennent dans beaucoup de raisonnements, notamment parmi ceux par récurrence.

Type suite d'entiers(☆)

Soit S(Z), l'ensemble des suites finies d'éléments de type entierl.

a. Ensemble des objets de type suite d'entiers ≡ S(Z)
b. Opérations primitives: cfr "opérations primitives sur les suites".

Type suite de caractères

Soit S(C), l'ensemble des suites finies d'éléments de type caractère.

a. Ensemble des objets de type suite de caractères ≡ S(C)
b. Opérations primitives: cfr "opérations primitives sur les suites".

Type suite de booléens

Soit S(B), l'ensemble des suites finies de valeurs de vérité.

a. Ensemble des objets de type suite de booléens ≡ S(B)
b. Opérations primitives: cfr "opérations primitives sur les suites".

Type ensemble d'entiers (☆)

Soit E(Z), l'ensemble des ensembles de cardinal fini d'éléments de type entierl.

a. Ensemble des objets de type ensemble d'entiers ≡ E(Z)
b. Opérations primitives: cfr "opérations primitives sur les ensembles".

(☆) Les motivations sous-jacentes au choix de ne pas permettre les valeurs zmin et zmax dans une suite ou un ensemble d'entiers sont présentées au chapitre II de la première partie de ce mémoire.

Type ensemble de caractères

Soit E(C), l'ensemble des ensembles de cardinal fini d'objets de type caractère.

a. Ensemble des objets de type ensemble de caractères ≡ E(C)

b. Opérations primitives: cfr "opérations primitives sur les ensembles".

Type ensemble de booléens

Soit E(B), l'ensemble des ensembles de cardinal fini d'objets de type booléen

a. Ensemble des objets de type ensemble de booléens ≡ E(B)

b. Opérations primitives: cfr "opérations primitives sur les ensembles".

Notations : En vue de ne pas se répéter inutilement, on utilisera par la suite la notation "ty" pour exprimer un des trois types entierl, booléen ou caractère. De même, par la notation TY, on exprimera un des trois ensembles Z, B ou C.

Opérations primitives sur les ensembles

Soient A et B, deux ensembles d'éléments de type ty.

a. vide : E(TY) ⟶ B
A ⇝ vide(A)

L'opération vide appliquée à un ensemble A renvoie le booléen vrai si A est vide et faux sinon.

b. Union : E(TY) x E(TY) ⟶ E(TY)
(A , B) ⇝ A ∪ B

opération traduisant l'union ensembliste.

c. Différence : E(TY) x E(TY) ⟶ E(TY)
(A , B) ⇝ A/B

opération traduisant la différence ensembliste.

d. <u>Elt</u> : E(TY) ⟶ TY

A ⟿ élt(A) précondition : vide(A) = false

opération qui appliquée à un ensemble non vide A renvoie un élément (quelconque) de cet ensemble

<u>Opérations primitives sur les suites</u>

Soient S1 et S2, deux suites d'éléments de type ty.

a. <u>vide</u> : S(TY) ⟶ B

S1 ⟿ vide(S1)

opération qui, appliquée à une suite S1, renvoie le booléen vrai si S1 est la suite vide et faux sinon.

b. <u>premier</u> : S(TY) ⟶ TY

S1 ⟿ premier(S1) précondition: vide(S1) = false

opération qui renvoie le premier élément d'une suite S1 non vdie.

c. <u>reste</u> : S(TY) ⟶ S(TY)

S1 ⟿ reste(S1) précondition: vide(S1) = false

opération qui, appliquée à une suite non vide S1, renvoie la suite reste(S1) constituée de tous les éléments de S1 sauf le premier.

Ex : S1 = [1,2,3,4] reste(S1) = [2,3,4]

d. <u>concat</u>: S(TY) x S(TY) ⟶ S(TY)

(S1 , S2) ⟿ concat(S1,S2)

opération qui, appliquée à une suite S1 et une autre S2, renvoie la concaténation de ces deux suites.

Ex: concat ([1,2],[1,3,4]) = [1,2,1,3,4]

Notations: Soit v, un objet de type ty.

a. On notera par {v}, l'ensemble d'éléments de type ty ayant v pour seul élément.

b. On notera par [v], la suite d'éléments de type ty ayant v pour seul élément.

c. La notation A ← val exprime le fait que A sera, dès ce moment, la notation symbolisant la valeur val.

Ex: A ← {v} indique qu'à partir de ce moment, A est l'ensemble ayant v pour unique élément.

4.3. EXPRESSIONS (sauf les appels de fonctions)

Syntaxe générale

<expression2> ::= <exp2 arithmétique>|<exp2 de type caractère>|<exp2 booléenne>|
<exp2 de type ensemble d'entiers>|<exp2 de type suite d'entiers>|<exp2 de type ensemble de bool>|<exp2 de type suite de bool>|<exp2 de type ensemble de car>|<exp2 de type suite de car>

a. Expressions arithmétiques

Syntaxe

<facteur2> ::= <entier>|<expression de désignation de type entier2>|
<appel de fonction de type entier2>|
max for <identificateur> in <exp2 de type ensemble d'élt. de type ty>
of (<exp2 arithmétique>)|
min for <identificateur> in <exp2 de type ensemble d'élt. de type ty>
of (<exp2 arithmétique>)|
sum for <identificateur> in <exp2 de type suite d'élt. de type ty>
of (<exp2 arithmétique>)|
prod for<identificateur> in<exp2 de type suite d'élt. de type ty>
of (<exp2 arithmétique>)

De plus, pour que l'expression max for id in... soit syntaxiquement correcte dans le contexte C, l'identificateur id ne peut être le nom d'une variable ou d'un tableau du contexte C. De même pour les expressions min for id..., sum for id ... et prod for id in ... .

(☆) ty est une notation pour ne pas devoir tripler les règles B.N.F. en indiquant qu'il s'agit successivement d'ensembles (ou de suites) d'entiers, de caractères ou de booléens.

<terme2> ::= <facteur 2.>|<terme2><opérateur multiplicatif><facteur2>

<opérateur multiplicatif> ::= * |div|mod.

<exp2 arithmétique> ::= <terme2>|<exp2 arithmétique><opérateur additif>
<terme 2.>| - <terme 2.>.

<opérateur additif> ::= +|-

Sémantique

Soient C, un contexte et E(C) son état,
fact, un facteur,
term, un terme,
expr, une expression arithmétique,
i, un identificateur (qui n'est pas le nom d'une variable ou d'un tableau de C),
ens, une expression de type ensemble d'élt. de type ty,
suite, une expression de type suite d'élt. de type ty.

Soient encore w1 et w2 respectivement un opérateur multiplicatif et additif.

Les expressions décrites ci-dessous sont évaluées dans E(C) de la façon suivante:

Si fact se résume à - une expression de désignation: cfr chapitre III
- une constante: cfr chapitre III
- (expr): cfr chapitre III.

Si fact est le facteur max (*min*) for i in ens of (expr), son évaluation se déroule de la façon suivante:

1 : Evaluer l'expression ens dans E(C). Si cette évaluation est indéterminée, c'est aussi le cas de l'évaluation de fact, sinon, notons A la valeur trouvée (de type ensemble d'élt. de type ty) et vfact, la valeur zmin (*zmax*). Notons de plus, C', le contexte construit à partir de C en lui ajoutant la variable i de type ty.

2 : Si l'ensemble A est vide, aller en # 5, sinon, notons v la valeur élt(A) de type ty. Affecter ensuite, à la variable i, la valeur v.

\# 3 : Evaluer exp dans E(C'). Si cette évaluation est indéterminée, c'est aussi le cas de l'évaluation de fact, sinon, notons r la valeur (de type <u>entier2</u>) obtenue.
Trois cas sont à distinguer:
- si <u>zmin</u> (*zmax*) (r) ⟶ rien faire
- si <u>zmax</u> (*zmin*) (r) ⟶ faire vfact ← zmax (*zmin*)
- si entier (r) ⟶ faire si r < vfact (> *vfact*), vfact ← r.

\# 4 : faire A ← A/{v} et aller en # 2.

\# 5 : L'évaluation de l'expression fact se termine et est déterminée, sa valeur est vfact.

Si fact est le facteur <u>sum</u> (*<u>prod</u>*) <u>for</u> i <u>in</u> suite <u>of</u> (exp), son évaluation se déroule de la façon suivante:

\# 1 : Evaluer l'expression suite dans E(C). Si cette évaluation est indéterminée, c'est aussi le cas de l'évaluation de fact, sinon, notons S la valeur(suite d'élt. de type ty) trouvée et vfact, une valeur de type entier2, initialement 0 (*1*). Notons encore, C', le contexte construit à partir de C en lui ajoutant la variable i de type ty.

\# 2 : Si la suite S est vide, aller en # 5, sinon, notons v, la valeur premier(S) de type ty. Affectons ensuite, à la variable i, la valeur v.

\# 3 : Evaluer l'expression arithmétique vfact + (*) (exp) dans E(C'). Si cette évaluation est indéterminée, c'est aussi le cas pour l'évaluation de fact, sinon, notons r, la valeur obtenue.
Faire vfact ← r.

\# 4 : Faire S ← reste(S) et aller en # 2.

\# 5 : L'évaluation de l'expression fact se termine et est déterminée, sa valeur est vfact.

La sémantique des évaluations des expressions term w1 fact, expr w2 term et -term est similaire à celle présentée au chapitre III, à ceci près que les opérations primitives (addition, multiplication, ...) sont, maintenant, les opérations primitives du type <u>entier2</u> correspondantes.

Exemples: Soient t1, t2 deux tableaux d'entiers.

facteurs: sum for i in [1..8] of (t1[i] * t2[i])
prod for i in [1..8] of (i)
max for i in {1..8} of(t1[i] + t2[i])
min for i in {1..8} of (max for j in {1..8} of (t1[j] + t2[i]))

b. Expressions de type caractère

Syntaxe

<exp2 caract simple> ::= max for <identificateur> in <exp2 de type ensemble d'élt de type ty> of (<exp2 de type caractère>) |
min for <identificateur> in <exp2 de type ensemble d'élt de type ty> of (<exp2 de type caractère>)

De plus, pour que l'expression max (*min*) for id in ... soit syntaxiquement correcte dans le contexte C, l'identificateur id ne peut être le nom d'une variable ou d'un tableau du contexte C.

<exp2 de type caractère> ::= <expression de désignation de type caractère> |
<exp2 caract simple> | '<caractère>' |
<appel de fonction de type caractère>

Sémantique

Soient C, un contexte et E(C), son état,
exp-simple, une exp caract simple,
i, un identificateur (qui n'est pas le nom d'une variable ou d'un tableau de C)
ens, une expression de type ensemble d'élt. de type ty et
expr, une expression de type caractère.

Si exp-simple est l'expression max (*min*) for i in ens of (expr), son évaluation dans E(C) se déroule de la façon suivante:

\# 1 : Evaluer l'expression ens dans E(C). Si cette évaluation est indéterminée, c'est aussi le cas de l'évaluation de exp-simple,

sinon, notons A, la valeur trouvée (de type ensemble d'élt de type ty) et vexp, une valeur de type caractère initialement '-' (';'). Notons encore C', le contexte construit à partir de C en lui ajoutant la variable i de type ty.

\# 2 : Si l'ensemble A est vide, aller en # 5, sinon, notons v, la valeur élt(A) de type ty. Affectons ensuite à la variable i, la valeur v.

\# 3 : Evaluer exp dans E(C'). Si cette évlauation est indéterminée, c'est aussi le cas de l'évaluation de exp-simple, sinon, notons r la valeur (de type caractère) obtenue. Si r >(<) vexp, faire vexp ← r.

\# 4 : Faire A ← A\{v} et aller en # 2.

\# 5 : L'évaluation de l'expression exp-simple se termine et est déterminée; sa valeur est vexp.

La sémantique de l'évaluation de expr si elle se résume à une expression de désignation, à un caractère 'c' ou à une expression du type (<exp2 de type caractère>) est identique à celle présentée au chapitre III.

Exemple: Soit t, un tableau de caractères de C.

expressions caract simple: max for i in {1..8} of (t[i])
min for i in {1..8} of (t[i])

c. Expressions booléennes

Syntaxe

<exp2 booléenne simple> ::= <expression de désignation de type booléen>| <valeur de vérité>|<appel de fonction de type booléen>| for all <identificateur> in <exp2 de type ensemble d'élt de type ty>: (<exp2 booléenne>)|There exists <identificateur> in <exp2 de type ensemble d'élt de type ty>: (<exp2 booléenne>)

De plus, pour que les expressions for all id in... et there exists id in... soient syntaxiquement correctes dans le contexte C, l'identificateur id ne peut être le nom d'une variable ou d'un tableau du contexte C.

```
<proposition2 atomique> ::= <exp2 booléenne simple>|
                            <expr arithmétique><opérateur relationnel>
                                        <exp2 arithmétique>|
                            <exp2 de type caractère><opérateur relationnel>
                                        <exp2 de type caractère>

<opérateur relationnel> ::= =|<=|>=|<>|>|<

<négation2 > ::= <proposition2 atomique>| not <proposition2 atomique>

<conjonction2> ::= <négation2> |<négation2> and <conjonction2>

<exp2 booléenne> ::= <conjonction2>|<conjonction2> or <exp2 booléenne>|
                            <conjonction2> ⇒ <exp2 booléenne>
```

Sémantique

Soient C, un contexte,

sexpr, une expression booléenne simple,

aexpr1, aexpr2, deux expressions arithmétiques ou deux de type caractère,

aprop, une proposition atomique,

nég, une négation,

conj, une conjonction,

bexpr, une expression booléenne,

w, un opérateur relationnel,

i, un identificateur (n'étant pas le nom d'une variable ou d'un tableau de C)

et ens une expression de type ensemble d'élt de type ty.

Voici comment se déroule, dans E(C), l'évaluation d'expressions booléennes de différentes formes.

Si sexpr se résume soit à une expression de désignation de type booléen, soit à une des deux valeurs de vérité, soit, encore, à l'expression (bexp), la sémantique de son évaluation est identique à celle du chapitre III.

Si sexpr est l'expression for all (*there exists*) i in ens: (bexp), son évaluation, dans E(C), se déroule comme suit.

\# 1 : Evaluation de ens dans E(C). Si celle-ci est indéterminée, c'est aussi le cas de l'expression sexpr, sinon, notons A, sa valeur (de type ensemble d'élt de type ty) et notons bool, une valeur booléenne initialement true (*false*). Notons encore C', le contexte construit à partir de C en lui ajoutant la variable i de type ty.

\# 2 : Si l'ensemble A est vide, aller en # 5, sinon, notons v, la valeur élt(A) de type ty. Affectons ensuite à la variable i, la valeur v.

\# 3 : Evaluer bexpr dans E(C'); si cette évaluation est indéterminée, c'est aussi le cas de l'évaluation de sexpr, sinon, notons b, la valeur de vérité obtenue. Si b est la valeur de vérité false (*true*), faire bool ← false (*true*).

\# 4 : Faire A ← A\{v}
aller en # 2.

\# 5 : L'évaluation de sexpr se termine et est déterminée; sa valeur est bool.

La sémantique de l'évaluation des expressions aexpr1 w aexpr2, not aprop, neg and conj, conj or bexp est identique à celle décrite au chapitre III.

L'évaluation de conj ⟹ bexpr, dans E(C), se déroule comme suit:

On évalue dans E(C) l'expression conj; si cette évaluation est indéterminée, c'est aussi le cas pour l'expression conj ⇒ bexpr, sinon, notons bool1, la valeur de vérité qu'elle renvoie. Si bool1 est la valeur de vérité faux, l'évaluation de conj ⇒ bexpr se termine (sans évaluer bexpr),

est déterminée et renvoie la valeur vrai. Dans le cas contraire, on évalue bexpr dans E(C); si cette évaluation est indéterminée, c'est aussi le cas de l'évaluation de conj $\Rightarrow$ bexpr, sinon notons bool2, la valeur de vérité qu'elle renvoie. L'évaluation de conj $\Rightarrow$ bexpr est déterminée et renvoie la valeur bool2.

Exemple: Soient x une variable entière et
t un tableau d'entiers d'indices [k..l].

Expressions booléennes simples:

for all i in [k..l]: $(t[i] \geqslant x)$

there exists i in [k..l]: $(t[i] > x)$

expression booléenne

$(x \geqslant k \text{ and } x \leqslant l) \Rightarrow t[x] \geqslant x$

d. Expressions de type ensemble d'entiers et de type suite d'entiers

Syntaxe

<liste d'entiers> ::= <exp2 arithmétique>|
<exp2 arithmétique>..<exp2 arithmétique>|
<expression de désignation de tableau>|
<expression de désignation de tableau>[<liste d'entiers>]|
<liste d'entiers>, <liste d'entiers>

<exp2 de type suite d'entiers> ::= <expression de désignation de type suite d'entiers>|
[<liste d'entiers>]|<appel de fonction de type suite d'entiers>|
[<identificateur> of <exp2 de type suite d'entiers>|<exp2 booléenne>]

<exp2 de type ensemble d'entiers> ::= <expression de désignation de type ensemble d'entiers>|
{<liste d'entiers>}|<appel de fonction de type ensemble d'entiers>|
{<identificateur> of <exp2 de type ensemble d'entiers>|<exp2 booléenne>}
|{<exp2 de type suite d'entiers>}

De plus, pour que les expressions [id of...|..] et {id of...|...} soient syntaxiquement correctes dans C, l'identificateur id ne peut être le nom d'une variable ou d'un tableau de C.

Sémantique

Soient C, un contexte et E(C) son état,
exp, exp1, exp2 des expressions arithmétiques,
tab, une expression de désignation de tableau de type entier,
i, un identificateur (qui n'est pas le nom d'une variable ou d'un tableau de C),
liste, liste1, liste2, des listes d'entiers,
suite, une expression de type suite d'entiers,
ens, une expression de type ensemble d'entiers,
bexp1, une expression booléenne,
dés-suite, dés-ens, des expressions de désignation respectivement de type suite d'entiers et d'ensemble d'entiers.

1. Evaluation des expression de type suite d'entiers

- Evaluation de [exp]→ Evaluation de l'expression arithmétique exp dans E(C); si elle est indéterminée, c'est aussi le cas de l'expression [exp], sinon notons v, sa valeur. Si v ∊ {zmin,zmax}, l'évaluation de [exp] est également indéterminée, sinon, sa valeur est la suite d'entiers constituée de l'unique élément v.

- Evaluation de [exp1..exp2] → Evaluation (dans un ordre quelconque) de exp1 et exp2 dans E(C). Si l'une d'entre elles est indéterminée, c'est aussi le cas de l'évaluation de [exp1..exp2], sinon, notons binf et bsup, les valeurs qu'elles renvoient. Si binf est la valeur zmin ou si bsup est la valeur zmax, c'est également indéterminé, sinon, la valeur de l'expression [exp2 ..exp2] est la suite des valeurs (entières) comprises entre binf et bsup, rangées par ordre croissant. Rem: Si binf > bsup, la valeur de l'expression est la suite vide.

- Evaluation de [tab[liste]]: Elle se déroule comme suit:

  On évalue l'expression de type suite [liste] dans E(C). Si cette évaluation est indéterminée, c'est aussi le cas de l'expression envisagée, sinon, notons $[v_1,...v_n]$ la suite qu'elle renvoie. Toutes les valeurs $v_1,...v_n$ doivent appartenir à l'ensemble des indices du tableau et toutes les variables $tab[v_i]$ doivent avoir été initialisées et donc avoir une valeur, notée w_i. Si ces conditions sont vérifiées, la suite $[w_1,...,w_n]$ est la valeur de l'expression [tab[liste]]. Dans l'alternative, l'évaluation de celle-ci est indéterminée.

- Evaluation de [tab]: Cas particulier du cas précédent où la liste liste correspond à la liste des indices rangés par ordre croissant. Dès lors, l'évaluation de [tab] (où tab désigne un tableau dont les indices sont [binf..bsup]) est identique à celle de [tab[binf..bsup]].

- Evaluation de [liste1,liste2]:

  Evaluation (dans un ordre quelconque) des expressions de type suite [liste2] et [liste 2] dans E(C). Si l'une d'entre elles est indéterminée, c'est aussi le cas de [liste1,liste2], sinon, notons S_1 et S_2, les valeurs qu'elles renvoient; concat (S_1,S_2) est la valeur de l'expression [liste1,liste2].

- Evaluation de dés-suite:

  Evaluation de l'expression de désignation dés-suite. Si celle-ci est déterminée et si la variable qu'elle désigne a été initialisée, alors la valeur de l'expression dés-suite est la valeur de la variable identifiée; dans le cas contraire, l'évaluation de celle-ci est indéterminée.

- Evaluation de [i <u>of</u> suite|bexp]: son évaluation dans le contexte E(C) se déroule comme suit:

  \# 1 : Evaluation de l'expression suite dans E(C). Si celle-ci est indéterminée, c'est aussi le cas de l'évaluation de [i <u>of</u> suite| bexp], sinon, notons S, la valeur renvoyée et R, une valeur de type <u>suite d'entiers</u> initialement vide. Notons également C', le contexte construit à partir de C en lui ajoutant la variable i de type <u>entier2</u>.

2 : Si la suite S est vide, aller en # 5, sinon, notons r, la valeur premier (S). Affectons ensuite à la variable i, la valeur r.

3 : Evaluer, dans E(C'), l'expression booléenne bexpr; si celle-ci est indéterminée, c'est aussi le cas de l'expression [i <u>of</u> suite| bexpr], sinon, notons bool, sa valeur. Si bool est la valeur de vérité vrai, faire R ← concat (R,[r]).

4 : Faire S ← reste(S)
Aller en # 2.

5 : L'évaluation de l'expreesion [i <u>of</u> suite|bexpr] se termine et est déterminée; sa valeur est R.

2. Evaluation des expressions de type <u>ensemble d'entiers</u>

L'évaluation des expressions de type <u>ensemble d'entiers</u> {exp}, {suite}, {exp1..exp2}, {tab}, {tab[liste]} ou de {liste1, liste2} est similaire à l'évaluation de l'expression de type <u>suite d'entiers</u> correspondante (resp. [exp], suite, [exp1..exp2], [tab], [tab[liste]] ou [liste1, liste2]): si celle-ci est indéterminée, c'est aussi le cas de l'expression de type ensemble, sinon, notons S la valeur de type suite qu'elle renvoie; la valeur de l'expression correspondante de type <u>ensemble d'entiers</u> est l'ensemble des valeurs (entières) de la suite S.

- Evaluation de dés-ens:
  Evaluation de l'expression de désignation dés-ens. Si celle-ci est déterminée et si la variable qu'elle désigne a été initialisée, alors la valeur de l'expression dés-ens est la valeur de la variable identifiée; dans le cas contraire, l'évaluation de celle-ci est indéterminée.

- Evaluation de {i <u>of</u> ens|bexp}: son évaluation dans E(C) se déroule comme suit:

1 : Evaluation de l'expression ens dans E(C). Si celle-ci est indéterminée, c'est aussi le cas de l'évaluation de {i <u>of</u> ens|bexpr}, sinon notons A, la valeur renvoyée et R, une valeur de type <u>ensemble d'entiers</u> initialement vide. Notons également C', le contexte construit à partir de C en lui ajoutant la variable i de type <u>entier2</u>.

2 : Si l'ensemble A est vide, aller en # 5, sinon, notons v, la valeur élt(A). Affectons ensuite à la variable i, la valeur v.

3 : Evaluer dans E(C') l'expression booléenne bexp; si celle-ci est indéterminée, c'est aussi le cas de l'expression {i <u>of</u> ens|bexp}, sinon, notons bool, la valeur qu'elle renvoie. Si bool est la valeur de vérité vrai, faire R ← R ∪ {v}.

4 : Faire A ← A\{v},
aller en # 2.

5 : L'évaluation de l'expression {i <u>of</u> ens|bexpr} se termine et est déterminée, sa valeur est R.

<u>Exemples</u> : Soient x et t une variable et un tableau d'entiers, suite et ens des variables de type <u>suite</u> et <u>ensemble d'entiers</u>.

Expressions de type <u>suite d'entiers</u>:

```
[3 x + 2],
[x..x + 10],
[3 x + 2, 4 x + 2, 5 x + 2, t[1..5]],
 suite
[k of suite |t[k] ⩾ 0]
[k of[x..x+10] |t[k] ⩾ 0]
[t]
```

Expressions de type <u>ensemble d'entiers</u>:

```
{3x + 2}
{x..x+10}
{[x..x+10]}
{3x+2, 4x+2, 5x+2, t[1..5]}
 ens
{k of ens | t[k] ⩾ 0}
{k of {x..x+10}.| t[k] ⩾ 0}
{t}
```

e. Expressions de type ensemble de caractères ou de type suite de caractères

Syntaxe

<liste de caract> ::= <exp2 caractère>|
<exp2 caractère>..<exp2 caractère>|
<expression de désignation de tableau>|
<expression de désignation de tableau>[<liste d'entiers>]|
<liste de caract>,<liste de caract>

<exp2 de type suite de caractères> ::= <exp de désignation de type suite de caractères>
|[<liste de caract>]|<appel de fonction de type suite de caractères>|
[<identificateur> of <exp2 de type suite de caractères>|<exp2 booléenne>]

<exp2 de type ensemble de caractères> ::= <exp de désignation de type ensemble de caractères>|
|{<liste de caract>}|<appel de fonction de type ensemble de caractères>|
{<identificateur> of <expr de type ensemble de caractères>|<exp2 booléenne>}
|{<exp2 de type suite de caractères>}

De plus, pour que les expressions {id of ... |...} et [id of ...|...] soient syntaxiquement correctes dans C, l'identificateur id ne peut être le nom d'une variable ou d'un tableau du contexte C.

Sémantique

L'évaluation des expressions de type ensemble ou suite de caractères est fort similaire (en fait, plus simple !) à celle des expressions de type ensemble ou suite d'entiers. En fait, la sémantique de l'évaluation des premières revient à celle des expressions de type ensemble ou suite d'entiers en

a. substituant systématiquement le type entier par le type caractère ainsi que les opérations primitives associées et en

b. supprimant les tests consistant à savoir si la valeur obtenue est zmin ou zmax.

4.4. DECLARATION DE VARIABLES ET TABLEAUX

La syntaxe et la sémantique des déclarations des variables et des tableaux est fort similaire à celles présentées pour le langage L1. La seule modification concerne les types. En effet, l'ensemble des types permis pour les variables et tableaux déclarés est plus vaste: une variable peut être déclarée de type v-type2 et un tableau, de type t-type2 où

<t-type2> ::= integer (☆) |boolean|char

<v-type2> ::= integer (☆) |boolean|char|set of integer (☆☆) |set of char| set of bool|list of integer (☆☆) |list of bool|list of char

Exemples: var i,j,k : integer;
ens: set of integer;
suit1, suit2: list of bool;
t : array [1..20] of char;

*

* *

(☆) Integer $\equiv$ Z $\cup$ {zmin,zmax}
(☆☆) Integer $\equiv$ Z (sans zmin ni zmax)

4.5. INSTRUCTIONS (sauf les appels de procédures)

Une instruction du langage L2 est une construction qui, étant donné un contexte C et son état, spécifie une suite d'opérations à effectuer (susceptibles de modifier E(C)) sans points de contrôle (assertions dans L1).

Syntaxe

<instruction2> ::= <instr2 d'affectation>|
<instr2 conditionnelle>|
<instr2 répétitive>|
<instr2 de branchement>|
<instr2 de lecture>|
<instr2 d'écriture>|
<appel de procédure2>|
<instr2 composée>

<instr2 d'affectation> ::= <expression de désignation> ::= <expression2>
où le type de l'expression de désignation est le même que le type de l'expression du membre de droite.

<instr2 conditionnelle> ::= if <exp2 booléenne> then <instruction2>| (1ère forme)
if <expr booléenne> then <instruction2> else <instruction2> (2ème forme)

<instr2 répétitive> ::= while <exp2 booléenne> do <instruction2>

<instr2 de lecture> ::= read (<expression de désignation>)
où l'expression de désignation doit être de type caractère
(procédure prédéfinie pour la lecture d'entiers: cfr chapitre III)

<instr2 d'écriture> ::= write (<exp2 de type caractère>)|
write (<exp2 arithmétique>)

Sémantique

La sémantique de l'exécution de ces instructions revient à celle décrite au paragraphe 3.5 relative aux instructions du langage L1 correspondantes. Une précision concernant l'exécution des instructions d'écriture (write (<exp2 arithmétique>)) et d'affectation (<expression de désignation> := <exp2 arithmétique>) est cependant nécessaire. Si, lors de l'exécution de celles-ci, l'évaluation de l'expression arithmétique renvoie la valeur zmin ou zmax, l'exécution de l'instruction en question est indéterminée.

Il existe, par contre, des différences plus importantes en ce qui concerne les instructions composées dont voici la syntaxe.

Syntaxe

<instr2 composée> ::= begin <liste d'instructions2> end|
begin:< identificateur><liste d'instructions2> end

De plus, pour que l'instuction composée begin : id... end soit syntaxiquement correcte dans le contexte C, l'identificateur id ne peut être le nom d'une étiquette de C.

<liste d'instructions2> ::= <instruction2>|
<liste d'instructions2>; <instruction2>

Sémantique

Soient C, un contexte et ins1, ... insn des isntructions ($n \geqslant 1$). L'exécution de l'instruciton begin ins1; ...; insn end, dans le contexte C, consiste à exécuter, successivement, chacune des instructions, ins1,.., insn.

Soit, de plus, x, un identificateur n'étant pas le nom d'une étiquette du contexte C. L'exécution de l'instruction begin: x ins1; ... insn end dans le contexte C se déroule comme suit:

\# 1 : Ajouter au contexte C l'étiquette de nom x, associée à l'instruction composée concernée.

\# 2 : Exécuter dans le contexte C l'instruction composée <u>begin</u> insl; ... insn <u>end</u> (lire la remarque ci-dessous !).

\# 3 : Si celle-ci se termine et est déterminée, l'exécution de l'instruction considérée est également déterminée et se termine en supprimant du contexte C l'étiquette de nom x.

Remarque importante: La sémantique de l'exécution d'une instruction composée est présentée ci-dessus comme étant la séquence des exécutions de toutes les instructions insl, ... insn qui la composent. Mais ce n'est pas toujours le cas ! En effet, cette séquence peut être interrompue à cause de l'exécution de l'une de ces instructions: il s'agit des instructions de branchement. La syntaxe et la sémantique de ces dernières sont tout à fait similaires à celles présentées au chapitre III.

*

* *

4.6. INSTRUCTION L1 / ASSERTION L2 : INTERFACE "SEMANTIQUE" (suite)

Comme l'indique le chapitre III, la place d'une assertion est définie à l'intérieur d'une instruction composée du langage L1. Une assertion doit pouvoir utiliser toutes les variables et tous les tableaux auxquels cette instruction peut faire référence (i.e. les variables et tableaux de son contexte). Le contexte d'une assertion doit dès lors contenir les variables et tableaux du contexte de l'instruction composée, noté C (☆). Remarquons qu'il ne se réduit pas à cela: une seconde partie de ce contexte comprend notamment les fonctions ou les procédures prédéfinies ou déclarées en L2 ou, encore, les variables ou tableaux "déclarés" dans les assertions (cfr parag. 4.9). Pour faciliter la suite de l'exposé, il est intéressant de distinguer les deux parties d'un contexte d'une assertion. C'est pourquoi, on le notera, $C_i \cup C_a$, où C_i est constitué des variables et des tableaux du contexte C et des fichiers d'entrée et de sortie de ce même contexte et où C_a est la seconde partie indiquée ci-dessus.
<u>Remarque</u>: les noms des variables et tableaux de C_a doivent être distincts des noms de ceux de C_i.

*

* *

(☆) En fait, ce n'est pas tout à fait les variables et les tableaux du contexte de l'instruction composée car le type entier est légèrement modifié dans le langage L2 par l'ajout des valeurs zmin et zmax. Mais, remarquons que le passage des entiers L1 en entiers L2 est trivial (l'inverse n'étant pas vrai).

4.7. DECLARATIONS ET APPELS DE FONCTIONS OU DE PROCEDURES

a. Déclarations de fonctions et de procédures

Syntaxe

La syntaxe n'est pas présentée exhaustivement. En effet, elle est fort semblable à celle du langage L1. C'est pourquoi on se contentera ici de donner les quelques petites différences:

1. Dans le langage L1, l'en-tête d'une fonction ou procédure est notamment composée d'une précondition; ce n'est pas le cas dans le langage L2.

2. Le type des arguments et des paramètres formels, en L1, est restreint aux entiers (entier1), aux booléens et aux caractères; dans le langage L2, sont permis les entiers (entier2), les booléens, les caractères, les suites et les ensembles d'entiers, de caractères ou de booléens, à l'exception des tableaux pour lesquels les types suite et ensemble ne sont pas permis.

3. <corps2> ::= <instr2 composée>

De plus, pour qu'une déclaration de fonction ou de procédure soit syntaxiquement correcte dans le contexte $C_i \cup C_a$, il faut que son nom ne soit pas encore le nom d'une fonction ou d'une procédure de C_a.

Sémantique

L'analyse d'une déclaration de fonction ou de procédure L2 entraîne l'ajout de celle-ci au contexte de toutes les assertions du programme (dans la partie C_a).

b. Appel de fonctions ou de procédures

Syntaxe

Identique à celle présentée au chapitre III.

Sémantique

Semblable à celle présentée au chapitre III, à ceci près:

1. pas d'évaluation de précondition (car pas de précondition !);
2. le contexte noté C' au chapitre III est initialement constitué des fonctions et procédures L2 prédéfinies ou déclarées dans le programme avec assertions.

*

*   *

4.8. FONCTIONS PREDEFINIES

Notations: ty est un artifice de notation pour symboliser le type entier, booléen ou caractère et TY pour symboliser, respectivement, Z, B ou C.

a. Fonctions prédéfinies à arguments de type suite

1. empty : S(TY) ⟶ B
   S ⇝ empty(S)

   où empty(S) renvoie la valeur de vérité vrai si la suite S est vide et faux sinon.

   Ex: empty ([10..1]) = true
   empty ([1..10]) = false

2. long : S(TY) ⟶ Z
   S ⇝ long(S)

   où long(S) donne la longueur de la suite S.

   Ex: long ([1,2,1]) = 3
   long ([1..10]) = 10
   long ([10..1]) = 0 (suite vide)

3. first (*last*): S(TY) ⟶ TY
   S ⇝ first (*last*)(S)
   précondition: S non vide

   où first (*last*), appliquée à une suite non vide S, renvoie son premier (*dernier*) élément.

   Ex: first ([3,2,1]) = 3
   last ([3,2,1]) = 1

4. head (*tail*): S(TY) ⟶ S(TY)
   S ⇝ head (*tail*)(S)
   précondition: S non vide

   où head (*tail*) appliquée à une suite non vide, renvoie la suite préfixe (*suffixe*) de S de longueur (long (S)-1), c'est-à-dire la suite S sans son dernier (*premier*) élément.

Ex: head ([3,2,1]) = [3,2]
    tail ([3,2,1]) = [2,1]

5. <u>concat</u> : S(TY) x S(TY) ⟶ S(TY)

   (S_1,S_2) ⟿ concat (S_1,S_2)

   où concat(S_1,S_2) renvoie la concaténation de S_1 et S_2.

Ex: concat ([1,2,1], [1,3]) = [1,2,1,1,3]
    concat ([10..1], [2,2,3]) = [2,2,3]

6. <u>egal</u> : S(TY) x S(TY) ⟶ B

   (S_1,S_2) ⟿ egal (S_1,S_2)

   où egal (S_1,S_2) est vrai ssi les suites S_1 et S_2 sont égales.

Ex: egal ([1,2,1], [1,1,2]) = <u>false</u>
    egal ([1,2,3], [1..3]) = <u>true</u>

7. <u>permut</u> : S(TY) x S(TY) ⟶ B

   (S_1,S_2) ⟿ permut (S_1,S_2)

   où permut (S_1,S_2) est vrai ssi S_2 est une permutation de la suite S_1. Une suite est une permutation d'une autre ssi toutes les valeurs se trouvant une ou plusieurs fois dans l'une, figurent autant de fois dans l'autre et vice versa.

Ex: permut ([1,1,2], [1,2,1]) = <u>true</u>
    permut ([1,1,2], [2,2,1]) = <u>false</u>

8. <u>ssuite</u> (<u>*segment*</u>): S(TY) x S(TY) ⟶ B

   (S_1,S_2) ⟿ ssuite (*segment*) (S_1,S_2)

   où ssuite (*segment*) (S_1,S_2) est vrai ssi la suite S_1 est une sous-suite (*un segment*) de S_2.

Ex: ssuite ([1,1,3], [4,5,1,3,1,3]) = <u>true</u>
    ssuite ([1,2,1], [1,1,2,4]) = <u>false</u>
    segment ([1,1], [2,1,1,3]) = <u>true</u>
    segment ([1,1], [1,2,1,3]) = <u>false</u>

9. préfixe (*suffixe*) : S(TY) x S(TY) ⟶ B

(S_1,S_2) ⇝ préfixe (*suffixe*) (S_1,S_2)

où préfixe (*suffixe*) (S_1,S_2) est vrai ssi S_1 est un préfixe (*suffixe*) de S_2.

Ex: préfixe ([1,2,2], [1,2,2,3]) = true
préfixe ([1,2,3], [1,2,2,3]) = false
suffixe ([1,2],[1,1,2]) = true
suffixe([1,2], [1,2,1]) = false

10. tricrois (*tridec*) : S(TY) ⟶ B

S ⇝ tricrois (*tridec*) (S)

où tricrois (*tridec*) (S) est vrai ssi S est une suite triée par ordre croissant (*décroissant*).

Ex: tricrois ([1,2,2,5]) = true
tricrois ([1,2,1,5]) = false
tridec ([5,2,1,1]) = true
tridec ([5,2,3,1]) = false

11. tristrcrois et tristrdec: idem, mais pour les ordres stricts.

Ex: tristrcrois ([1,2,2,5]) = false
tristrcrois ([1,2,5]) = true

b. Fonctions prédéfinies à arguments de type ensembles

1. empty : E(TY) ⟶ B

A ⇝ empty(A)

où empty(A) est vrai ssi l'ensemble A est vide.

Ex: empty ({10..1}) = true

2. card : E(TY) ⟶ Z

A ⇝ card(A)

où card(A) renvoie le cardinal de l'ensemble A.

Ex: card ({1,1,2,1}) = 2
card ({1..10}) = 10

3. élt : E(TY) ⟶ TY

A ⇝ élt(A)

précondition: A non vide

où élt appliqué à un ensemble A non vide renvoie un élément quelconque de l'ensemble A.

Ex: élt ({1}) = 1

élt ({1,2}) = 1 ou 2

4. union (*inter*) : E(TY) X E(TY) ⟶ E(TY)

(A,B) ⇝ union (*inter*) (A,B)

où union (*inter*) (A,B) est l'ensemble résultant de l'union (*intersection*) ensembliste de A et B, i.e. A ∪ (∩) B.

Ex: union ({1,2},{1,3}) = {1,2,3}

inter ({1,2},{1,3}) = {1}

5. differ : E(TY) X E(TY) ⟶ E(TY)

(A,B) ⇝ differ (A,B)

où differ (A,B) renvoie l'ensemble résultant de la différence ensembliste de A par B, i.e. A/B.

Ex: differ ({1,2,1},{1}) = {2}

6. egal : E(TY) X E(TY) ⟶ B

(A,B) ⇝ egal (A,B)

où egal (A,B) est vrai ssi les deux ensembles A et B sont égaux (égalité ensembliste)

Ex: egal ({1,2,1},{2,1}) = true

egal ({1,2,3},{1,2}) = false

7. inclus : E(TY) X E(TY) ⟶ B

(A,B) ⇝ inclus (A,B)

où inclus (A,B) est vrai ssi A est inclus dans B.

Ex: inclus ({1,1,2},{3,2,1}) = true

8. max (*min*) : E(TY) ⟶ TY (Rem: TY ≠ B)

A ⇝ max (*min*) (A)

où max (*min*) (A) renvoie la même valeur que l'expression max (*min*) for i in A of (i).

Ex: max ({1,2,5,3}) = 5

min ({0..10}) = 0

4.9. MEMORISATION DES VALEURS INTERMEDIAIRES

Rappelons-nous qu'une assertion est une affirmation au sujet de l'état actuel des variables. On admettra facilement que, pour être utile, une assertion doit généralement faire référence à des valeurs qu' ont eues certaines variables ou tableaux (☆). Il s'agit, dès lors, de mémoriser ces valeurs en les identifiant. Remarquons qu'une technique simple serait de laisser le soin à l'utilisateur de déclarer suffisamment de variables dans le programme (au moyen du langage L1) et de les utiliser en leur affectant différentes valeurs à différents endroits du programme. Mais si cette technique a l'avantage d'être simple, elle a de gros désavantages: d'abord, d'un point de vue "logique" et "esthétique", il serait fort malsain de mélanger ce qui est de l'ordre du calcul et ce qui est de l'ordre du raisonnement. Ensuite,ces variables et les instructions consistant à leur affecter des valeurs, sont tout à fait superflues lorsque l'on exécute le programme sans vérification d'assertions; ces variables seront pourtant créées et ces instructions exécutées. Il est donc intéressant de permettre une "pseudo-instruction" interne aux assertions qui n'aurait aucune conséquence néfaste du point de vue "temps calcul" ou "place mémoire" lorsque l'utilisateur choisit l'exécution sans vérification d'assertions.

Syntaxe

<pseudo-instruction de mémorisation> ::= let <identificateur>be<expression2>

De plus, notons $C = C_i \cup C_a$, le contexte de l'assertion dans laquelle se trouve la pseudo-instruction let id be exp. Pour que celle-ci soit syntaxiquement correcte dans $C_i \cup C_a$, l'identificateur id doit être le nom d'une variable appartenant à la partie C_a du contexte et de même type que l'expression exp (et, par la même occasion, il ne peut pas être le nom d'une variable ou d'un tableau de C_i).

(☆) Remarquons d'ailleurs l'emploi fréquent des notations du type i_o, A_o,...

<u>Sémantique</u> (uniquement si exécution avec vérification d'assertions)

a. Si, dans une ou plusieurs assertions du corps du programme, d'une procédure ou d'une fonction, apparaissent une ou plusieurs pseudo-instructions <u>let</u> id <u>be</u> exp (où id est un identificateur fixé), cela correspond à la "déclaration" d'une variable de nom id et de type fixé suivant le type de l'expression exp (type qui doit être le même pour toutes les expressions d'une pseudo-instruction portant sur cette nouvelle variable). Par "déclaration", on entend, ajout de cette variable dans le contexte (dans la partie C_a) de toutes les assertions de l'instruction composée L1 qui est le corps du programme, de la procédure ou de la fonction en question.

b. Soient $C = C_i \cup C_a$, le contexte de la pseudo-instruction décrite ci-dessous, vty, un identificateur d'une variable de type ty de C_a et expty, une expression de type ty. L'exécution de la pseudo-instruction <u>let</u> vty <u>be</u> expty dans C s'effectue comme suit:

1 : Evaluation de expty dans le contexte C; si elle est indéterminée, c'est aussi le cas de l'exécution de la pseudo-instruction en question; sinon, notons valty sa valeur.

2 : Affectation de la valeur valty à la variable vty.

<u>Exemple</u>: corps d'un programme simple (a et b, deux variables entières de C).

<u>begin</u>

{* let a_o be a; let b_o be b *}

a := a + b;

{* a = a_o + b_o <u>and</u> b = b_o *}

b := a - b;

a := a - b

{* a = b_o <u>and</u> b = a_o *}

<u>end</u>.

*

*   *

4.10. "FONCTIONS" PREDEFINIES SUPPLEMENTAIRES (*)

1. Fonction testant la non-initialisation

Syntaxe

<fonction de détermination> ::= déf (<liste d'expressions de désignation>)

<liste d'expressions de désignation> ::= <expression de désignation de variable>|

<expression de désignation de tableau>

<expression de désignation de tableau>[<exp2 arithmétique>..<exp2 arithmétique>]|

<liste d'expressions de désignation>,<liste d'expressions de désignation>

De plus, déf (...) ne peut figurer que dans des énoncés d'assertions.

Sémantique

Soient C, un contexte et

expdés1, ..., expdésn une liste d'expressions de désignation

déf (expdés1, ..., expdésn) est considérée comme une fonction L2 booléenne; Son évaluation dans E(C) revient à l'évaluation de déf (expdés1) and déf (expdés2)... and déf (expdésn).

Soient encore,

expdésvar, une expression de désignation de variables,

expdéstab, une expression de désignation de tableau et

exp1, exp2, deux expressions arithmétiques.

- Evaluation de déf (expdésvar): on évalue d'abord l'expression de désignation expdésvar. Si elle est indéterminée, c'est aussi le cas de déf expdésvar) sinon, notons var la variable qu'elle désigne. L'évaluation de déf (expdésvar) renvoie la valeur vrai si var a été initialisée et faux sinon.

(*) Les guillemets indiquent le statut et le caractère tout spécial de ces fonctions. En effet, on remarque que leurs syntaxes et leurs sémantiques diffèrent fort de celles des fonctions L2 présentées précédemment.

- Evaluation de déf (expdéstab): On évalue d'abord l'expression de désignation de tableau expdéstab. Soit t le tableau désigné. L'évaluation de déf (expdéstab) renvoie la valeur vrai si tous les éléments du tableau ont été initialisés et faux sinon.

- Evaluation de déf (expdéstab [exp1..exp2]): se déroule de la même façon que for all i in {exp1..exp2} : (déf(expdéstab[i])).

Exemples: Soient x et t, une variable et un tableau d'entiers.

"Fonction" déf : déf (x)
déf (t)
déf (t,x)
déf (x,t[i..5], t[7..10])

2. Fonction concernant la terminaison

Syntaxe

<pseudo-instruction d'initialisation>::= variant (<identificateur>,"<exp2 arithmétique>",<exp2 arithmétique>)

<fonction de terminaison>::= term (<identificateur>)

De plus, variant (...,...,...) et term (...) ne peuvent figurer que dans des énoncés d'assertion et si term(id) apparaît dans le corps du programme, d'une procédure ou d'une fonction L1, variant(id,...,...) doit y apparaître une et une seule fois.

Sémantique

a. Si, dans une assertion du corps du programme, d'une procédure ou d'une fonction L1 apparaît la pseudo-instruction variant(id,...,...), cela correspond à la déclaration de deux variables id* et id< (☆).

(☆) Les noms choisis pour les variables sont des artifices de notation pour qu'elles ne soient accessibles que par l'intermédiaire de variant ou ou de term.

Par "déclaration", on entend ajout de cette variable au contexte (partie C_a) de toutes les assertions de l'instruction composée Ll qui est le corps du programme, de la procédure ou de la fonction Ll concernée.

b. variant: Soient C, le contexte de la pseudo-instruction décrite ci-dessous et expl, exp2, deux expressions arithmétiques; l'exécution, dans E(C), de la pseudo-instruction variant (id, "expl", exp2) est déterminée ssi l'évaluation de exp2, dans E(C), l'est et renvoie une valeur, v, strictement positive. Dans ce cas, elle a pour effet d'affecter à la variable id*, la valeur zmax et à la variable id<, la valeur v.

term: Soit variant (id, "expl", exp2), la pseudo-instruction associée à term (id); pour que l'évaluation de term (id), dans E(C) soit déterminée, il faut que

a. on ait déjà exécuté cette pseudo-instruction au moins une fois (i.e. id* et id< doivent être initialisées) et que
b. l'évaluation de expl dans E(C) soit déterminée et renvoie une valeur positive ou nulle, notée vid.

Si ces deux conditions sont réunies, l'évaluation de term(id) est déterminée et renvoie la valeur de vérité vrai si la valeur de l'expression expl a diminué d'au moins id depuis sa dernière évaluation par l'intermédiaire de term ou si l'on vient d'exécuter variant (id, "expl",exp2). On peut résumer cela en disant que l'évaluation de term(id) renvoie la même valeur que l'expression booléenne id* ⩾ vid + id<. Elle se termine, ensuite, en mettant à jour id : on affecte à celle-ci la valeur vid.

Rem.: Le fait de donner la valeur zmax à id* lors de l'exécution de variant (id, "expl", exp2) permet de ne pas distinguer le cas où l'évaluation de term(id) suit une exécution de variant (id, "expl", exp2) sans autre évaluation de term(id) entre, du cas contraire.

Exemples: Voir les programmes présentés comme exemples à la fin de ce rapport.

4.11. LES ENONCES D'ASSERTIONS ET DE PRECONDITIONS

a. Enoncés d'assertion

Rappel: Une assertion se situe dans une instruction composée du langage L1 (cfr paragraphe 3.6) et est de la forme:
{ * <énoncé d'assertion> * }

Syntaxe

<pseudo-instruction> ::= <pseudo-instruction de mémorisation>|<pseudo-instruction d'initialisation>

<énoncé d'assertion> ::= <vide>|<exp2 booléenne>|
<pseudo-instruction>;<énoncé d'assertion>

Sémantique

a. L'évaluation d'un énoncé d'assertion vide renvoie toujours la valeur de vérité vraie quel que soit le contexte C.

b. Soit C, un contexte: l'évaluation dans C d'un énoncé d'assertion qui se résume en une expression booléenne se déroule de la même manière que l'évaluation dans C de cette dernière.

c. Soient ass, un énoncé d'assertion et
ps-inst, une pseudo-instruction.
L'évaluation de l'énoncé d'assertion ps-instr; ass, dans C, se déroule comme suit:

"Exécution", dans C, de ps-instr; si elle est indéterminée, c'est aussi le cas de l'évaluation de l'énoncé d'assertion concerné. Dans l'alternative, on "évalue", dans C, l'énoncé d'assertion ass.

b. Enoncés de précondition

Rappel : une fonction ou une procédure du langage L1 peut avoir dans sa déclaration une précondition. Cette dernière est évaluée à l'appel s'il s'agit d'une exécution avec vérification d'assertions.

Syntaxe

<énoncé de précondition> ::= <exp2 booléenne>.

Sémantique

Revient à l'évaluation de l'expression booléenne qui la constitue.

*

*   *

Chapitre V

DEUX EXEMPLES EN GUISE DE CONCLUSION

5.1. CHOIX ET PRESENTATION DES EXEMPLES

Nous avons choisi deux exemples faisant intervenir un maximum de concepts du langage L2. De plus, ces exemples se veulent représentatifs de la classe des problèmes envisagés. Il s'agit du problème de la factorielle et du tri d'un vecteur d'entiers.

Comme ces deux problèmes ont déjà été envisagés dans la première partie de ce mémoire, nous ne présenterons ici que leurs spécifications et les programmes traduisant, dans notre formalisme, leurs organigrammes et leurs assertions présentés dans cette première partie (☆). Il y a bien sûr plusieurs façons de traduire un organigramme dans le langage L1: nous avons choisi la plus directe en utilisant, par exemple, les instructions de branchement pour traduire les boucles. Une autre façon de faire, moins immédiate, serait d'employer les instructions répétitives mais cela nécessiterait le dédoublement des invariants.

(☆) Une démonstration de la correction de ces problèmes est envisagée au cours "Séminaire de programmation".

5.2. LA FACTORIELLE

a. Spécification

La fonction fact1 est une fonction qui, étant donné une valeur entière positive v, renvoie la valeur v! (correspond au problème présenté aux paragraphes 2.6 et 2.7 de la première partie de ce mémoire).

b. Programme

Ce problème est programmé sous la forme d'une fonction L1 appelée fact1. La fonction fact2 utilisée dans ses assertions est une fonction L2 traduisant également la factorielle.

```
function1 fact1 (x: integer) : integer;
   var y,z : integer;
   begin
       {* déf(x)*}
       {* let xo be x ; x ≥ 0 *}
        z:= 1;
        y:= 0;
       {* variant (diff, "x-y", 1);*}
        begin : étiq
            { * x = xo and tricrois (1,y,x) and z = fact2(y) *}
            { * term(diff) *}
            if (x=y) then exit étiq ;
            y:= y+1;
            z:= z*y;
            goto étiq
        end;
       {* x = xo and z = fact2(x) *}
        fact1:= z
   end;
```

où fact2 est une fonction L2 définie de la façon suivante:

```
function2 fact2 (x : integer) : integer;
   begin
     fact2:= prod for i in [1..x] of (i)
   end;
```

5.3. TRI D'UN VECTEUR

a. Spécification

Etant donné un vecteur d'entiers composé de dix éléments, lire sur le fichier d'entrée les valeurs de ses éléments, les permuter ensuite de sorte que le vecteur soit trié par ordre croissant et enfin, imprimer le vecteur résultant sur le fichier de sortie.

b. Programme

Le programme présenté dans ce paragraphe utilise les quatre procédures suivantes:

1. Lecture: procédure concernant la lecture du vecteur ayant pour seul paramètre un tableau d'entiers passé par adresse.
2. Ecriture: procédure concernant l'écriture du vecteur ayant pour seul paramètre un tableau d'entiers passé par adresse.
3. Tri: procédure ayant deux paramètres de type entier passés par adresse; son exécution a pour effet de permuter les valeurs de ces paramètres de sorte que la valeur du premier d'entre eux soit plus petite ou égale à celle du second.
4. Auxiliaire: procédure qui a deux paramètres: un premier k, de type entier, passé par adresse et un second, a[1..10], un tableau d'entiers également passé par adresse. Sa spécification est la suivante: étant donné une variable entière k comprise entre 1 et 10 et un tableau d'entiers a[1..10], l'exécution de la procédure auxiliaire consiste à effectuer une permutation du vecteur a de sorte que a[k] soit le maximum des valeurs de a[1..k] sans modifier ni k ni a[k+1..10].

Seule la déclaration de cette dernière procédure sera explicitée dans ce programme.

```
program trivect ;
   var k : integer;
       a : array [1..10] of integer ;
   procedurel lecture (var a : array [1..10] of integer); ... (☆)
   procedurel ecriture (var a : array [1..10] of integer); ...(☆)
   procedure l  tri (var x,y : integer); ... (☆)
   procedure l  auxiliaire (var k : integer; var a : array [1..10] of integer);
       var l : integer;
       begin
           {* def (a,k)*}
           {* let a1 be[a]; let k1 be k; tricrois ([1,k,10])*}
           l:= 1;
           {* variant (variantl, "k-l",1);*}
           begin : etiql
                 { * tricrois (1,l,k) and permut ([a],a1) and k=k1 and
                     suffixe ([a[l+1..10]],a1) and a[l] = max ({a[1..l]})* }
                 { * term (variantl) *}
                   if (k=l) then exit etiql;
                   tri (a[l],a[l+1]);
                   goto etiql
             end
             {*permut ([a],a1) and suffixe ([a[k+1..10]], a1) and
               k = k1 and a[k] = max ({a[1..k]})* }
       end;
```

(☆) déclarations non explicitées ici.

```
begin
   lecture (a);
   {* def (a) *}
   {* let a_o be [a];*}
   k:= n;
   {* variant (variantØ, "k",1); *}
   begin: etiqØ
      {* tricrois ([0,k,10]) and permut ([a],a_o) and tricrois ([a[k+1..10]])
        and for all i in {i..k}:(for all j in {k+1..n}: (a[i] ≤ a[j]))*}
      {* term (variantØ)*}
       if (k=0) then exit etiqØ;
       auxiliaire (k,a);
       k:= k+1;
       goto etiqØ
   end;
   {*permut ([a];a_o) and tricrois ([a])*}
   ecriture (a)
end.
```

RÉFÉRENCE

[1] B. Le Charlier, "Définition du langage LSD / 80",
Institut d'Informatique, septembre 1980.