MASTER EN SCIENCES INFORMATIQUES

Contribution à l'implémentation d'un langage graphique de description d'assertions

Rulkin, Jeannine

Award date:
1989

Awarding institution:
Universite de Namur

Contribution à l'implémentation d'un langage graphique de description d'assertions

Jeannine Rulkin

Mémoire réalisé sous la
direction du Professeur B. Le Charlier
en vue de l'obtention du titre de
Licencié et Maître en Informatique.

Namur Juin 1989.

Bref résumé de ce mémoire

La nécessité d'arriver à une meilleure qualité des programmes est actuellement reconnue, et des méthodes de construction de programme sont développées à cet effet. Ainsi, aux Facultés Universitaires de Namur, une telle méthode est enseignée : la méthode de l'invariant ([PROG] et [SYS88]). Ce mémoire contribue à l'élaboration d'un système d'aide à l'enseignement de la programmation basé sur cette méthode, en cours de réalisation à l'institut d'informatique. La construction d'un programme est basée sur la description des situations de ce programme. Une situation est l'expression de conditions sur l'état des variables et tableaux du programme à un moment déterminé de son exécution. Ce mémoire définit un langage pour exprimer ces conditions. Ce langage doit être complété par un support graphique permettant la description de l'état des différents tableaux du programme. Les caractéristiques de ce langage sont illustrées par quelques exemples. Ils montrent la puissance du langage associé à un support graphique : grâce à la représentation graphique, l'expression des conditions est plus intuitive, et même quelquefois implicite à la représentation. Ensuite, un scénario d'exploitation du système, offrant la possibilité de décrire la situation à l'aide du langage de conditions combiné à une representation graphique est défini. La définition de ce scénario permet de dégager une architecture du système basée sur les concepts définis dans le scénario. Enfin, ce mémoire réalise l'implémentation d'un concept central pour la mise en oeuvre du système : le concept de "cadre". Ce concept correspond à l'implémentation d'un éditeur pour la saisie des conditions et des diverses informations nécessaires au cours du déroulement du scénario défini.

Abstract

The correctness of program construction is currently recognized as a major issue, and appropriate methodologies are therefore developped. Such a methodology, called the "Invariant Method" ([PROG] et [SYS88]), is teached at the Institut d'Informatique of the Facultés Universitaires de Namur. This thesis contributes to the construction of an interactive computer system, currently realized at Namur, and designed to help the students to learn the methodology. The process of program construction is based on a description of the situations of this program. A situation expresses conditions on the program's variables and arrays state, at a given stage of its execution. This thesis defines a language allowing the expression of these conditions. This language is intended to be used with a graphical support. This graphical representation leads to a more intuitive representation of the conditions, which are sometimes implicit to the representation. The thesis also defines a script describing the execution of the system, combining the use of the language with graphical representations to express conditions. It also defines an architecture for the system, based on the concepts presented in the script. Finally, a central part of this architecture is implemented, namely the notion of "cadre", which corresponds to a text editor, allowing the input of conditions and other informations during the system's execution.

Remerciements

Que toutes les personnes m'ayant aidée au cours de cette dernière année d'études trouvent ici l'expression de mes remerciements.

Je remercie particulièrement Jean-Pierre Hogne pour sa lecture attentive du texte, et Baudouin Le Charlier, promoteur de ce mémoire, pour ses explications et ses conseils tout au long de la réalisation de ce travail.

Table des matières

Chapitre 1

Introduction

1. Introduction

La compréhension de ce mémoire sera facilitée par la lecture de l'article "Un système d'aide à l'enseignement d'une méthode de programmation" rédigé par Marc Derroitte et Baudouin Le Charlier. Cet article se trouve dans l'annexe 1 de ce mémoire.

Je prierai donc les lecteurs de bien vouloir prendre connaissance de ce document.

Rappelons brièvement le contenu de cet article et exposons la modeste contribution de ce mémoire aux objectifs exposés.

2. "Un système d'aide à l'enseignement d'une méthode de programmation"

Une méthodologie de construction de programmes basée sur la notion d'invariant est enseignée à l'Institut d'Informatique des Facultés Universitaires de Namur (cfr notamment le cours de "Preuves de programmes" enseigné par Monsieur B. Le Charlier en première licence).

Un système d'aide à l'apprentissage d'une méthodologie de construction de programmes selon une telle méthode est en cours de réalisation à l'Institut. Le système à définir est un support pour l'enseignement de la programmation : il aidera les étudiants à comprendre et à appliquer la méthode de l'invariant.

Un programme est caractérisé par différentes situations : la précondition, la situation générale (ou invariant) et la postcondition.

Une situation représente un instant donné de l'exécution du programme, elle est caractérisée par l'ensemble des valeurs des variables et tableaux du programme à cet instant, et par l'ensemble des conditions déterminant l'état dans lequel se trouve le programme à cette étape de l'exécution.

Je n'insisterai pas davantage sur les notions de précondition, de postcondition et d'invariant qui sont clairement définies dans le cours de première licence en informatique "Preuves de programme" ([PROG]), et rappelées dans [SYS88].

Selon la méthode de l'invariant, le programmeur doit raisonner en termes de situations et non plus en termes de succession d'actions. Il doit considérer une instruction (ou un ensemble d'instructions) comme une opération permettant de passer d'une situation à une autre : par exemple, les instructions de l'itération d'un programme permettent de passer d'une situation générale à la même situation générale, mais avec un ensemble éventuellement différent de valeurs associées ses variables et tableaux.

L'étudiant devra déduire les suites d'instructions constituant son programme des différentes situations qu'il aura définies.

Le système à concevoir sera basé d'une part sur un langage graphique de description de situations dont le concept fondamental est la notion de "segment", et d'autre part sur un langage mathématique d'expression d'assertions, basé sur les notions de "suite" et d'"ensemble".

L'objectif du système est d'aider l'étudiant à raisonner en termes de situations sur des problèmes simples, afin qu'ayant acquis une certaine rigueur de raisonnement il puisse résoudre des problèmes plus compliqués. Le système ne doit pas viser à aider l'étudiant lors de la résolution de problèmes complexes.

La classe des problèmes abordés sera donc la construction de programmes Pascal comportant au plus une boucle et manipulant uniquement des constantes, des variables simples et des tableaux à une dimension.

L'étudiant disposera d'un énoncé précis, et éventuellement d'une idée intuitive de solution. Pour construire le programme résolvant le problème soumis, il sera invité à suivre les étapes suivantes :

Introduction de la spécification du programme

Cette étape comporte trois parties distinctes :

- Création de la liste des objets manipulés par le programme (constantes, variables, tableaux),
- Introduction de la précondition du programme,
- Introduction de la postcondition du programme.

Introduction de la situation générale

Pour les programmes ayant une forme itérative, la situation générale est une description schématique de l'ensemble des conditions vérifiées par les objets utilisés avant chaque évaluation de la condition d'itération.

Choix des instructions

L'utilisateur du système sera invité à introduire les différentes suites d'instructions composant le programme. Ces suites d'instructions seront en principe déduites de la situation générale décrite.

Rappelons que les programmes traités comporteront au plus une séquence d'instructions itérative (souvent appelée "boucle du programme"). Les suites d'instructions à définir sont donc :

- Les instructions d'initialisation,
- La condition de terminaison de la boucle,
- Les instructions d'itération,
- Et, éventuellement, les instructions de clôture.

Vérification de la terminaison

La vérification de la terminaison se fait par la définition d'une fonction entière et bornée des valeurs des variables, qui croît strictement à chaque exécution de la suite d'instructions de l'itération.

Ecriture du programme

Le programme est construit par l'assemblage des différentes suites d'instructions introduites par l'étudiant.

Le système final envisagé sera doté de deux langages d'expression de spécifications et de situations :

- Un langage graphique de représentation de schémas, basé sur la notion de "segment",
- Un langage mathématique d'expression d'assertions, basé sur les notions de suite et d'ensemble.

Le système effectuera notamment les vérifications suivantes :

- Vérifications syntaxiques,
- Vérification de la cohérence entre l'expression graphique et mathématique d'une même situation,
- Vérification de la correction d'une séquence d'instructions par rapport à une situation initiale et une situation finale,

- Vérifications de cohérence entre différentes étapes de la démarche,
- Vérifications de complétude,
- ...

Nous espérons que suite à ces vérifications, le système sera à même de fournir des commentaires sur les différentes anomalies rencontrées afin d'éclairer au maximum l'utilisateur sur ses erreurs de raisonnement.

Notons également que le système devra proposer plusieurs scénarios d'utilisation en fonction des besoins de l'utilisateur (par exemple, l'utilisateur doit avoir le choix entre : suivre les différentes étapes en s'appuyant sur le langage de description graphique ou sur le langage mathématique).

Langage de description de schémas

Le langage de description graphique (ou encore langage de description de schémas) est basé sur le concept de segment.

Un programme est caractérisé par un ensemble de situations : une situation pouvant être du type précondition, situation générale ou postcondition.

Une situation est déterminée par l'ensemble des variables et tableaux du programme : appelons cet ensemble l'environnement. Une situation peut être déterminée par un ensemble de conditions caractérisant l'état de son environnement.

Une manière particulière de décrire l'état d'un tableau est d'utiliser des schémas : un tableau peut être décrit à l'aide d'un ou plusieurs schémas. Un schéma est constitué d'un ou plusieurs segments, un segment étant caractérisé par une borne inférieure et une borne supérieure.

Une borne peut être soit mobile soit fixe. Si elle est fixe elle est liée à la valeur d'une variable à l'instant d'exécution du programme correspondant à la situation décrite.

Les situations, les schémas et les segments peuvent éventuellement être nommés.

Notons que les conditions peuvent porter sur :

- une situation,
- un schéma,
- un segment,
- une borne,

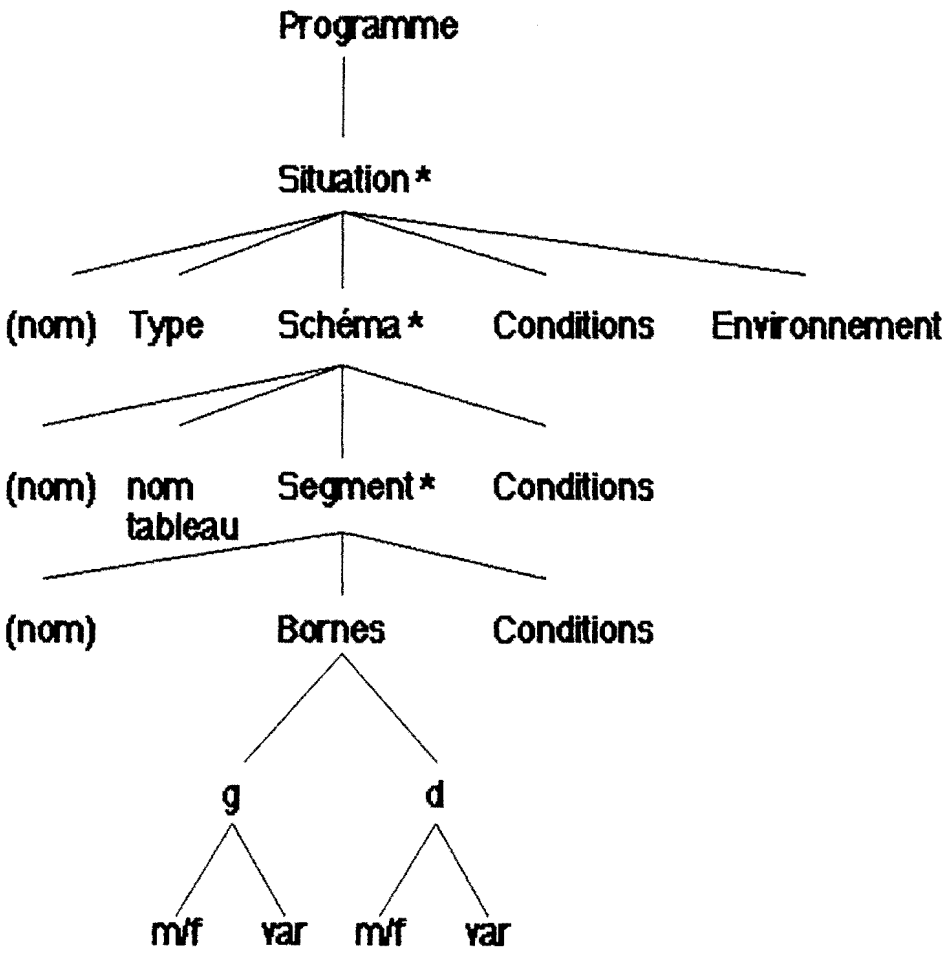
les conditions sur les bornes étant reprises dans les conditions sur les segments.

Cependant, cette classification des conditions est arbitraire, quelquefois il est difficile de déterminer à quelle classe une condition appartient :

- une condition peut porter sur plusieurs segments,
- une condition peut porter sur plusieurs tableaux,
- ...

De toute façon, l'important est de constater que quel que soit l'endroit où elle sont exprimées, les conditions portent toujours sur la situation décrite.

En fait, un programme (vu par le système) peut être modélisé comme suit :



Le langage proposé sera donc un outil de description et de manipulation de schémas dont le concept fondamental sera la notion de segment.

les principaux concepts et mécanismes de ce langage sont décrits dans le point "4. Langage de description de schémas" de l'article présenté dans l'annexe 1.

Langage mathématique

Les différentes caractéristiques de base de ce langage sont présentées dans le point "5. Langage mathématique" de l'annexe 1.

Le langage de description d'assertions défini dans le mémoire de D. Fisette ([FISETTE85]) serait une base possible de départ pour un tel langage mathématique.

Un scénario standard d'utilisation du système

Pour réaliser un tel système d'aide à la programmation, un scénario standard d'utilisation est proposé. Ce scénario standard est décrit plus en détail dans l'article en annexe 1, spécifiant de manière plus précise les différentes fonctionnalités du système proposé.

Rappelons simplement les différentes étapes que comportera ce scénario standard :

- Construction de la spécification
- Description de la situation générale
- Choix des instructions
- Vérification de la terminaison
- Ecriture du programme

D'autres scénarios d'utilisation du système seront proposés :

- suivre la démarche en n'exprimant les situations que sous forme de schémas,
- inversement, ne fournir au système que l'expression mathématique des situations,
- exécuter une ou plusieurs étapes particulières de la démarche, par exemple la dernière pour vérifier qu'un programme quelconque est syntaxiquement correct,
- ...

Plusieurs niveaux de vérification peuvent être définis selon la complexité du problème :

1. Vérifications simples de cohérence et complétude avec messages d'erreur.

2. Présentation des erreurs détectées sous forme de contre-exemples.

3. Interprétation de ces erreurs sur base d'une certaine expertise (détection des erreurs de raisonnement classiques).

4. Démonstrations formelles de l'exactitude des raisonnements.

Dans les limites du possible, ce système tentera de fournir à l'utilisateur l'aide de niveau maximum.

3. Participation de ce mémoire à la réalisation du système

Ce mémoire contribue à l'élaboration d'un système offrant un langage graphique de description d'assertions.

Il est constitué de trois parties distinctes :

- Partie 1 :

  La définition d'un langage pour l'expression de conditions, approprié à l'outil de description graphique du système.

  Ce langage est présenté sur base d'exemples choisis pour illustrer ses caractéristiques originales, au chapitre 2. Il est complètement défini à l'annexe 2.

- Partie 2 :

  La définition d'un premier scénario d'utilisation du système.

  Ce scénario est présenté au chapitre 3.

- Partie 3 :

  Un projet de découpe en modules pour la mise en oeuvre du système, et l'implémentation de deux modules de l'architecture proposée.

  Cette première découpe en modules est définie à l'annexe 3 et présentée dans le chapitre 4.

  L'implémentation de deux modules de cette architecture est reprise dans l'annexe 4, et est commentée dans le chapitre 5.

Chapitre 2

Présentation du langage de conditions

1. Introduction

Ayant fixé les objectifs d'un système d'aide à la programmation et la contribution de ce mémoire à la réalisation de ce système, passons à la présentation de la première partie de ce mémoire : la définition d'un langage de conditions pour la description d'assertions (langage approprié à la description graphique des situations d'un programme).

Le but du système n'est pas de résoudre des problèmes complexes mais d'aider les étudiants à acquérir un raisonnement rigoureux en s'exerçant sur des problèmes simples.

Les programmes à construire ne comporteront que des variables de type entier et booléen, et des tableaux de type entier à une seule dimension. Il en découle que le langage de conditions ne manipulera que de telles structures.

Notons que par la suite ce langage pourrait être étendu de la manière suivante :

- variables de type caractère,
- tableaux de type caractère à une seule dimension,
- tableaux de type booléen à une seule dimension.

Dans le langage de conditions, il existe une liste de fonctions prédéfinies. Par souci de facilité de définition et d'implémentation cette liste est limitative : elle contient les seules fonctions exprimables dans le langage.

Etant donné que le langage de description graphique est basé sur le concept de segment, les fonctions prédéfinies du langage portent sur des segments.

Le langage de conditions ne permet pas l'expression directe des quantificateurs bien connus : "$\forall$" et "$\exists$".

Cependant, le quantificateur universel est, comme prévu dans l'article présenté dans l'annexe 1, remplacé par l'expression "dot", notée $\bullet$.

Le "$\bullet$" désigne : "un élément quelconque (du segment sur lequel il porte)".

Exemple

Soit a [1:n] un tableau de type entier,

si S est le segment a [1:i], où $0 \leq i \leq n$, alors

l'expression : $\bullet$ (S) ≥ 0

signifie : $\forall\ j : 1 \leq j \leq i : S[j] \geq 0$.

L'expression d'un quantificateur existentiel n'était pas prévue dans l'article en annexe 1, cependant nous avons constaté que des conditions du style "il existe un élément du segment tel que ..." sont souvent employées dans l'expression de situations.

Nous avons donc introduit une manière implicite d'exprimer la quantification existentielle similaire à l'expression implicite de la quantification universelle.

Le quantificateur "$\exists$" est remplacé par l'expression "$\mathbf{\Lambda}$".

Cette expression désigne un "certain élément du segment (sur lequel elle porte)", et peut être lue comme suit :

"il existe un élément du segment".

Exemple

Soit a [1:n] un tableau de type entier,

si S est le segment a [1:i], où $0 \leq i \leq n$, alors

l'expression : $\mathbf{\Lambda}$ (S) ≥ 0

signifie : $\exists\ j : 1 \leq j \leq i : S[j] \geq 0$.".

Pour plus de détails concernant l'introduction de ces pseudo-quantificateurs et les conséquences sur l'évaluation d'une condition, veuillez consulter la définition complète du langage en annexe 2.

La lecture de la définition du langage de conditions pouvant s'avérer fastidieuse, celle-ci est présentée en annexe (cfr l'annexe 2). Nous illustrerons ici l'utilisation de ce langage à travers quelques exemples simples.

L'analyse de ces exemples nous permettra de voir quelles sont les conditions que l'on peut exprimer à l'aide de ce langage,

éventuellement de dégager ses faiblesses et de suggérer des améliorations possibles.

Pour chaque exemple, nous donnerons une spécification informelle, ensuite nous exprimerons cette spécification (précondition et postcondition) dans un langage plus formalisé (notations utilisées dans le cours de première licence "Preuves de programmes" de B. Le Charlier). Nous exprimerons aussi de cette manière la situation générale (ou invariant) du programme.

Ensuite, nous exprimerons les différentes situations (précondition, postcondition et situation générale) à l'aide du langage de conditions défini à l'annexe 2.

Nous exprimerons les différentes conditions relatives :

- à la situation décrite,
- aux différents schémas décrivant l'état des tableaux :
    - conditions sur les différents segments,
    - conditions sur les bornes de ces segments.

L'expression des conditions sur les schémas et les segments sera illustrée graphiquement.

La représentation graphique reprend toutes les conditions exprimées dans le langage de conditions. Nous verrons cependant que certaines conditions ne sont pas reprises explicitement sur les schémas car elles sont déductibles de la représentation graphique.

L'organigramme attendu suite à la démarche de réflexion demandée à l'étudiant sera aussi présenté.

Ensuite nous examinerons les conditions exprimées (ou non-exprimées !).

Afin de ne pas alourdir le texte et pour éviter toute redondance, les explications ne seront pas répétées pour les exemples suivants.

2. Illustration 1 : un programme sans tableau

Le langage de description graphique n'est pas destiné aux programmes ne manipulant pas de tableaux, cependant nous allons montrer par un exemple simple qu'il peut être utilisé pour décrire les situations de tels programmes.

2.1. Spécifications

Programme de multiplication de deux entiers :

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.

Ce qui de manière plus formelle et selon la méthode de l'invariant peut être exprimé comme suit :

2.1.1. Précondition

- a, b initialisées
- $b \geq 0$

2.1.2. Postcondition

- $p = a * b$
- a, b inchangées

2.1.3. Invariant

- $0 \leq i \leq b$
- $p = a * i$

2.3. Expression des conditions à l'aide du langage de conditions

2.3.1. La précondition

Cette situation sera caractérisée par la condition suivante :

- b >= 0

2.3.2. La postcondition

Cette situation sera caractérisée par la condition suivante :

- p = a * b

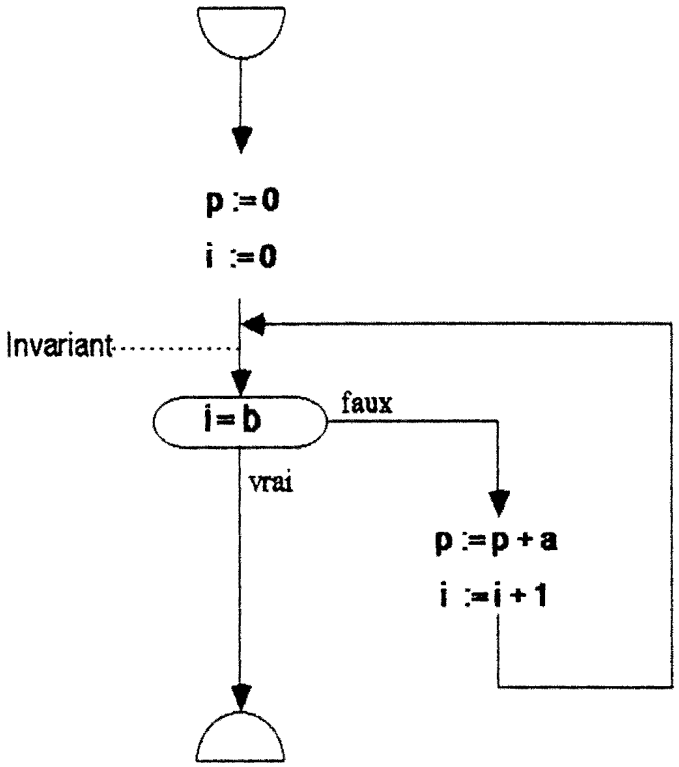
2.3.3. La situation générale

Cette situation sera caractérisée par les conditions suivantes :

- i <= b
- i >= 0
- p = a * i

2.2. Organigramme

a, b, p, i : variables de type entier



2.4. Commentaires

Dans cet exemple, nous pouvons déjà constater un certain nombre de limites de ce langage de conditions.

Ce langage ne permet pas d'exprimer le fait que lorsqu'une variable est une donnée, celle-ci doit avoir reçu une valeur (avoir été initialisée).

Bien que par le biais des fonctions prédéfinies il soit possible d'exprimer la non-modification des valeurs des éléments d'un tableau, il n'est pas possible d'exprimer une telle condition sur les valeurs des variables simples. En effet, jusqu'à présent, les fonctions

prédéfinies du langage portent uniquement sur des segments ou tableaux.

Peut être faudrait-il créer de nouvelles fonctions ayant comme argument(s) des variables ?

Etant donné la définition de la vérification d'une situation donnée dans l'annexe 2, les conditions exprimées sur la situation (par exemple les conditions exprimant la situation générale) n'ont pas besoin d'être reliées par une conjonction logique "et". En effet, ce "et" est implicite : les conditions listées sont implicitement enchaînées par des conjonctions "et" puisqu'une situation est vérifiée si toutes les conditions exprimées sur celle-ci sont vraies.

Il est évident que la description des différentes situations de ce programme ne comporte aucun schéma puisqu'il ne manipule aucun tableau.

3. Illustration 2 : un programme manipulant un tableau

Analysons maintenant des programmes manipulant des structures de type tableau. Commençons par un programme simple nécessitant l'utilisation de la quantification existentielle implicite (**Λ**).

3.1. Spécifications

Programme vérifiant l'appartenance d'une valeur à un tableau :

Etant donné a, un tableau d'entiers initialisé, et x, une variable entière initialisée, affecter à la variable booléenne présent la valeur **vrai** si la valeur d'un des éléments du tableau a est égale à la valeur de la variable x, et **faux** sinon.

Ce qui de manière plus formelle et selon la méthode de l'invariant peut être exprimé comme suit :

3.1.1. Précondition

- a, x initialisés

3.1.2. Postcondition

- a inchangé
- x inchangée
- présent = (x ∈ {a [1:n]})

3.1.3. Invariant

- $0 \leq i \leq n$
- présent = (x ∈ {a [1:i] })

3.2. Expression des conditions à l'aide du langage de conditions

3.2.1. La précondition

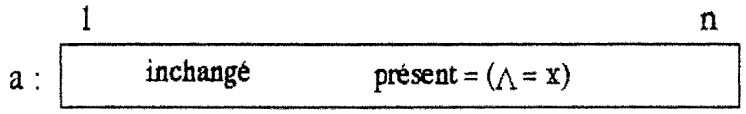
3.2.2. La postcondition

La description de la postcondition comporte un seul schéma décrivant l'état du tableau a.

Ce schéma est déterminé par les conditions

- a **inchangé**
- présent = (**Λ** (a[1:n]) = x)

Représentation du schéma



3.2.3. La situation générale

La description de la situation comporte un schéma décrivant l'état du tableau a.

Ce schéma est divisé en deux segments :

- le segment a1 = a [1:i], et
- le segment a2 = a[i+1:n].

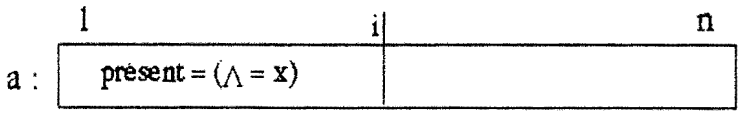
L'état du segment a1 est déterminé par les conditions

- présent = (**Λ** (a1) = x)

(conditions sur la borne i)

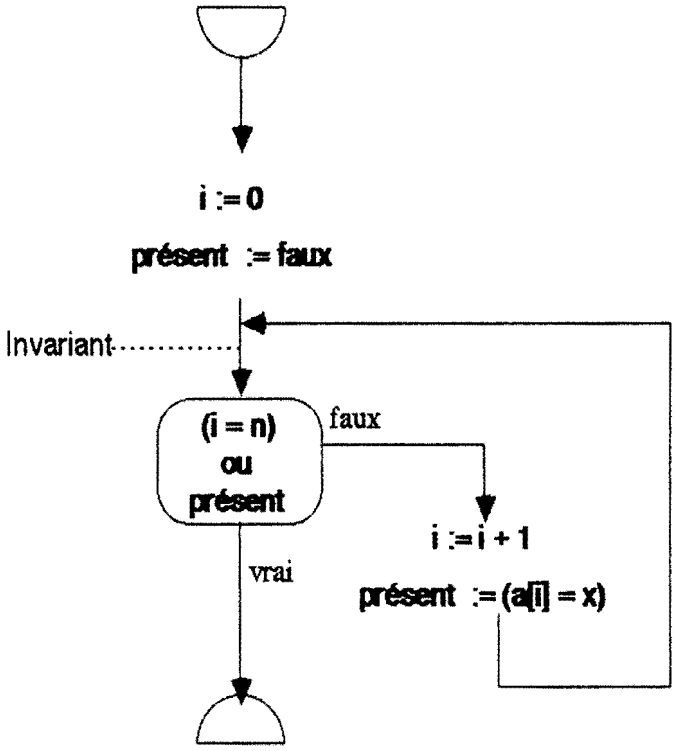
- i >= 0
- i <= n

Représentation du schéma



3.3. Organigramme

- i, x : variables de type entier
- présent : variable de type booléen
- a [1:n] : tableau de type entier (n ≥ 1)



3.4. Commentaires

Comme dans l'exemple précédent, il est impossible d'exprimer le fait qu'une variable doit avoir été initialisée avant d'être utilisée, et il est également impossible d'exprimer une telle condition sur un tableau.

Dans l'expression de la postcondition, nous rencontrons l'utilisation de la fonction prédéfinie **inchangé**.

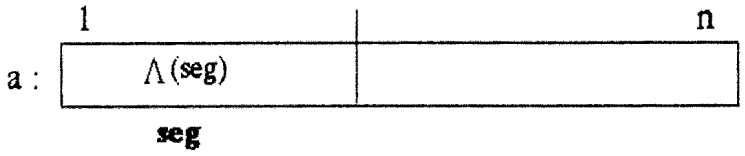
Cette fonction permet d'exprimer le fait que les valeurs des éléments du tableau n'ont pas été modifiées au cours de l'exécution du programme.

Nous ne disposons cependant pas d'une fonction permettant d'exprimer le fait qu'une variable n'a pas été modifiée.

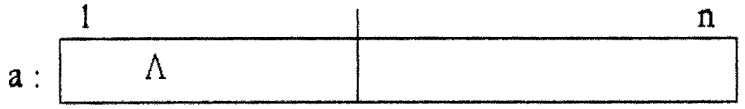
Dans la description graphique cette condition peut être notée "**inchangé**", la mention du nom du tableau n'est pas nécessaire : en effet, la description graphique permet de retrouver le segment (ou tableau, le tableau étant un segment particulier) sur lequel porte la fonction **inchangé**.

De manière plus générale :

le schéma :



est équivalent au schéma :



Cet exemple nous permet d'illustrer une utilisation simple de l'expression **Λ** :

- la condition "présent = (x ∈ {a [1:n]})" de la postcondition et
- la condition "présent = (x ∈ {a [1:i] })" de l'invariant

ne sont pas exprimables comme telles dans le langage de conditions. Cependant ces conditions peuvent être reformulées comme suit :

- présent = (∃ j : 1 ≤ j ≤ n : a [j] = x), et
- présent = (∃ j : 1 ≤ j ≤ i : a [j] = x).

Dès lors, le langage de conditions permet d'exprimer la quantification existentielle comme suit :

- présent = (**Λ** (a[1:n]) = x),
- présent = (**Λ** (a[1:i]) = x).

Dans la représentation graphique de la situation générale, les conditions sur la borne i du segment al ne sont pas exprimées car elles sont implicites à la représentation des segments constituant le schéma.

En effet, lorsqu' aucune condition n'est exprimée sur les bornes des segments constituant un schéma, la représentation graphique de celui-ci donne lieu aux interprétations suivantes :

- entre bornes d'un même segment :

  soient bi, la borne inférieure et bs, la borne supérieure,

  - soit bi est inférieure ou égale à bs : bi ≤ bs,
  - soit bi est supérieure à bs : bi > bs (cas d'un segment vide),

- entre bornes de segments contigus :

  soient S1 et S2, respectivement le premier et le deuxième segment d'un tableau, bs1, la borne supérieure de S1, et bi2, la borne inférieure de S2,

  - bs1 est égale à bi2 moins un : bs1 = bi2 - 1 (ou encore : bi2 = bs1 + 1).

Or, i est la borne supérieure du segment a1,

- soit elle est supérieure ou égale à la borne inférieure de ce segment : i ≥ 1,
- soit elle est inférieure : i < 1 et dans ce cas : i = 0.

La représentation permet donc de déduire la condition : i ≥ 0.

Le segment a2 étant contigu au segment a1,

- sa borne inférieure est égale à la borne supérieure de a1 plus 1,

or la borne inférieure de a2 est

- soit inférieure ou égale à la borne supérieure de ce segment, et donc : i + 1 ≤ n (ou encore i ≤ n - 1),
- soit supérieure à cette borne : i + 1 > n (ou encore i > n - 1), dans ce cas le segment a2 est vide et la borne supérieure de a1 coïncide avec la borne supérieure du tableau a : i = n.

La représentation permet donc de déduire la condition : i ≤ n.

La représentation graphique du schéma offre donc la formulation implicite des conditions sur les bornes.

4. Illustration 3 : la recherche dichotomique

Reprenons l'exemple précédent, mais cette fois en implémentant une recherche dichotomique.

4.1. Spécifications

Programme vérifiant l'appartenance d'une valeur à un tableau :

Etant donné a, un tableau d'entiers initialisé, et x, une variable entière initialisée, affecter à la variable booléenne présent la valeur **vrai** si la valeur d'un des éléments du tableau a est égale à la valeur de la variable x, et **faux** sinon.

La recherche ne sera plus séquentielle comme dans l'exemple précédent, mais dichotomique.

Ce qui de manière plus formelle et selon la méthode de l'invariant peut être exprimé comme suit :

4.1.1. Précondition

- a trié : $a[1] \leq a[2] \leq \ldots \leq a[n]$
- x initialisée

4.1.2. Postcondition

- a inchangé
- x inchangée
- présent = $(x \in \{a\ [1:n]\})$

4.1.3. Invariant

- $0 \leq g \leq d+1 \leq n+1$
- $x \notin \{a\ [1:g-1]\}$
- $x \notin \{a\ [d+1:n]\}$

4.2. Expression des conditions à l'aide du langage de conditions

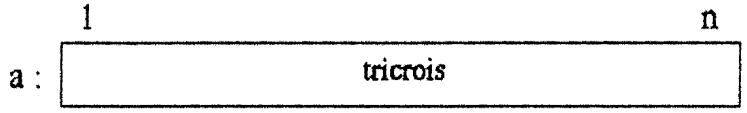
4.2.1. La précondition

La description de la situation comporte un seul schéma décrivant l'état du tableau a.

Ce schéma est déterminé par la condition

- a **tricrois**

Représentation du schéma



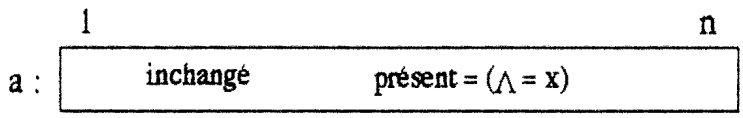
4.2.2. La postcondition

La description de la situation comporte un seul schéma décrivant l'état du tableau a.

Ce schéma est déterminé par les conditions

- a **inchangé**
- présent = (**Λ** (a[1:n]) = x)

Représentation du schéma



4.2.3. La situation générale

La description de la situation comporte un schéma décrivant l'état du tableau a.

Ce schéma est divisé en trois segments :

- le segment a1 = a [1:g-1]
- le segment a2 = a[g:d],
- le segment a3 = a[d+1:n].

Le segment a1 est caractérisé comme suit

- • (a[1:g-1]) <> x

Le segment a2 est caractérisé comme suit

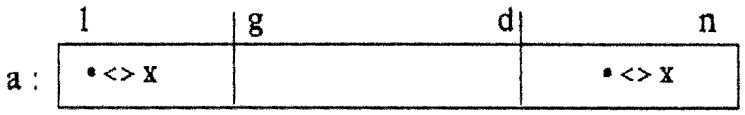
(conditions sur les bornes d et g)

- 0 <= g
- g <= d + 1
- d <= n

Le segment a3 est caractérisé comme suit

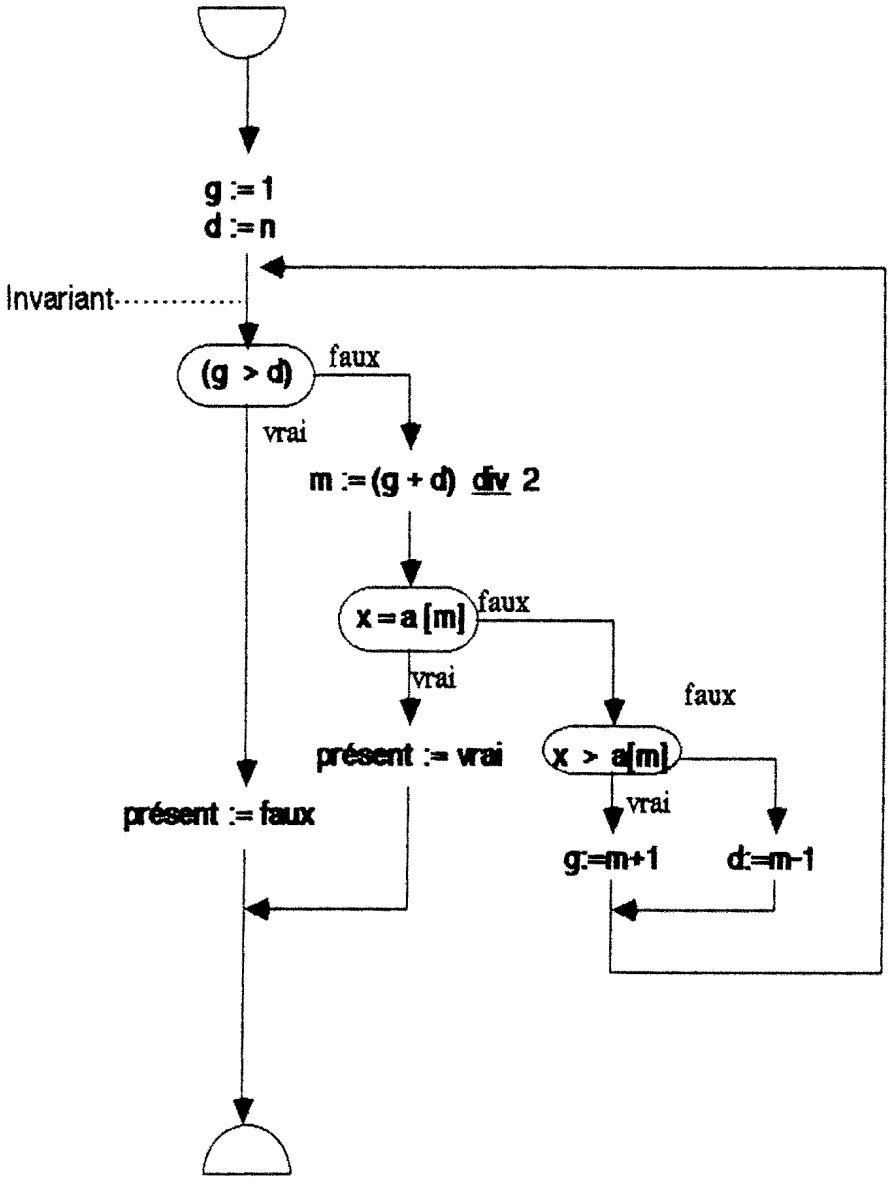
- • (a[d+1:n]) ◇ x

Représentation du schéma



4.3. Organigramme

- g, d, m : variables de type entier
- présent : variable de type booléen
- a [l:n] : tableau de type entier (n ≥ 1)



4.4. Commentaires

Ce programme est plus efficace que le précédent, mais ne peut être utilisé pour effectuer une recherche que dans des tableaux triés par ordre croissant.

Il diffère donc du précédent,

- dans l'expression de la précondition : le tableau manipulé doit être trié,
- dans l'expression de la situation générale (ou invariant) : la recherche n'est plus implémentée de manière séquentielle.

L'expression de la précondition fait apparaître l'utilisation d'une fonction prédéfinie du langage : la fonction **tricrois**, elle exprime le fait que le tableau **a** doit être trié en ordre croissant.

Notons que le langage offre d'autres fonctions permettant d'exprimer qu'un segment (ou tableau) est trié en ordre décroissant (fonction **tridec**), trié en ordre croissant strict (fonction **tristrcrois**), ou encore trié en ordre décroissant strict (fonction **tristrdec**).

Dans l'exemple précédent, la situation générale était caractérisée par la condition : présent = (**Λ** (a1) = x), exprimant que la variable booléenne présent vaut vrai s'il existe un élément du segment a1 dont la valeur est égale à la valeur de la variable x, et faux sinon, le segment a1 représentant la partie examinée du tableau.

Dans la recherche dichotomique, les conditions caractérisant la situation générale sont les suivantes :

- x ∉ {a [1:g-1]},
- x ∉ {a [d+1:n]}.

Ces conditions expriment le fait que les parties du tableau examinées ne contiennent pas d'éléments dont la valeur soit égale à celle de la variable x, ou encore que tous les éléments des segments examinés sont différents de x, ce qui dans le langage de conditions se traduit simplement à l'aide du quantificateur universel • :

- • (a[1:g-1]) ◇ x,
- • (a[d+1:n]) ◇ x.

Constatons également que les conditions sur les différents segments d'un schéma sont implicitement reliées par des "et", en effet, une situation est vérifiée si toutes les conditions de la situation sont vérifiées : conditions sur la situation, conditions sur les schémas, conditions sur les segments et conditions sur les bornes.

5. Illustration 4 : un programme manipulant deux tableaux

Le programme suivant manipule deux tableaux, et les situations décrites comportent des schémas décrivant l'état de chacun.

5.1. Spécifications

Programme recherchant la plus petite valeur commune à deux tableaux :

Etant donné a et b, deux tableaux d'entiers initialisés ayant au moins une valeur commune, affecter à la variable x la plus petite valeur commune aux tableaux a et b.

Ce qui de manière plus formelle et selon la méthode de l'invariant peut être exprimé comme suit :

5.1.1. Précondition

- a, b triés en ordre croissant :

  $a[1] \leq a[2] \leq ... \leq a[n]$

  $b[1] \leq b[2] \leq ... \leq b[m]$

- $\exists\ i, j,\ 1 \leq i \leq n\ ,\ 1 \leq j \leq m\ :\ a[i] = b[j]$

5.1.2. Postcondition

- a, b inchangés
- x = la plus petite valeur commune de a et b

5.1.3. Invariant

- $1 \leq i \leq n$
- $1 \leq j \leq m$
- $\{a[1:i-1]\} \cap \{b[1:j-1]\} = \emptyset$
- $a[i-1] < b[j]$
- $b[j-1] < a[i]$

+ convention $a[0] = b[0] = -\infty$

5.2. Expression des conditions à l'aide du langage de conditions

5.2.1. La précondition

La précondition est caractérisée par la condition :

- **Λ** (a[1:n]) = **Λ** (b[1:m]),

et par les schémas suivants :

- le schéma 1 décrivant l'état du tableau a,
- le schéma 2 décrivant l'état du tableau b.

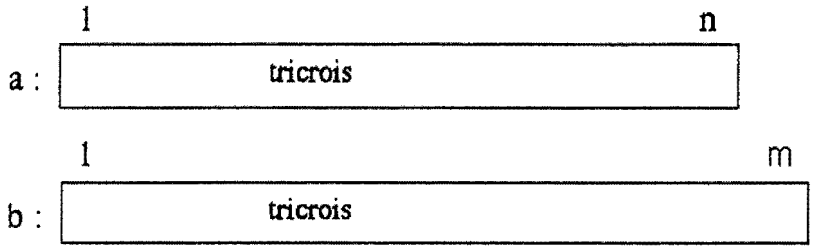
Condition sur le schéma 1

- a **tricrois**

Condition sur le schéma 2

- b **tricrois**

Représentation des schémas



5.2.2. La postcondition

La postcondition est caractérisée par les conditions suivantes :

- x = a[i] = b[j]
- • (a[1:i-1]) <> • (b[1:j-1])

et par les schémas suivants :

- le schéma 1 décrivant l'état du tableau a, constitué :
  - du segment a1 = a [1:i-1], et
  - du segment a2 = a[i:n],
- le schéma 2 décrivant l'état du tableau b, constitué :
  - du segment b1 = b [1:j-1], et
  - du segment b2 = b[j:n].

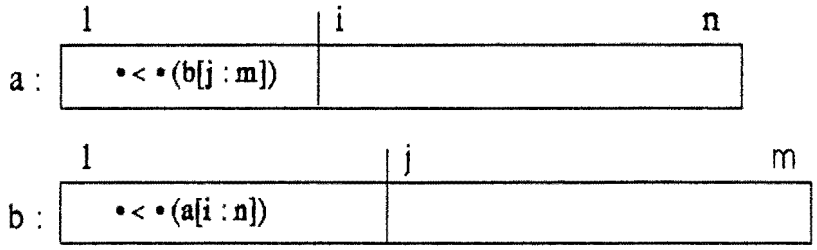
Condition sur le segment a1

- • (a[1:i-1]) < • (b[j:m])

Condition sur le segment b1

- • (b[1:j-1]) < • (a[i:n])

Représentation des schémas



5.2.3. La situation générale

La situation générale est caractérisée par la condition :

- • (a[1:i-1]) <> • (b[1:j-1])

et par les schémas suivants :

- le schéma 1 décrivant l'état du tableau a, constitué :
  - du segment a1 = a [1:i-1], et
  - du segment a2 = a[i:n],
- le schéma 2 décrivant l'état du tableau b, constitué :
  - du segment b1 = a [1:j-1], et
  - du segment a2 = a[j:n].

Condition sur le segment a1

- a[i-1] < b[j]

Conditions sur le segment a2

(conditions sur la borne i)

- 1 <= i
- i <= n

Condition sur le segment b1

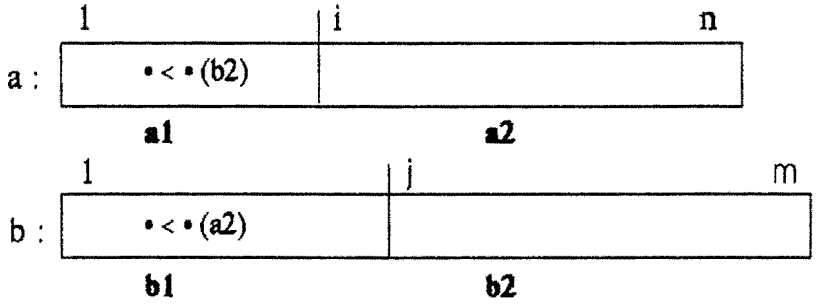
- b[j-1] < a[i]

Conditions sur le segment b2

(conditions sur la borne j)

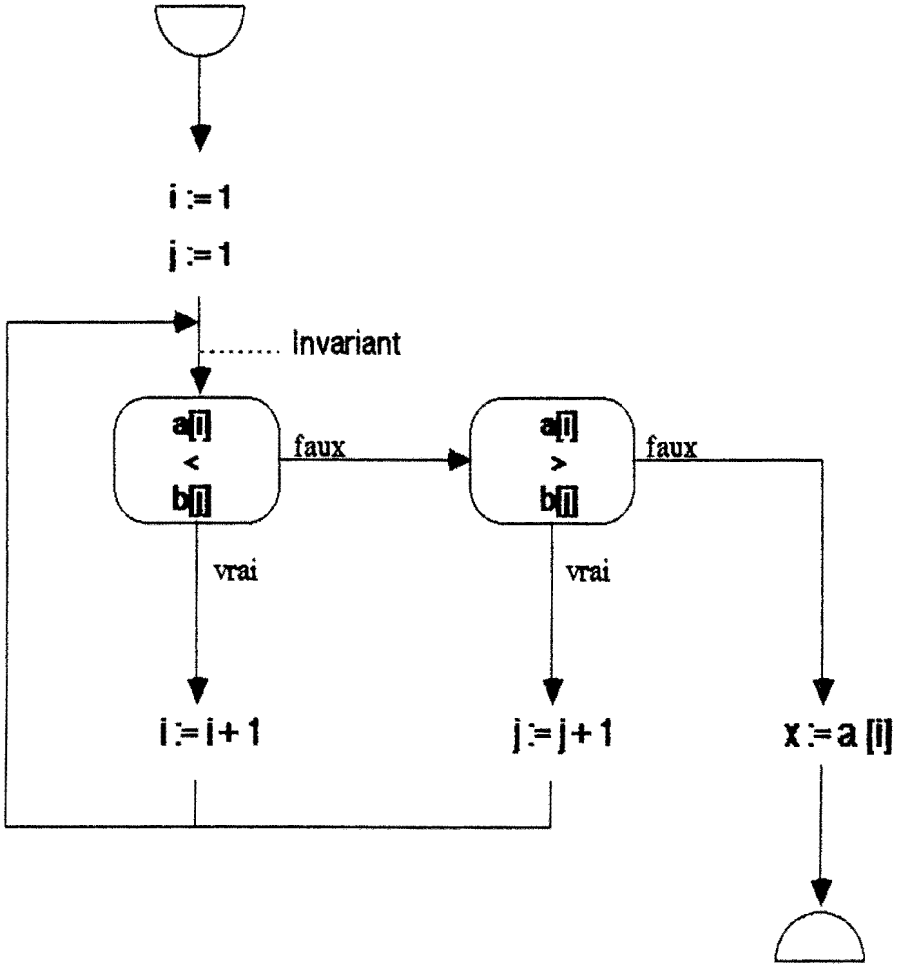
- 1 <= j
- j <= m

Représentation des schémas



5.3. Organigramme

- a[1:n], b[1:m] : tableaux d'entiers (n, m ≥ 1)
- i, j, x : variables entières



5.4. Commentaires

Dans la précondition, l'existence d'au moins une valeur commune aux deux tableaux est exprimée comme suit :

- **Λ** (a[1:n]) = **Λ** (b[1:m]),

notons qu'un symbole "∃" est représenté par deux symboles **Λ** :

- ∃ i,j ⇔ ∃ i, ∃ j.

Notons ici la souplesse du langage graphique :

la condition **Λ** (a[1:n]) = **Λ** (b[1:m]) portant sur la situation peut être introduite au niveau d'un des deux schémas, soit dans la description du tableau a : **Λ** = **Λ** (b), soit dans la description du tableau b : **Λ** (a) = **Λ**, puisque toutes les conditions sont implicitement reliées par des "et".

Le langage de conditions n'offre pas les concepts de suite et d'ensemble (développés dans le mémoire de D. Fisette [FISETTE85]).

Le langage offert est destiné à exprimer des conditions à l'aide d'un support graphique.

Ainsi, il n'est pas possible d'exprimer directement la condition

- {a[1:i-1]} ∩ {b[1:j-1]} = Ø,

caractérisant la situation générale.

Cependant, cette condition peut être reformulée de la manière suivante :

- ∀ k, l : 1 ≤ k ≤ i-1, 1 ≤ l ≤ j-1 : a [k] ◇ b [l],

pouvant facilement être exprimé dans le langage de conditions :

- • (a[1:i-1]) ◇ • (b[1:j-1]),

qui correspond à une expression de l'idée intuitive "tous les éléments du segment a1 sont différents de tous les éléments du segments b1" sans faire intervenir la notion plus complexe d'ensemble de valeurs.

6. Illustration 5 : combinaisons de • et de Λ

Nous allons voir comment le langage de conditions réalise la combinaison d'un quantificateur universel et d'un quantificateur existentiel.

6.1. Spécifications

Programme de contraction d'un vecteur d'entiers trié :

Soit a[1:n], n ≥ 1, un vecteur d'entiers trié par ordre croissant. Modifier le contenu de b[1:n] de sorte que b[1:j], où 1 ≤ j ≤ n, contienne les valeurs de a[1:n] triées par ordre croissant et sans répétitions.

Ce qui de manière plus formelle et selon la méthode de l'invariant peut être exprimé comme suit :

6.1.1. Précondition

- n ≥ 1
- a trié par ordre croissant : a[1] ≤ a[2] ≤ ...≤ a[n]

6.1.2. Postcondition

- 1 ≤ j ≤ n
- les j premiers éléments de b rangés par ordre strictement croissant :

  b[1] < b[2] < ... < b[j]
- {a [1..n]} = {b [1..j]}
- a inchangé

6.1.3. Invariant

- 1 ≤ j ≤ i ≤ n
- les j premiers éléments de b rangés par ordre strictement croissant :

  b[1] < b[2] < ...< b[j]
- {a [1..i]} = {b [1..j]}

6.2. Expression des conditions à l'aide du langage de conditions

6.2.1. La précondition

La description de la situation comporte un schéma décrivant l'état du tableau a.

Ce schéma est caractérisé par les conditions suivantes

- a **tricrois**

(Condition sur la borne supérieure n de a)

- n >= 1

Représentation du schéma

6.2.2. La postcondition

La description de la situation comporte deux schémas :

- le schéma 1 décrivant l'état du tableau a,
- le schéma 2 décrivant l'état du tableau b, constitué :
  - du segment b1 = b [1:j], et
  - du segment b2 = b[j+1:n].

Conditions sur le schéma 1

- a **inchangé**
- • (a[1:n]) = **Λ** (b[1:j])

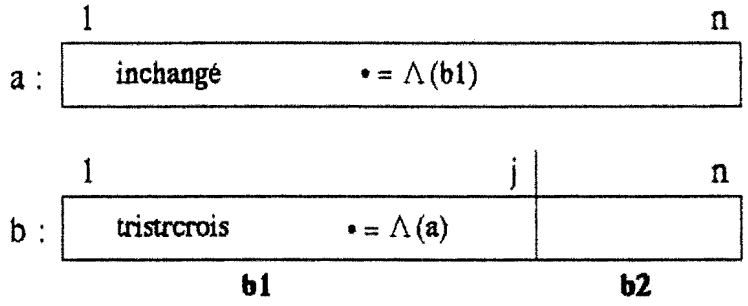
Conditions sur le segment b1

- b1 **tristrcrois**
- • (b[1:j]) = **Λ** (a[1:n])

(conditions sur la borne j)

- j >= 0
- j <= n

Représentation des schémas



6.2.3. La situation générale

La description de la situation comporte deux schémas :

- le schéma 1 décrivant l'état du tableau a, constitué :
  - du segment a1 = a [1:i], et
  - du segment a2 = a[i+1:n],
- le schéma 2 décrivant l'état du tableau b, constitué :
  - du segment b1 = b [1:j], et
  - du segment b2 = b[j+1:n].

Conditions sur le segment a1

- • (a[1:i]) = **Λ** (b[1:j])

(conditions sur la borne i et sur la borne j)

- i >= 1
- i <= n
- i >= j

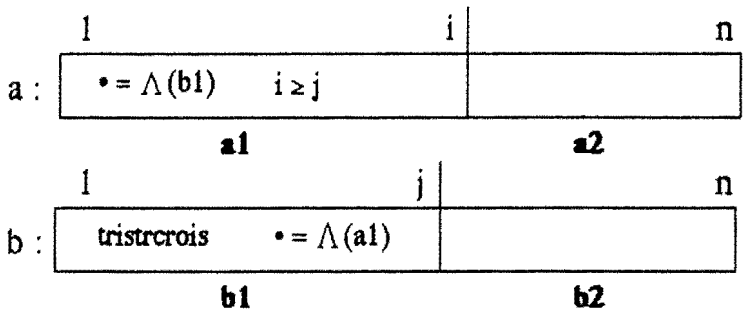
Conditions sur le segment b1

- b1 **tristrcrois**
- • (b[1:j]) = **Λ** (a[1:i])

(conditions portant sur la borne j et sur la borne j)

- j >= 1
- j <= n

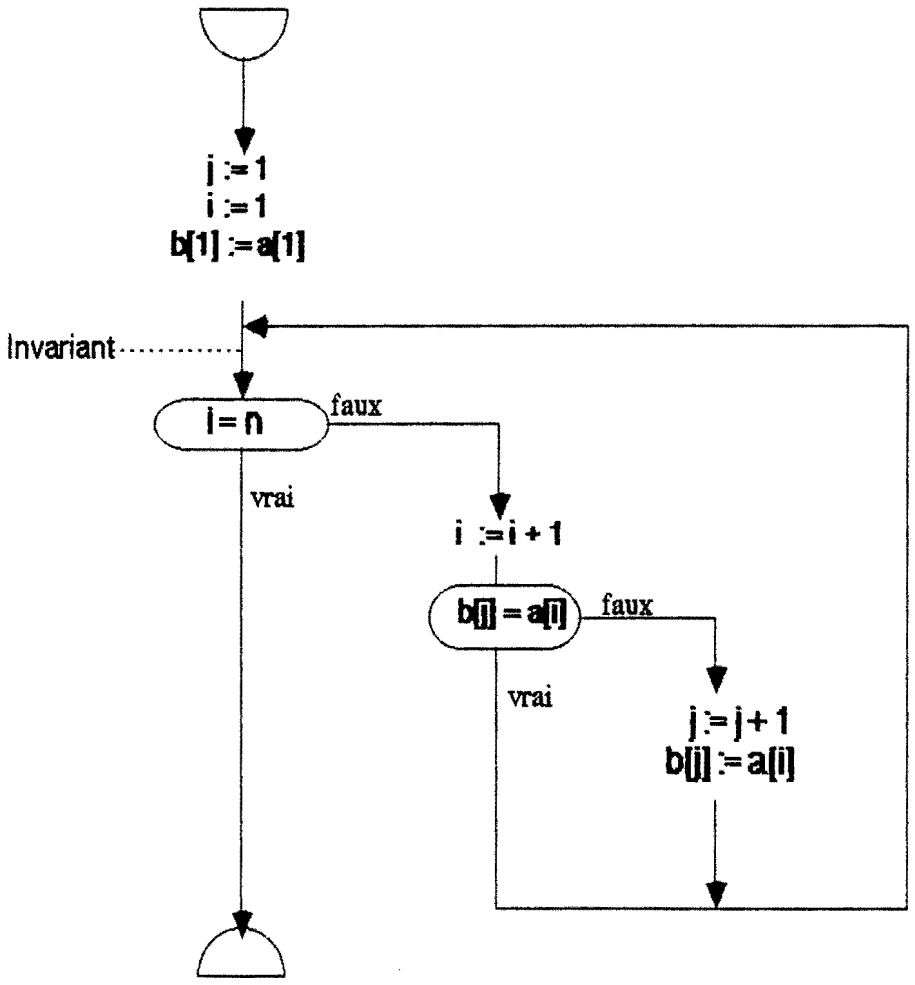
Représentation des schémas



6.3. Organigramme

a, b : tableaux de type entier

i, j : variables entières



6.4. Commentaires

Dans l'expression graphique de la précondition, nous trouvons la condition "n ≥ 1", condition sur la borne supérieure du tableau a, or nous avons déjà expliqué que les conditions sur les bornes étaient implicites à la représentation des segments. Cependant, les conditions exprimées par la représentation graphique permettent d'avoir un segment vide. La condition "n ≥ 1" exprime que le tableau a ne peut être vide : il doit au moins contenir un élément.

La postcondition et la situation générale de ce programme contiennent des conditions faisant intervenir des opérateurs sur les ensembles de valeurs d'un segment.

- {a [1..n]} = {b [1..j]},

- {a [1..i]} = {b [1..j]}.

Le langage de conditions n'offre pas de tels opérateurs.

La première de ces deux conditions est équivalente à la formulation suivante :

- $\forall$ k : 1 ≤ k ≤ j, $\exists$ r : 1 ≤ r ≤ n : b[k] = a[r],

- $\forall$ r : 1 ≤ r ≤ n, $\exists$ k : 1 ≤ k ≤ j : a[r] = b[k].

Ce qui dans le langage de conditions s'exprime comme suit :

- • (b[1:j]) = **Λ** (a[1:n]),

- • (a[1:n]) = **Λ** (b[1:j]).

Il me semble nécessaire d'insister sur l'importance de l'ordre d'apparition des symboles • et **Λ** dans les conditions.

En effet, soit la première de ces deux conditions :

- • (b[1:j]) = **Λ** (a[1:n]), (1)

elle signifie :

- $\forall$ k : 1 ≤ k ≤ j, $\exists$ r : 1 ≤ r ≤ n : b[k] = a[r],

si l'apparition des symboles est intervertie de la manière suivante :

- **Λ** (a[1:n]) = • (b[1:j]), (2)

cette condition signifie :

- $\exists$ r : 1 ≤ r ≤ n, $\forall$ k : 1 ≤ k ≤ j : a[r] = b[k].

Ainsi, supposons :

- le tableau a = [1, 2, 2, 3, 3, 3, 4, 5],
- le segment b [1:j] = [1, 2, 3, 4, 5],

dans ce cas, la condition (1) sera évaluée à vrai, alors que la (2) sera évaluée à faux.

Par contre, si:

- a = [1, 2, 2, 3, 3, 3, 4, 5], et
- b [1:j] = [2, 2],

alors, la condition (1) vaudra faux, et la (2) sera évaluée à vrai.

7. Illustration 6 : plusieurs schémas pour un même tableau

7.1. Spécifications

Programme permutant un vecteur, en plaçant ses éléments strictement négatifs à gauche et les autres, à droite :

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

Ce qui de manière plus formelle et selon la méthode de l'invariant peut être exprimé comme suit :

7.1.1. Précondition

- a [1:n] initialisé

7.1.2. Postcondition

- a [1:n] permuté de sorte que : $\exists\ i_S : 1 \le i_S \le n$:

  $(\forall\ k : 1 \le k \le i_S : a[k] < 0$

  et

  $\forall\ q : i_S + 1 \le q \le n : a[q] \ge 0)$

7.1.3. Invariant

- $1 \le i \le j+1 \le n+1$
- a [1:n] permuté de sorte que
  - $\forall\ k : 1 \le k \le i-1 : a[k] < 0$
  - $\forall\ q : j+1 \le q \le n : a[q] \ge 0$

7.2. Expression des conditions à l'aide du langage de conditions

7.2.1. La précondition

7.2.2. La postcondition

La description de la situation comporte deux schémas décrivant l'état du tableau a :

- le schéma 1,
- le schéma 2, constitué :
  - du segment a1 = a [1:i], et
  - du segment a2 = a[i+1:n].

Condition sur le schéma 1

- **permuté** (a [1:n])

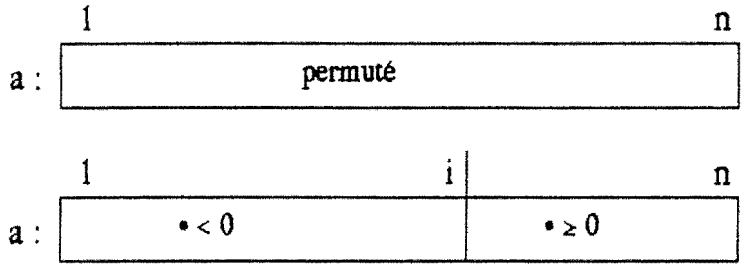
Condition sur le segment a1

- • (a[1:i]) < 0

Condition sur le segment a2

- • (a[i+1:n]) >= 0

Représentation des schémas



7.2.3. La situation générale

La description de la situation comporte deux schémas décrivant l'état du tableau a :

- le schéma 1,
- le schéma 2, constitué :
  - du segment a1 = a [1:i-1],
  - du segment a2 = a [i:j],
  - du segment a3 = a [j+1:n].

Condition sur le schéma 1

- **permuté** (a [1:n])

Condition sur le segment a1

- • (a[1:i-1]) < 0

Conditions sur le segment a2

(conditions sur la borne i et la borne j)

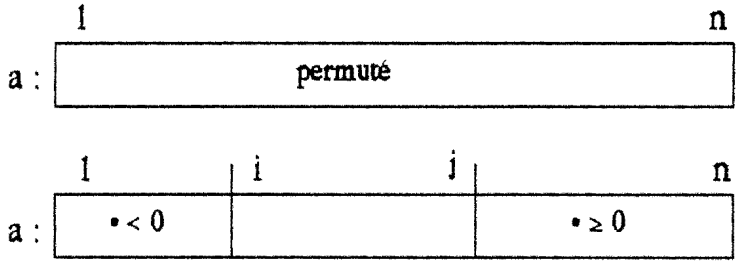
- i >= 1

- i <= j+1
- j <= n

Condition sur le segment a3

- • (a[j+1:n]) >= 0

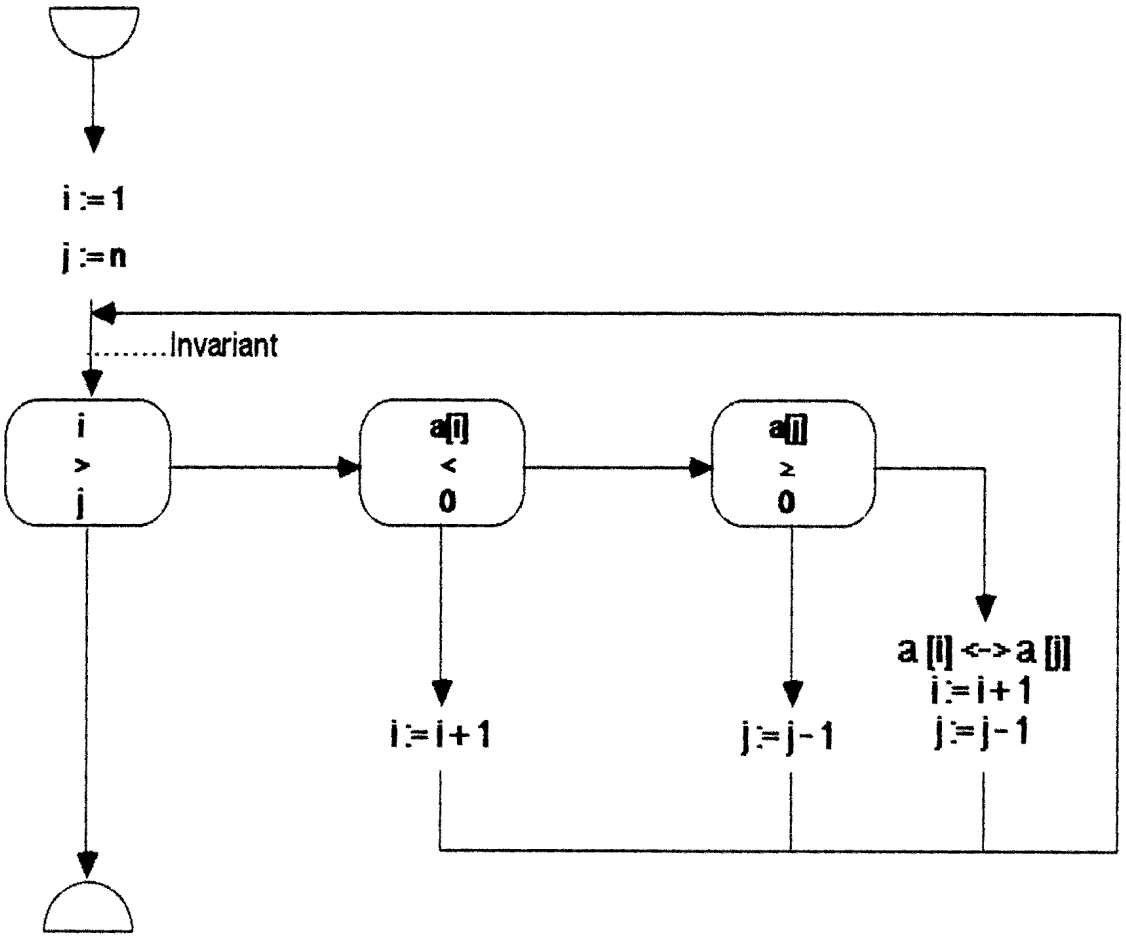
Représentation des schémas



7.3. Organigramme

a : tableau de type entier

i, j : variables entières



7.4. Commentaires

Dans l'expression de la postcondition et de la situation générale de ce programme, nous rencontrons l'utilisation de la fonction prédéfinie **permuté**, **a permuté** signifie que le tableau a contient les mêmes éléments qu'avant l'exécution de la première instruction du programme, mais que ceux-ci ont été permutés au sein même du tableau.

La postcondition et la situation générale montrent des cas où l'état d'un tableau peut être décrit à l'aide de plusieurs schémas.

Remarquons que dans ces deux situations, l'état du tableau a aurait pu être décrit par un seul schéma.

Soit l'expression de la postcondition :

elle aurait pu être exprimée par un seul schéma divisé en deux segments a1 et a2, avec les conditions suivantes :

Condition sur le segment a1

- • (a[1:i]) < 0
- **permuté** (a [1:n])

Condition sur le segment a2

- • (a[i+1:n]) >= 0

8. Illustration 7 : utilisation de la fonction examiné

8.1. Spécifications

Programme de calcul de la valeur d'un polynôme :

Etant donné un tableau 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 du tableau a) évalué en la valeur de x.

Ce qui de manière plus formelle et selon la méthode de l'invariant peut être exprimé comme suit :

8.1.1. Précondition

- $n \geq 0$
- a [0:n] initialisé
- x initialisée

8.1.2. Postcondition

- x et a inchangés
- $y = \sum_{i=0}^{n} a[i]\, x^{n-i}$

8.1.3. Invariant

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

8.2. Expression des conditions à l'aide du langage de conditions

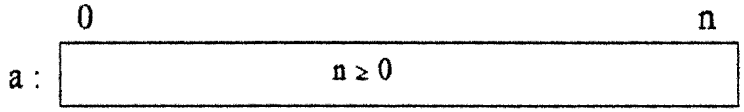
8.2.1. La précondition

La description de la situation comporte un schéma décrivant l'état du tableau a.

Ce schéma est caractérisé par la condition

- n >= 0

Représentation du schéma



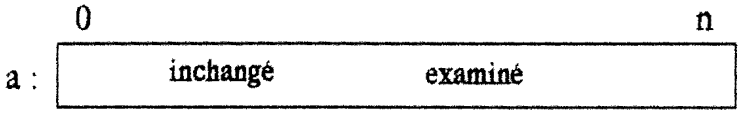
8.2.2. La postcondition

La description de la situation comporte un schéma décrivant l'état du tableau a.

Ce schéma est caractérisé par les conditions

- **inchangé** (a [0:n])
- a **examiné**

Représentation du schéma



8.2.3. La situation générale

La description de la situation comporte deux schémas décrivant l'état du tableau a :

- le schéma 1,
- le schéma 2, constitué :
  - du segment a1 = a [1:i], et
  - du segment a2 = a[i+1:n].

Condition sur le schéma 1

- **inchangé** (a [1:n])

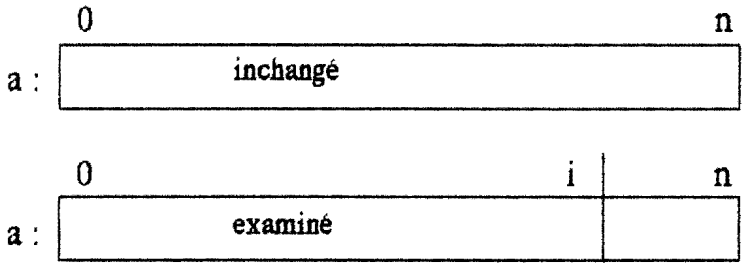
Conditions sur le segment a1

- **examiné**

(conditions sur la borne i)

- i >= 0
- i <= n

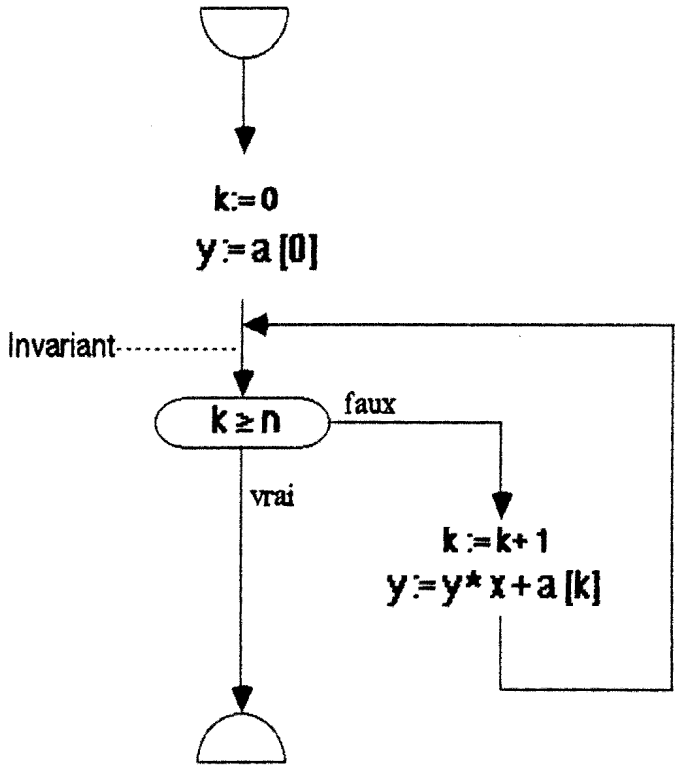
Représentation des schémas



8.3. Organigramme

a : tableau de type entier

k, x, y : variables entières



8.4. Commentaires

Le langage de conditions n'offre pas un opérateur itératif permettant d'exprimer les conditions :

- $y = a[0]\,x^n + a[1]\,x^{n-1} + \ldots + a[n-1]\,x + a[n]$,
- $y = a[0]\,x^i + a[1]\,x^{i-1} + \ldots + a[i-1]\,x + a[i]$.

En effet, l'opérateur itératif Σ n'est défini que pour exprimer la somme des éléments d'un segment.

Il faudrait donc envisager de définir une fonctions itérative (prédéfinie) pouvant prendre des expressions comme arguments.

Cependant, étant dans l'impossibilité d'expliquer clairement la valeur de la variable y à un moment donné de l'exécution, il est possible d'exprimer qu'une partie du tableau a déjà été examinée grâce à la fonction prédéfinie **examiné**.

9. Illustration 8 : notion de borne fixe et de borne mobile

Cet exemple est développé dans l'annexe 1.

9.1. Spécifications

Programme de recherche dans un tableau du segment de somme maximale :

Soit a, un tableau initialisé de n éléments de type entier. Ces éléments, notés a [i] avec $1 \leq i \leq n$, sont positifs, négatifs ou nuls. Déterminer les indices d et f tels que la somme des valeurs des éléments du segment a [d:f] soit maximale.

(la somme des valeurs des éléments d'un segment vide est nulle)

Ce qui de manière plus formelle et selon la méthode de l'invariant peut être exprimé comme suit :

9.1.1. Précondition

- a [1:n] initialisé

9.1.2. Postcondition

- a inchangé
- $1 \leq d \leq f+1 \leq n+1$
- $\forall\ i,j : 1 \leq i \leq j+1 \leq n+1 : \sum_{k=i}^{j} a[k] \leq \sum_{k=d}^{f} a[k]$

9.1.3. Invariant

- $1 \leq k \leq i+1 \leq n+1$
- $1 \leq d \leq f+1 \leq i+1$
- $\sum_{p=d}^{f} a[p] = m$
- $\sum_{p=k}^{i} a[p] = s$
- $\forall\ d',f' : 1 \leq d' \leq f'+1 \leq n+1 : \sum_{p=d'}^{f} a[k] \leq m$
- $\forall\ k' : 1 \leq k' \leq i+1 : \sum_{p=k'}^{i} a[k] \leq s$

9.2. Expression des conditions à l'aide du langage de conditions

9.2.1. La précondition

9.2.2. La postcondition

La description de la situation comporte trois schémas décrivant l'état du tableau a :

- le schéma 1,
- le schéma 2, divisé en trois segments : a1, a2, et a3,
- le schéma 3, divisé en trois segments : a1', a2', et a3'.

Condition sur le schéma 1

- **inchangé** (a [1:n])

Conditions portant sur le segment a2

- Σ = m

(conditions sur la borne d)

- d >= 0
- d <= n

(conditions sur la borne f)

- f >= 0
- f <= n

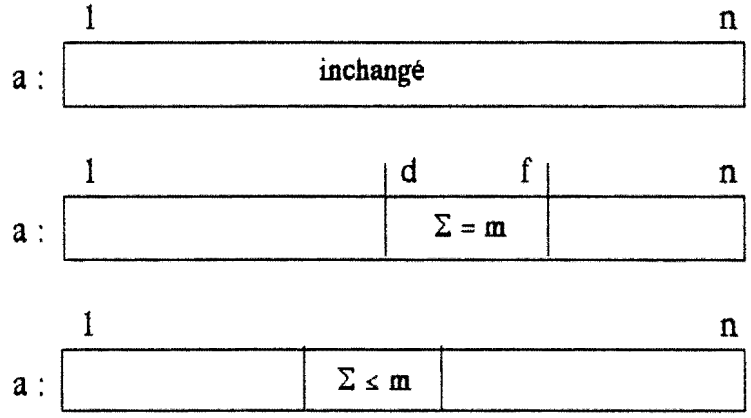
(condition sur les borne d et f)

- d <= f

Condition sur le segment a2'

- Σ <= m

Représentation des schémas



9.2.3. La situation générale

La description de la situation comporte quatre schémas décrivant l'état du tableau a :

- le schéma 1, divisé en quatre segments : $a1_1$, $a2_1$, $a3_1$, et $a4_1$,
- le schéma 2, divisé en quatre segments : $a1_2$, $a2_2$, $a3_2$ et $a4_2$,
- le schéma 3, divisé en trois segments : $a1_3$, $a2_3$, et $a3_3$,
- le schéma 4, divisé en trois segments : $a1_4$, $a2_4$, et $a3_4$.

Conditions sur le segment $a2_1$

- $\Sigma = m$

(conditions sur la borne d et sur la borne f)

- d >= 1
- d <= f+1
- f <= i+1

Conditions sur le segment $a3_1$

(conditions sur la borne i)

- i >= 1
- i <= n

Condition sur le segment $a2_2$

- $\Sigma <= m$

Conditions sur le segment $a3_2$

(conditions sur la borne i)

- $i >= 1$
- $i <= n$

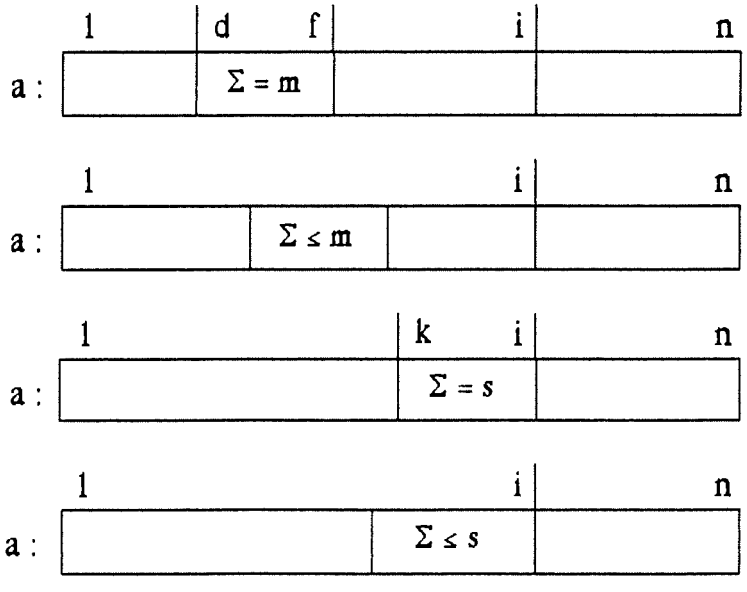
Conditions sur le segment $a2_3$

- $\Sigma = s$
- $k >= 1$
- $k <= i+1$
- $i <= n$

Conditions portant sur le segment $a2_4$

- $\Sigma <= s$
- $i >= 1$
- $i <= n$

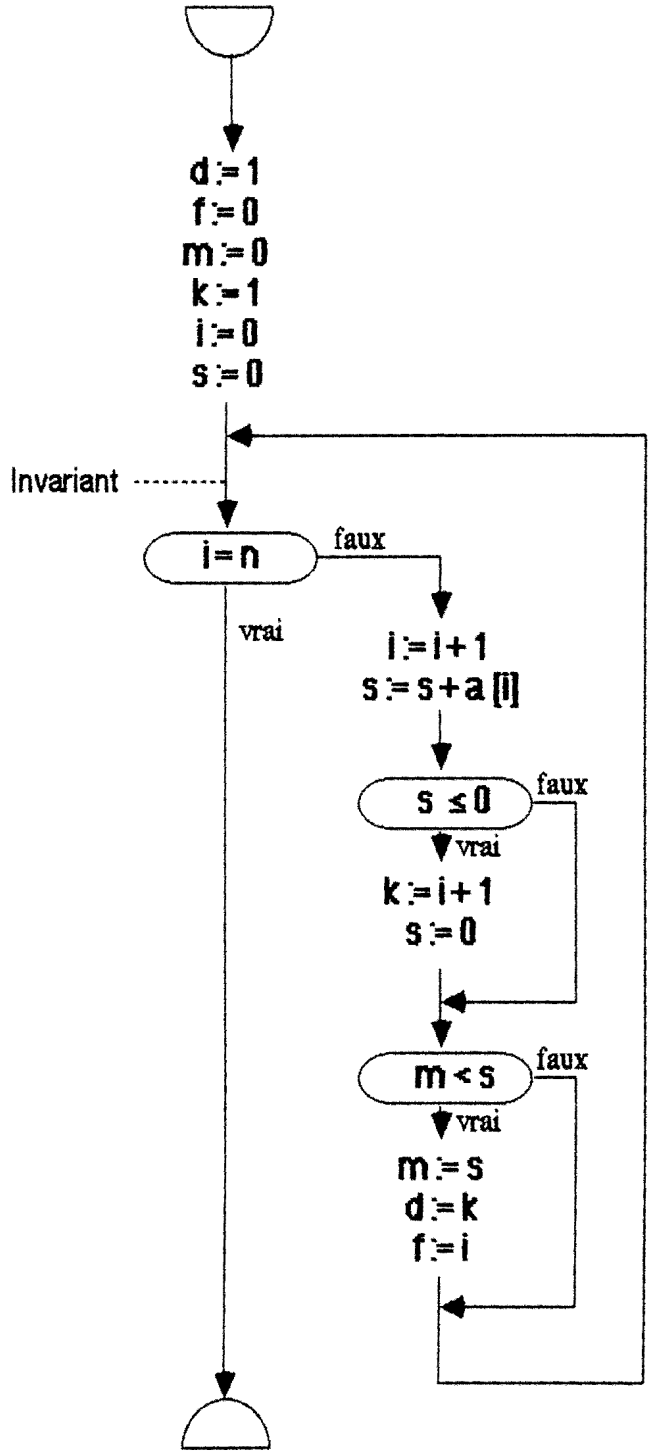
Représentation des schémas



9.3. Organigramme

a : tableau de type entier

d, f, k, i, s, m : variables entières



9.4. Commentaires

Dans cet exemple, nous trouvons plusieurs utilisations de la fonction prédéfinie **somme**. Dans la présentation, cette fonction est exprimée à l'aide du symbole Σ , mais elle peut être exprimée de différentes manières (cfr la syntaxe précise dans la définition du langage à l'annexe 2 : dans le point "5. Fonctions prédéfinies du langage"). Cette fonction désigne la somme des éléments du segment sur lequel elle porte.

Exemple

Soit un segment S = [2, 8, 3, 6],

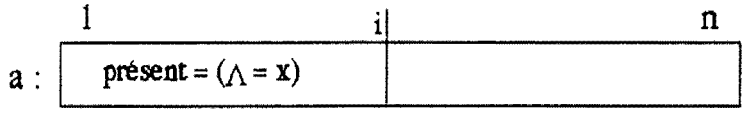
Σ (S) = 19,

dans ce cas

- la condition Σ (S) $\leq$ 20 sera évaluée à vrai, et
- la condition Σ (S) $>$ 20 sera évaluée à faux.

Dans les expressions des situations des exemples précédents, tous les segments avaient des bornes fixes : la valeur d'une borne à un instant donné était, soit liée à la valeur d'une variable à cet instant, soit déterminée par une constante.

Exemple



Le tableau a est divisé en deux segments :

- le segment a [1:i], et
- le segment a [i+1:n].

La valeur de la borne inférieure du premier segment et celle de la borne supérieure du deuxième sont respectivement déterminées par les constantes 1 et n.

La valeur de la borne supérieure bs_1 du premier segment et celle de la borne inférieure bi_2 du deuxième sont liées à la valeur de la variable i de la manière suivante :

- bs_1 = i,
- bi_2 = i+1.

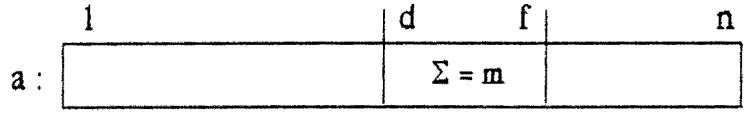
Une borne dont la valeur n'est ni déterminée par une constante ni liée à la valeur d'un variable est appelée borne mobile.

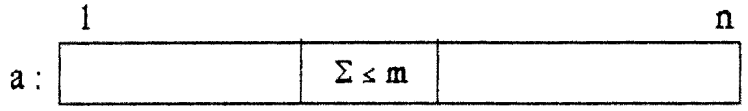
Un segment ayant au moins une borne mobile est appelé segment mobile.

Soit dans la postcondition, l'expression de la condition

$$- \forall i,j : 1 \le i \le j+1 \le n+1 : \sum_{k=i}^{j} a[k] \le \sum_{k=d}^{f} a[k]$$

est traduite à l'aide des deux schémas :





Le premier de ces schémas est constitué de trois segments dont les bornes sont fixes.

Le deuxième schéma est constitué de trois segments mobiles :

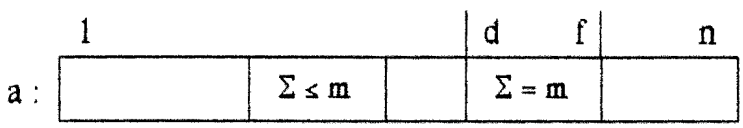
- la borne supérieure du premier segment est mobile,
- les deux bornes du deuxième sont mobiles, et
- la borne inférieure du troisième est mobile.

Clairement (comme suggéré par le schéma) cela signifie que les bornes du deuxième segment peuvent varier entre 1 et n.

Afin de voir si la postcondition est vérifiée, le système devra donc effectuer l'évaluation de toutes les situations particulières (cfr la définition de situation particulière et la vérification d'une situation dans la définition du langage à l'annexe 2) de la situation.

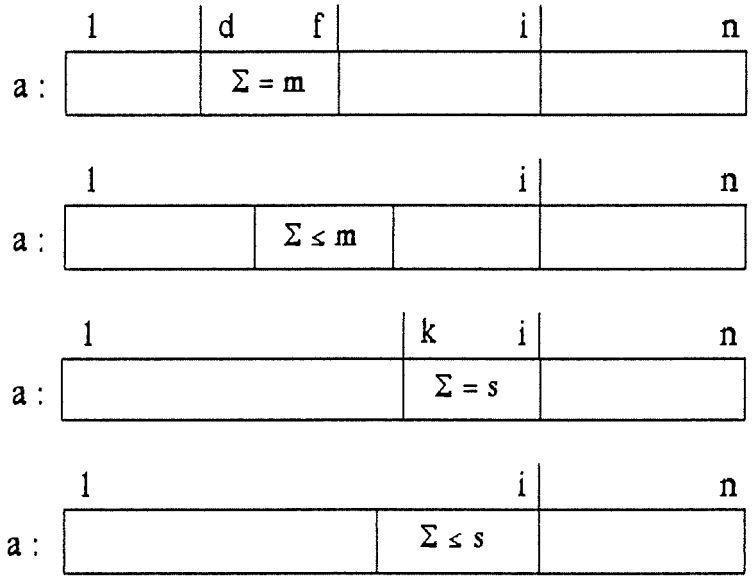
Intuitivement, cela signifie que la condition $\Sigma \le m$ doit être vérifiée pour toutes les valeurs possibles des bornes mobiles du schéma.

Remarquons que le système n'attache pas la même signification au schéma suivant :



De ce schéma, le système déduira que le deuxième segment, soit a2, est situé avant le segment a [d:f] (cette interprétation est due aux relations existant entre segments contigus) et donc dans ce cas, les valeurs des bornes de ce segment a2 varieraient entre 1 et d.

De manière similaire, l'expression de la situation générale de ce programme est traduite de la façon suivante :



Lors de l'évaluation de la situation générale, le système devra donc vérifier que toutes les conditions de tous les segments sont vérifiées pour toute valeur de borne possible.

Notamment, étant donné le deuxième schéma, il devra vérifier que la condition $\Sigma \leq m$ est évaluée à vrai, avec la fonction Σ prenant comme argument le segment a [s:j], et ce $\forall$ s, j : $1 \leq s \leq i$, $1 \leq j \leq i$.

Chapitre 3

Scénario d'exploitation du système

1. Introduction

Un scénario standard est décrit dans l'annexe 1.

Ce scénario comporte les étapes suivantes :

- la construction de la spécification du programme,
- la description de la situation générale,
- le choix des instructions,
- la vérification de la terminaison,
- l'écriture du programme.

Un premier scénario d'exploitation va être décrit dans ce mémoire.

La description de ce scénario est effectuée en deux parties :

-1- présentation de concepts généraux,

-2- description et présentation des différents écrans qui seront utilisés lors de l'exécution du système.

Ensuite, le scénario proposé sera comparé au scénario standard décrit dans l'annexe 1.

2. Concepts de base

Les concepts présentés dans cette section sont les suivants :

- la notion de menu d'un écran,
- la notion de champ,
- la notion de liste,
- la notion de cadre.

Tous les écrans qui apparaîtront lors de l'exécution du système, comportent une partie menu dans laquelle l'utilisateur est amené à faire un choix.

La notion de champ est définie afin de réaliser la saisie de données courtes et précises, telles les noms de programmes, les noms de situations ....

Le concept de liste a notamment été développé afin de permettre l'introduction des objets manipulés dans un programme, mais également pour permettre d'assembler des éléments ayant des propriétés communes. Exemples :

- liste des programmes enregistrés,
- liste des schémas décrivant une situation,
- liste des segments formant un tableau,
- ...

Le concept de cadre correspond en fait à un outil d'édition : il a été crée afin de permettre de saisir les différentes conditions portant sur les situations, les schémas, les segments, les bornes ... ainsi que la saisie des instructions constituant un programme.

2.1. Menu

Un écran présente un menu. Lors de l'affichage de l'écran, l'utilisateur doit choisir une option dans le menu proposé. Afin d'effectuer son choix, l'utilisateur dispose de deux procédés :

Procédé 1 :

L'utilisateur introduit le numéro de l'option choisie, suivi de la touche "RETURN".

Si le numéro introduit ne correspond pas à une option du menu offert, alors le système affichera un message d'erreur et l'utilisateur devra recommencer sa sélection.

Procédé 2 :

L'utilisateur positionne le curseur sur l'option choisie à l'aide des flèches et enfonce la touche "RETURN".

2.2. Introduction d'un champ

Un champ est une zone de l'écran caractérisée par :

- sa longueur,
- sa position à l'écran,
- son contenu à un instant donné, qui est une chaîne de caractères.

L'utilisateur peut effectuer une seule opération sur un champ, appelée "l'introduction d'un champ".

L'introduction d'un champ est définie comme suit :

- si cette donnée n'a pas encore été introduite au moins une fois,

  alors **introduction d'une nouvelle donnée** dans ce champ,

- si cette donnée a déjà été introduite au moins une fois,

  alors **modification de la donnée** existant dans ce champ.

L'utilisateur indique la fin d'introduction d'un champ en enfonçant la touche "RETURN".

Lors de l'introduction d'un champ, l'utilisateur dispose des fonctions suivantes :

- déplacement du curseur d'un caractère vers la gauche ou d'un caractère vers la droite en enfonçant la touche "←" ou la touche "→",

- positionnement du curseur en début ou en fin de la chaîne de caractères représentant la donnée que l'on introduit en enfonçant la touche "HOME" ou la touche "END",
- effacement du caractère sous le curseur en enfonçant la touche "DEL",
- effacement du caractère à gauche du curseur en enfonçant la touche "BACKSPACE",
- effacement complet de la donnée en enfonçant simultanément les touches "CTRL" et "D" (la touche "CTRL" désignant la touche contrôle du clavier).

2.3. Listes et manipulations de listes

Il faut distinguer deux types de listes :

- listes d'identifiants,
- listes de données.

Afin de mieux comprendre la distinction à effectuer entre ces deux types, voyons quelles sont leurs utilisations respectives.

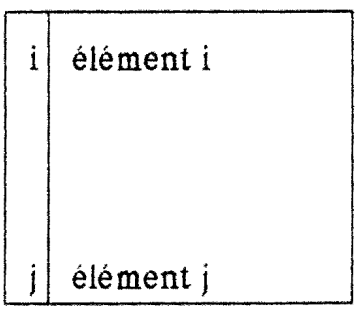
Une liste de données est principalement utilisée pour enregistrer la liste des objets du programme. L'introduction de ceux-ci ne nécessite pas la définition d'un écran pour la saisie d'un objet, et ils peuvent tous apparaître sur le même écran.

Une liste d'identifiants est utilisée pour gérer un ensemble d'éléments ayant des propriétés communes :

- liste des programmes enregistrés,
- liste des schémas décrivant une situation,
- liste des segments formant un tableau,
- ...

L'introduction d'un élément de telles listes est effectuée à l'aide d'un écran particulier.

L'affichage de la liste s'effectue comme suit :



Un élément sera toujours affiché sur une seule ligne.

Déplacements du curseur : positionnement sur un élément

- déplacement d'un élément vers le bas ou d'un élément vers le haut en enfonçant la touche " ↓ " ou la touche " ↑ ".
- déplacement d'un certain nombre d'éléments vers le bas ou vers le haut en enfonçant la touche "PAGE DOWN" ou la touche "PAGE UP".

Manipulations d'une liste

L'utilisateur peut effectuer trois sortes d'opérations sur une liste :

- **Introduction d'un élément**

  L'utilisateur enfonce simultanément les touches "CTRL" et "I", ensuite,

  s'il s'agit d'une liste de données alors l'utilisateur introduit les différents champs de l'élément,

  s'il s'agit d'une liste d'identifiants alors passage à l'écran correspondant à l'introduction d'un élément de cette liste.

- **Modification d'un élément**

  L'utilisateur positionne le curseur sur l'élément de la liste qu'il désire modifier et enfonce la touche "RETURN",

  s'il s'agit d'une liste de données alors l'utilisateur peut modifier les différents champs de l'élément,

  s'il s'agit d'une liste d'identifiants alors passage à l'écran correspondant à l'introduction d'un élément de cette liste.

- **Suppression d'un élément**

  L'utilisateur positionne le curseur sur l'élément de la liste qu'il désire supprimer et enfonce la touche "DEL".

Fin de manipulation d'une liste

Lorsque l'utilisateur a effectué toutes les manipulations qu'il désirait sur la liste, il enfonce simultanément les touches "CTRL" et "Q", ce qui provoque le retour au menu proposé par l'écran.

Remarque

Dans une liste de données, lorsque l'utilisateur a introduit un élément, le système effectue les vérifications adéquates concernant cet élément. Si le système décèle une erreur, l'utilisateur reçoit un message indiquant le type d'erreur rencontrée et l'invitant à réintroduire cet élément.

2.4. Cadre et manipulations de cadre

Désignons par **fiche** un tableau de caractères caractérisé par :

- un nombre de lignes,
- un nombre de colonnes (=nombre de caractères dans une ligne).

A une **fiche** est attachée une fenêtre que nous appelerons **cadre**. Le cadre représente la partie visible à l'écran de la fiche à laquelle il est attaché.

Un **cadre** est caractérisé par :

- un nombre de lignes (ou sa hauteur),
- un nombre de colonnes (ou sa longueur),
- son positionnement sur la fiche (exemple : coordonnées du premier élément du cadre dans la fiche),
- son positionnement à l'écran.

Manipulations d'un cadre :

Déplacements du curseur

- Déplacement du curseur d'un caractère vers la gauche ou d'un caractère vers la droite en enfonçant la touche "←" ou la touche "→".

  (Remarque : si le curseur se trouvait sur le premier ou le dernier caractère de la ligne du cadre alors déplacement du cadre sur la fiche d'une colonne vers la gauche ou d'une colonne vers la droite)

- Déplacement du curseur d'une ligne vers le haut ou vers le bas en enfonçant la touche " ↑ " ou la touche " ↓ ".

  (Remarque : si le curseur se trouvait sur la première ou la dernière ligne du cadre alors déplacement du cadre sur la fiche d'une ligne vers le haut ou d'une ligne vers le bas)

- Déplacement du curseur d'un certain nombre de lignes vers le haut ou vers le bas en enfonçant la touche "PAGE UP" ou la touche "PAGE DOWN".

  (Remarque : ajustement du positionnement du cadre sur la fiche)

- Positionnement du curseur en début de ligne ou en fin de ligne de la fiche en enfonçant la touche "HOME" ou la touche "END".

  (Remarque : ajustement du positionnement du cadre sur la fiche si nécessaire)

Remarque :

Insistons sur la signification du "déplacement du cadre sur la fiche" : du point de vue de l'apparition à l'écran, le cadre reste affiché au même endroit à l'écran, et dès lors l'utilisateur a l'impression que ce dernier est fixe et que la fiche se déplace derrière celui-ci.

Fonctions d'édition

- Effacement du caractère sous le curseur en enfonçant la touche "DEL".
- Effacement du caractère à gauche du curseur en enfonçant la touche "BACKSPACE".
- Effacement de la ligne contenant le curseur en enfonçant simultanément les touches "CTRL" et "D".
- Insertion d'une ligne à la position du curseur en enfonçant la touche "INS".
- Effacement de la ligne à partir de la position du curseur en enfonçant simultanément les touches "CTRL" et "L".
- Fin d'édition de la fiche en enfonçant simultanément les touches "CTRL" et "Z".

2.5. L'option édition dans un menu

Lorsque l'utilisateur choisit cette option dans le menu proposé, il peut introduire les différents champs et manipuler les différents cadres qui sont repris dans l'écran proposé.

Pour ce faire, l'utilisateur doit d'abord sélectionner un champ ou un cadre en positionnant le curseur sur le champ ou le cadre désiré et enfoncer la touche "RETURN".

Lorsque l'utilisateur a marqué la fin d'introduction d'un champ ou la fin d'édition d'un cadre, il peut sélectionner un autre champ ou un autre cadre.

Lorsque l'utilisateur a terminé l'introduction des différents champs et l'édition des différents cadres, il enfonce simultanément les touches "CTRL" et "Q", ce qui a pour effet de renvoyer au menu proposé par cet écran.

2.6. L'option analyse dans un menu

Lorsque l'utilisateur sélectionne cette option dans un menu, le système effectue l'analyse adéquate des différentes données introduites grâce à cet écran.

S'il n'y a pas d'erreur le système retourne au menu proposé par l'écran, sinon les erreurs sont communiquées une à une à l'utilisateur et ensuite retour au menu proposé par l'écran.

2.7. Conventions pour la présentation des écrans

Dans la présentation de la constitution des écrans :

«...» désigne l'emplacement d'un champ,

‹*...*› désigne l'emplacement d'un cadre,

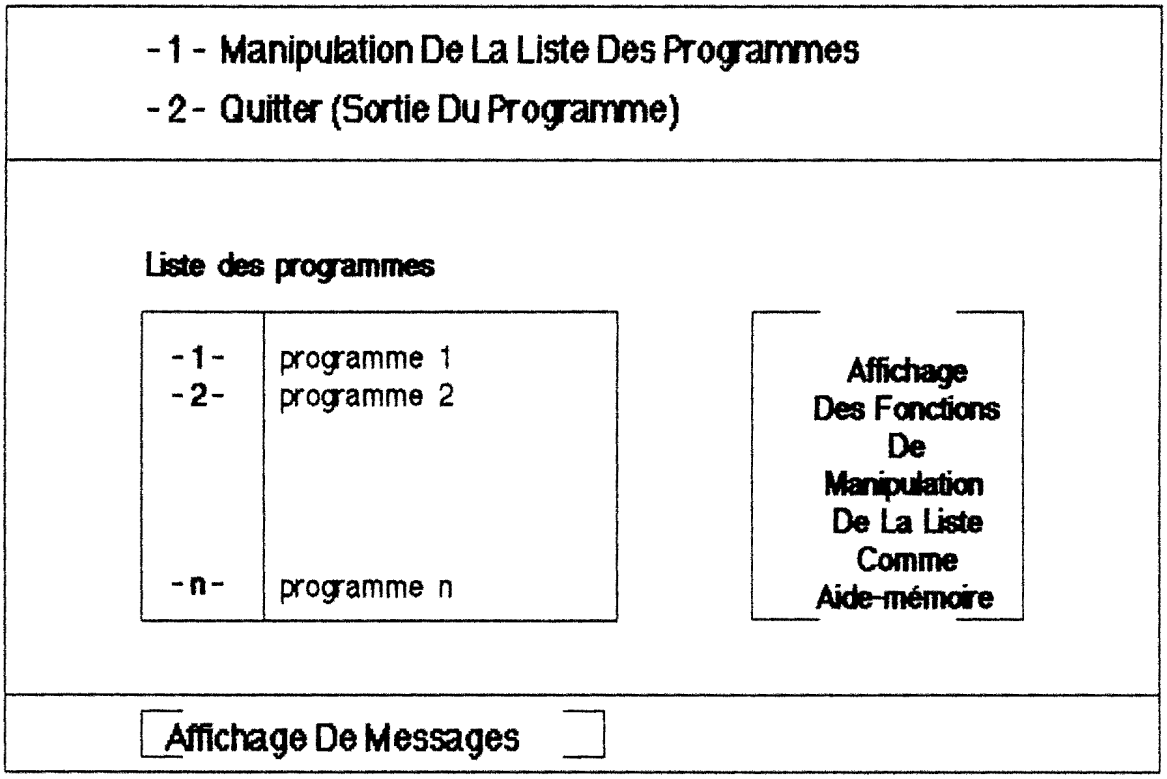
[AFFICHAGE DE MESSAGES] désigne l'emplacement réservé à l'affichage de messages par le système,

[AFFICHAGE DE FONCTIONS DE ...] désigne l'emplacement de l'affichage d'un aide mémoire pour l'utilisateur.

3. Définition des écrans

3.1. ECRAN 1 : menu principal

3.1.1. Constitution de l'écran



3.1.2. Description du menu

Si l'utilisateur choisit :

l'option 1 : - manipulation de la liste des programmes -

Il peut manipuler cette liste.

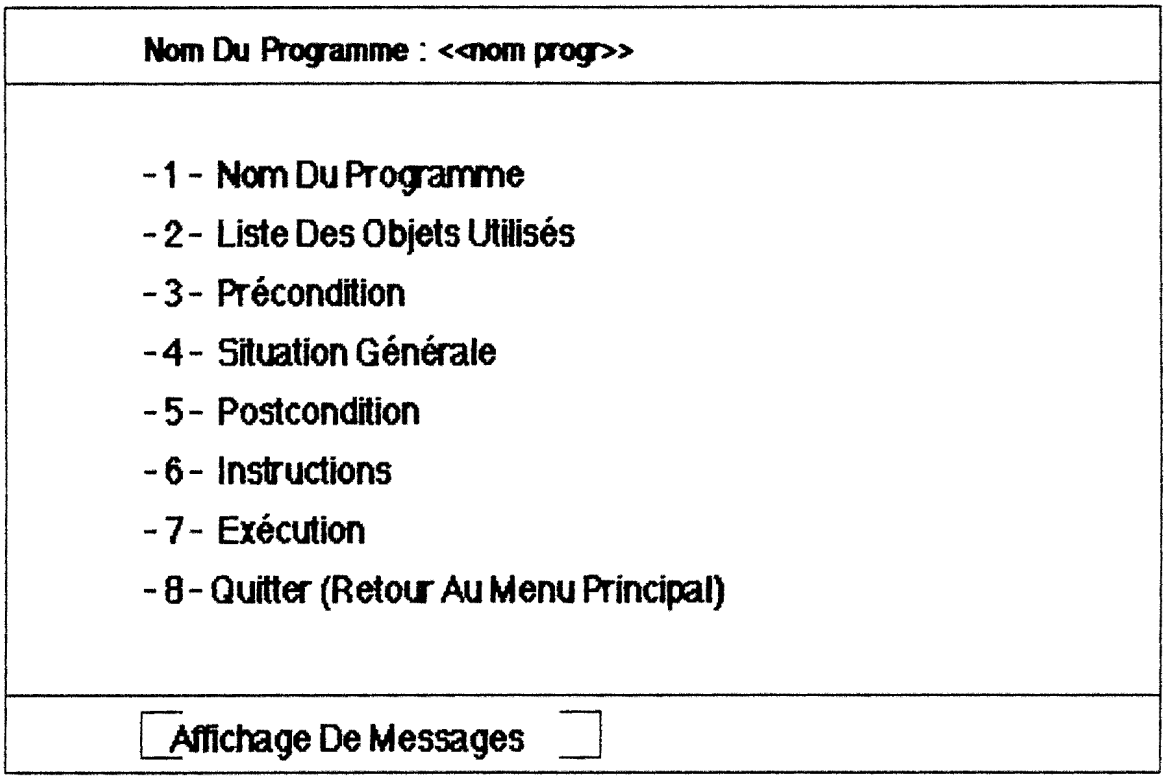
La liste des programmes est une liste d'identifiants : les fonctions d'introduction et de modification d'un élément provoquent le passage à l'écran "**introduction d'un programme**".

l'option 2 : - quitter (sortie) -

Sortie du programme.

3.2. ECRAN 2 : introduction d'un programme

3.2.1. Constitution de l'écran



3.2.2. Description du menu

Si l'utilisateur choisit :

l'option 1 : - nom du programme -

Il peut introduire le champ «nom prog».

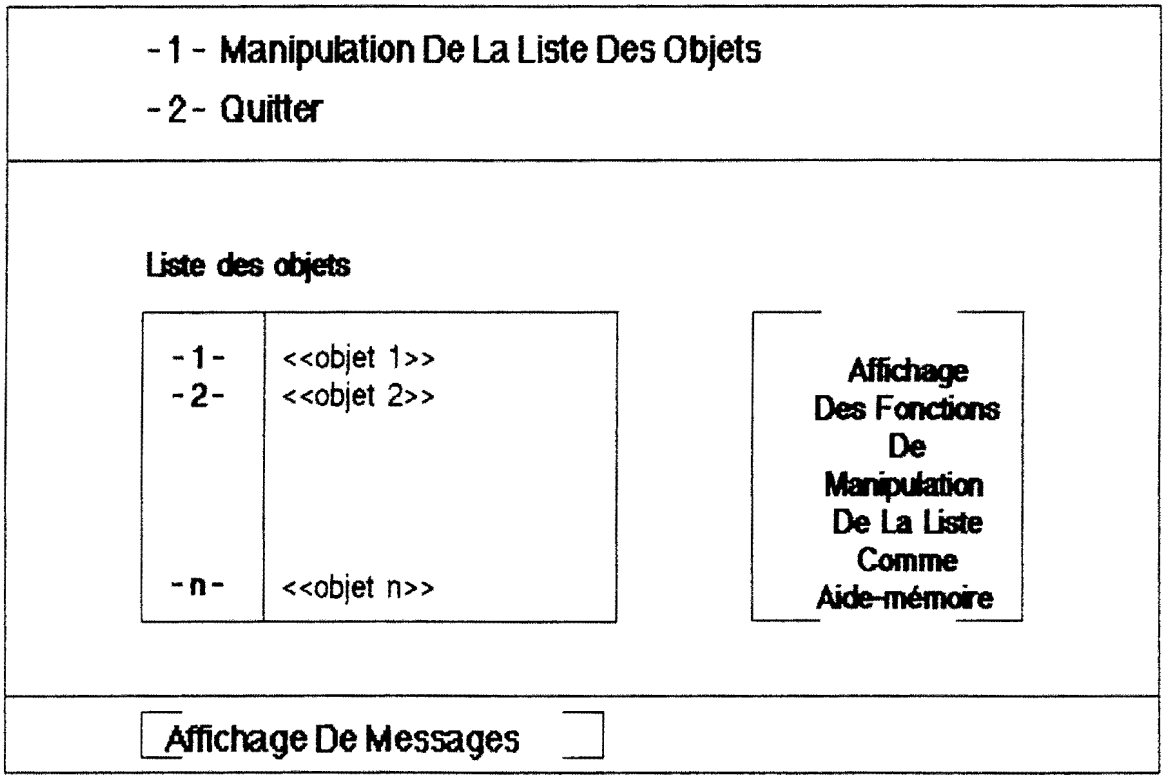
(Remarque : le système vérifiera l'unicité du nom du programme dans la liste des noms de programme)

En ce qui concerne les autres options, selon celle choisie, l'utilisateur verra apparaître l'écran correspondant à cette option selon la table de correspondance suivante :

Option choisie	Passage à l'écran :
Nom Du Programme	(reste à cet écran)
Liste Des Objets Utilisés	Introduction de la liste des objets utilisés
Précondition	Introduction d'une situation
Situation Générale	Introduction d'une situation
Postcondition	Introduction d'une situation
Instructions	Introduction des instructions
Exécution	Exécution du programme
Quitter (Retour Au Menu Principal)	Menu principal

3.3. ECRAN 3 : introduction de la liste des objets utilisés

3.3.1. Constitution de l'écran



3.3.2. Description du menu

Si l'utilisateur choisit :

l'option 1 : - manipulation de la liste des objets -

Il peut manipuler cette liste.

Dans ce cas, la liste est une liste de données, à savoir, une liste des descriptions des objets manipulés par le programme : constantes, variables et tableaux.

Un objet comporte quatre champs :

champ 1 : mention indiquant s'il s'agit d'un objet principal ou d'un objet auxiliaire.

champ 2 : type de l'objet (constante, variable ou tableau).

champ 3 : si l'objet est de type variable ou de type tableau, ce champ est réservé pour le nom de l'objet,

si l'objet est de type constante, ce champ est réservé pour la valeur de la constante.

champ 4 : type de valeur de l'objet.

Lorsque l'utilisateur a introduit ou modifié un objet, le système effectue la vérification suivante :

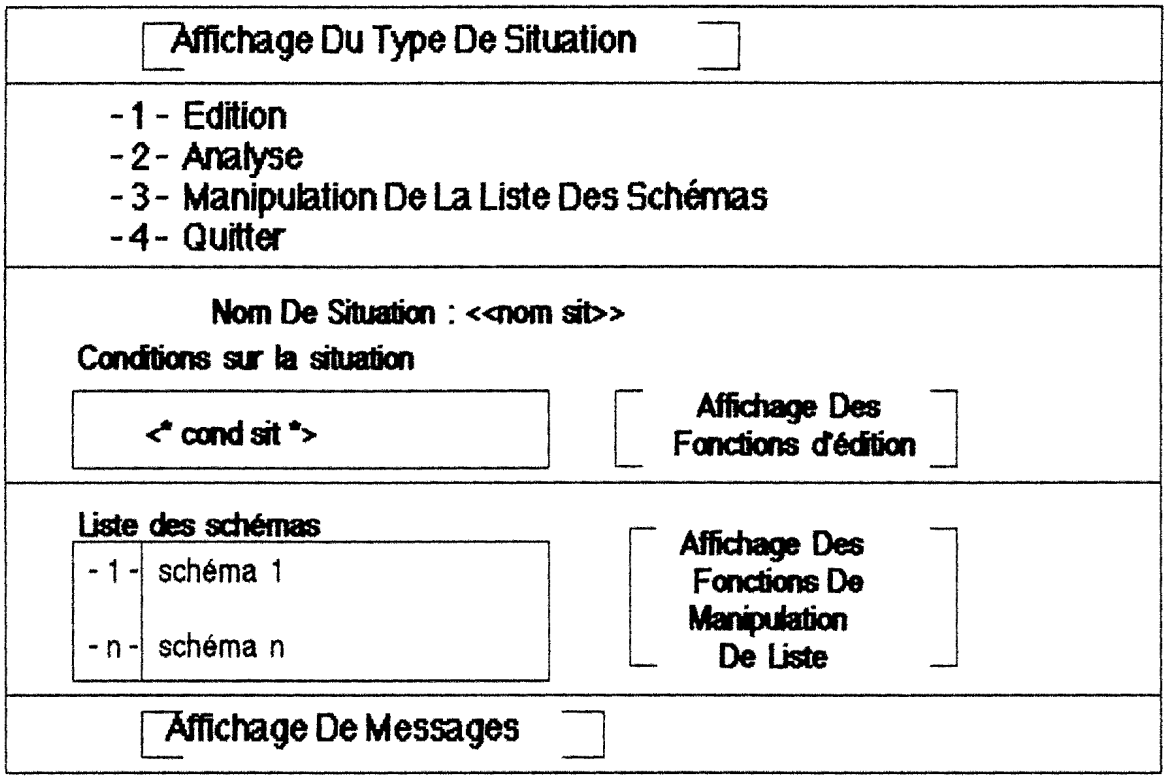
- unicité du nom de l'objet dans la liste des objets.

l'option 2 : - quitter -

Retour à l'écran "**introduction d'un programme**".

3.4. ECRAN 4 : introduction d'une situation

3.4.1. Constitution de l'écran



3.4.2. Description du menu

Si l'utilisateur choisit :

l'option 1 : - édition -

Il peut introduire le champ «nom sit» et éditer le cadre «*cond sit*».

(Remarque : l'utilisateur doit introduire une seule condition par ligne)

l'option 2 : - analyse -

- Le système effectue l'analyse syntaxique des différentes données introduites lors de l'édition,
- Le système vérifie l'appartenance des objets apparaissant dans les conditions à la liste des objets utilisés.

l'option 3 : - manipulation de la liste des schémas de la situation -

Il peut manipuler cette liste.

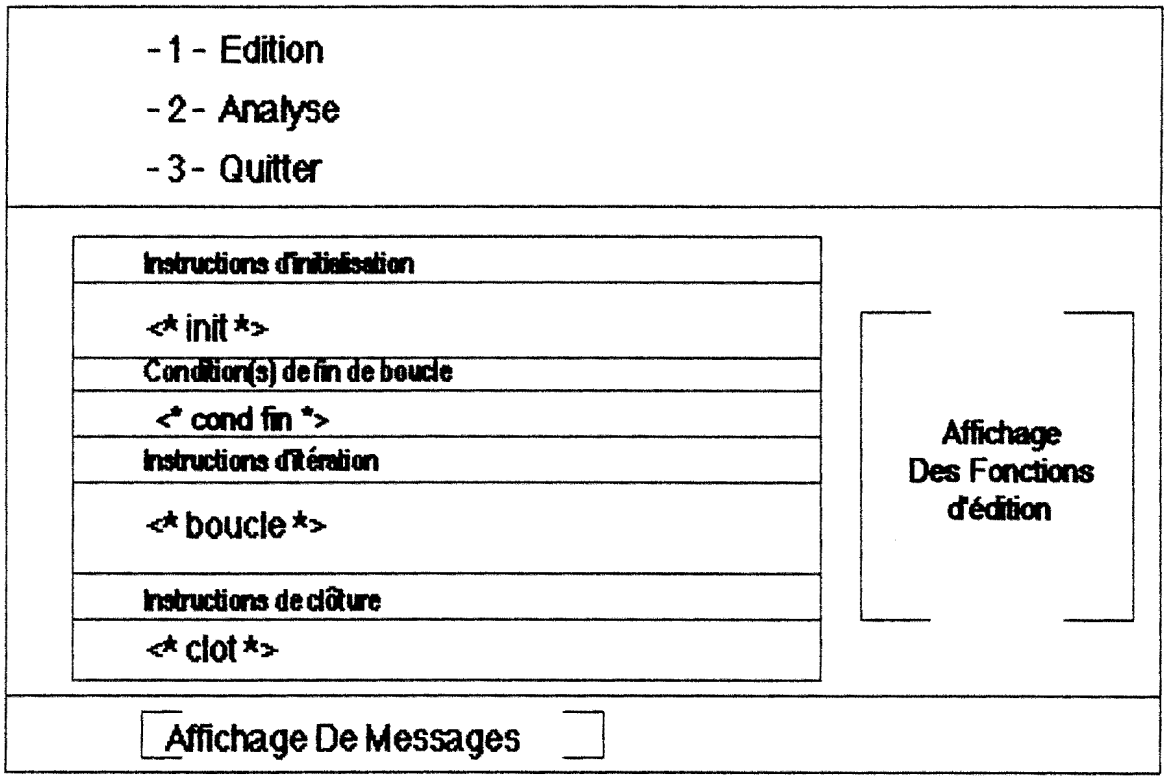
La liste des conditions sur la situation est une liste d'identifiants : les fonctions d'introduction ou de modification d'un élément provoquent le passage à l'écran "**introduction d'un schéma**".

l'option 4 : - quitter -

Retour à l'écran "**introduction d'un programme**".

3.5. ECRAN 5 : introduction des instructions

3.5.1. Constitution de l'écran



3.5.2. Description du menu

Si l'utilisateur choisit :

l'option 1 : - édition -

Il peut éditer les différents cadres présentés par cet écran.

(Remarque : l'utilisateur doit introduire une seule instruction par ligne)

l'option 2 : - analyse -

Le système effectue les vérifications suivantes :

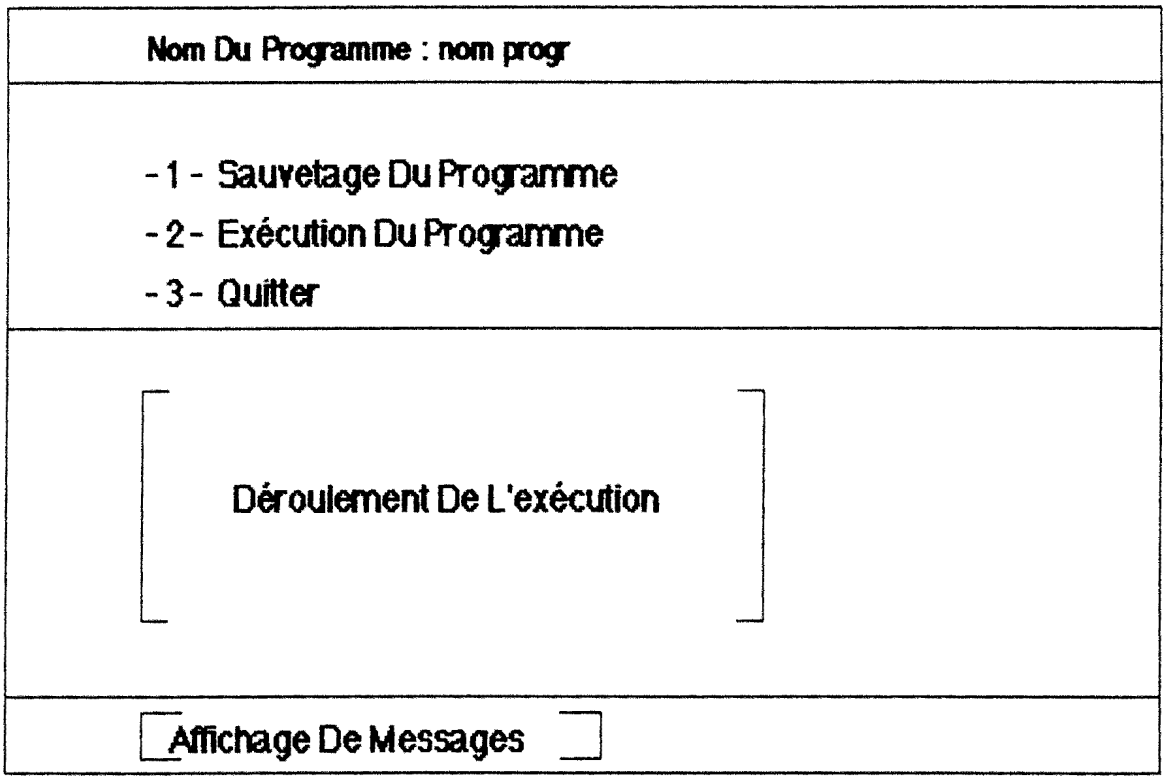
- analyse syntaxique des instructions,
- appartenance des objets apparaissant dans les instructions à la liste des objets utilisés.

l'option 3 : - quitter -

Retour à l'écran "**introduction d'un programme**"

3.6. ECRAN 6 : exécution du programme

3.6.1. Constitution de l'écran



3.6.2. Description du menu

Si l'utilisateur choisit :

l'option 1 : - sauvetage du programme -

Les différents éléments constituant le programme sont enregistrés sur fichier.

(Si le programme n'a pas encore été nommé l'utilisateur devra introduire le champ «nom prog»)

l'option 2 : - exécution du programme -

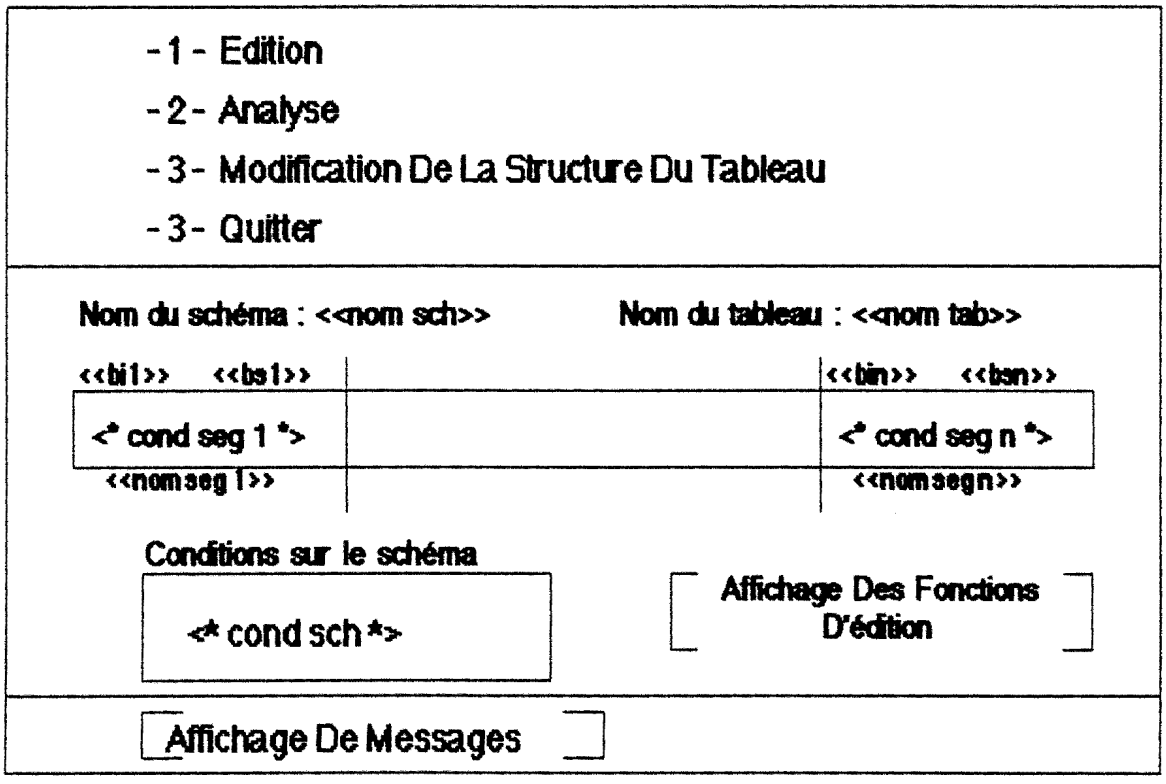
- L'utilisateur sera invité à introduire les différentes données du programme,
- le système interprétera le programme avec les données introduites,
- il affichera les résultats du programme après interprétation.

l'option 3 : - quitter -

Retour à l'écran "**introduction d'un programme**".

3.7. ECRAN 7 : introduction d'un schéma

3.7.1. Constitution de l'écran



3.7.2. Description du menu

Si l'utilisateur choisit :

l'option 1 : - édition -

Il peut éditer les différents champs et cadres présentés par cet écran.

(Remarque : dans un cadre, l'utilisateur doit introduire une seule condition par ligne)

l'option 2 : - analyse -

- Le système effectue l'analyse syntaxique des différentes données introduites lors de l'édition,
- il vérifie l'unicité du nom de schéma,
- il vérifie que le nom de tableau est bien repris dans la liste des objets utilisés,
- il vérifie l'appartenance des objets apparaissant dans les conditions à la liste des objets utilisés.

l'option 3 : - modification de la structure du tableau -

- L'utilisateur positionne le curseur sur un segment à l'aide des flèches,
- soit il enfonce la touche "RETURN", dans ce cas ajout d'un segment à gauche du segment sélectionné (c'est-à-dire, ajout d'un cadre pour les conditions de ce segment, d'un champ pour le nom du segment, et de champs pour les bornes),
- soit il enfonce la touche "DEL", dans ce cas le segment est supprimé (c'est-à-dire suppression des différents champs et du cadre représentant ce segment),
- soit il enfonce la touche "ESC", dans ce cas retour au menu offert par l'écran.

l'option 4 : - quitter -

Retour à l'écran "**introduction d'une situation**".

4. Conclusion

Le scénario proposé reprend les étapes :

-1- Construction de la spécification

(a) introduction des objets utilisés

(b) introduction de la précondition

(c) introduction de la postcondition

-2- Description de la situation générale

( ) introduction des objets utilisés

-3- Choix des instructions

On remarquera en effet, en consultant l'écran II du scénario "introduction d'un programme", que :

- l'étape -1- (a) correspond au choix -2- du menu proposé,
- l'étape -1- (b) correspond au choix -3- du menu proposé,
- l'étape -1- (c) correspond au choix -5- du menu proposé,
- l'étape -2- correspond au choix -4- du menu proposé,
- l'étape -3- correspond au choix -6- du menu proposé.

L'étape de "vérification de la terminaison" n'est pas prévue dans l'implémentation de ce premier scénario.

Notons que l'étape "écriture d'un programme peut être implémentée sans l'intervention de l'utilisateur. En effet, grâce aux différentes instructions effectuée par l'utilisateur et à la liste des objets que l'utilisateur a introduites, le système dispose de suffisamment de renseignements pour réaliser l'assemblage complet du programme.

Remarquons qu'une étape de reconstitution du programme pourrait être prévue afin de voir si l'étudiant sait réaliser l'assemblage du programme.

Le programme ainsi créé par le système comprendra :

- les instructions du programme introduites par l'utilisateur,
- les différentes situations attachées à ce programme, et qui ont été introduites par l'utilisateur.

Ainsi, lors de la demande de l'exécution du programme introduit, le système interprétera non seulement les instructions de ce programme, mais il vérifiera également que les situations du programme sont vérifiées à l'instant désiré (c'est-à-dire instant de l'exécution du programme correspondant à l'instant décrit par la situation).

En ce qui concerne le niveau de vérification du système, dans un premier temps, il sera peu élevé et ne comportera que les vérifications suivantes :

- vérifications syntaxiques des différentes conditions introduites,
- vérification de l'unicité des variables introduites dans la liste des objets,
- vérifications syntaxiques des instructions introduites,
- vérification de l'appartenance à la liste des objets du programme des variables relevées dans les conditions et les instructions introduites.

Rappellons également que ce premier scénario n'implémente l'introduction des situations que de manière graphique.

Chapitre 4

Une découpe en modules

1. Commentaires sur l'architecture

Les fonctionnalités du système et le scénario proposé au chapitre 3 nous permettent dès à présent de dégager une première architecture.

Insistons sur le fait que cette découpe en modules est provisoire et sera certainement modifiée en cours d'implémentation et lors des évolutions probables du premier scénario décrit.

Rappelons que le système doit effectuer des vérifications syntaxiques et des vérifications sémantiques (celles-ci étant limitées dans un premier temps) sur les différentes données introduites par l'utilisateur, de ce fait, un module implémentant les différentes primitives de vérifications s'impose : le **Module Analyse**.

Lorsque l'utilisateur aura introduit les objets manipulés par son programme, les descriptions des différentes situations et l'introduction des différentes séquences d'instructions composant son programme, le système devra "exécuter" ce programme. les différentes primitives d'exécution et d'évaluation nécessaires pour effectuer cette exécution seront implémentées dans le **Module Exécution Programme**.

La description du scénario est basée sur des concepts importants tels les concepts de liste et de cadre.

Nous définirons donc des modules implémentant ces structures et les primitives permettant de les manipuler : le **Module Gestion Liste** et le **Module Gestion Cadre**.

Au cours de l'exécution du système, les structures suivantes apparaissent : les programmes introduits par l'utilisateur, les objets manipulés par ces programmes, les situations décrivant ces programmes, les schémas décrivant l'état des tableaux et les segments constituant un schéma.

Ces structures et les primitives de manipulation correspondantes seront implémentées dans le **Module Gestion Structures**.

En outre, ce module contiendra :

- différentes procédures d'affichage : affichages d'écran, affichages de messages, affichage des résultats de l'exécution d'un programme introduit, ...,
- différentes procédures de saisie : saisie des valeurs des variables du programme introduit avant de l'exécuter, saisie du choix effectué dans un menu, saisie du choix d'un cadre ou d'un champ à éditer, ...

La gestion du curseur apparaît dans les modules gestion liste, gestion cadre et gestion structure, c'est pourquoi les structures et les primitives permettant de gérer la position du curseur sont implémentées dans un module de niveau inférieur : le **Module Gestion Curseur**.

Ce module offre également les primitives suivantes :

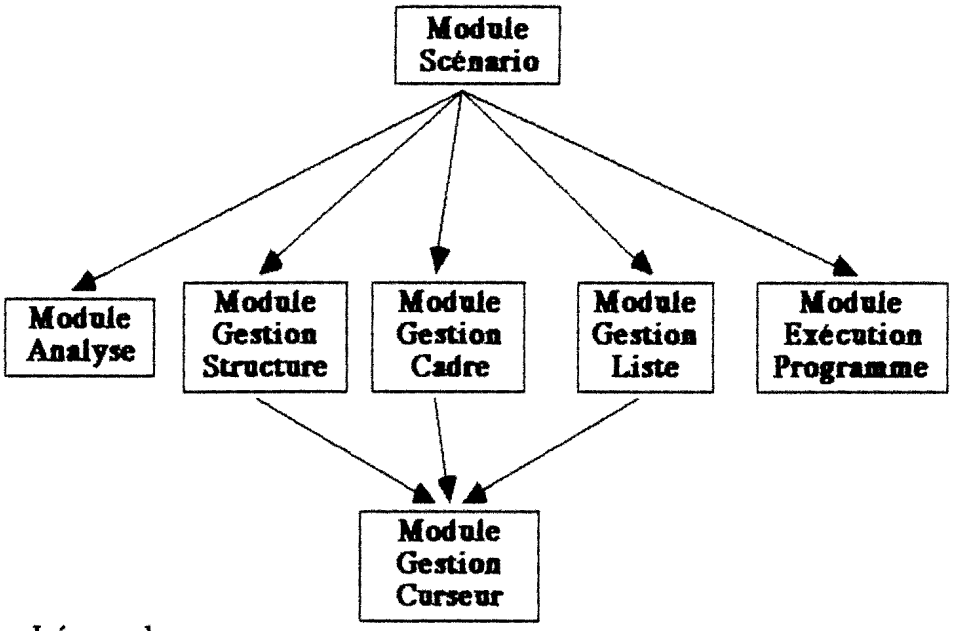
- primitive de saisie d'un caractère au clavier,
- primitive d'affichage d'un caractère à l'écran,
- primitive implémentant les fonctions d'édition,
- .....

Dès lors le nom "Gestion Curseur" n'est plus très adéquat. Cependant, le choix d'un autre nom s'avère difficile, nous n'avons trouvé aucun autre nom résumant les différentes primitives reprises dans ce module.

Bien entendu, l'implémentation du scénario d'exploitation, ainsi que les primitives réalisant les différentes parties de celui-ci seront regroupées dans un module supérieur : le **Module Scénario**.

L'architecture de base proposée comportera donc les modules suivants :

- le module scénario,
- le module analyse,
- le module exécution programme,
- le module gestion structure,
- le module gestion cadre,
- le module gestion liste,
- le module gestion curseur.



Légende

module A ⟶ module B
signifie que le module A
peut utiliser les primitives offertes
par le module B

Le contenu des différents modules est affiné dans l'annexe 3. Il s'agit d'une première approche : les types à développer et les primitives à spécifier sont simplement énumérés.

Un développement plus approfondi de l'architecture a été effectué, mais n'étant pas terminé et n'ayant pu être testé, il n'est pas présenté dans ce mémoire.

Ainsi, les types suivants ont été analysés en vue de leur implémentation :

- le type liste dans le module gestion liste,
- le type champ dans le module gestion structure,
- un type de structure pour mémoriser l'environnement d'une situation dans le module scénario, et
- les différents types de structures constituant une condition devant être implémentées dans le module analyse.

Rappelons également que les modules gestion curseur et gestion cadre ont été implémentés (cfr le chapitre 5 et l'annexe 4).

Chapitre 5

Développement de deux modules :

Gestion écran

Gestion cadre

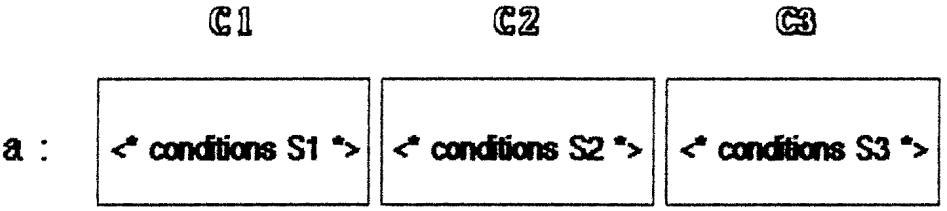
1. Introduction

Le concept de "cadre" est un concept important dans le développement du système. Il permet, par exemple, de réaliser la saisie des différentes conditions portant sur les différents segments constituant un tableau afin d'exprimer l'état de ce tableau à un instant donné de l'exécution du programme, en ayant une visualisation du tableau et des différentes conditions qui y sont relatives.

Exemple

Le tableau a comprenant les segments S1, S2, et S3

Il peut être vu comme trois cadres C1, C2 et C3, contigus et dont le contenu correspond aux différentes conditions attachées au segment représenté :



Etant donné que le concept de "cadre" est utilisé pour la réalisation d'un point important du logiciel -la saisie des conditions sur les segments en permettant une visualisation graphique des segments et des conditions qui y sont attachées- la partie programmation de ce mémoire consiste à développer ce type et partiellement les procédures qui y sont liées, c'est-à-dire implémenter le module gestion cadre.

Rappelons que le module gestion cadre utilise le module gestion curseur, par conséquent son implémentation suppose une implémentation de gestion curseur.

2. Gestion curseur

2.1. Présentation des types définis

Ce module offre les procédures et fonctions permettant de gérer la position du curseur à l'écran, ainsi que les effets à l'écran des différentes fonctions de déplacement et d'édition définies dans le scénario d'exploitation du système proposé au chapitre 3.

A cette fin sont définis les types :

- POSECRAN,
- DEPLACEMENTCURSEUR

(La définition exacte de ces types se trouve dans l'annexe 4)

Le type "**DEPLACEMENTCURSEUR**" permet de définir un espace de travail à l'écran de la forme d'un rectangle. Cet espace représente les dimensions de la partie utilisable de l'écran, c'est-à-dire, la portion de l'écran dans laquelle le curseur peut être positionné, ou encore espace de déplacement du curseur.

Le type "**POSECRAN**" permet de mémoriser la position courante du curseur à l'écran. Cette position est relative à l'espace de déplacement du curseur.

Exemple

Supposons une configuration supportant un écran composé de 25 lignes et 80 colonnes,

si le curseur ne peut se déplacer que dans un espace de 10 lignes sur 10 colonnes à l'écran, le premier emplacement de cet espace étant positionné à la dixième colonne de la dixième ligne de l'écran,

si le curseur est positionné à la dixième colonne de la dixième ligne de l'écran, alors la variable POSECRAN indiquera la position (1,1), c'est-à-dire la première position de l'espace de déplacement du curseur.

2.2. Commentaires sur les procédures offertes

Les procédures peuvent se classer dans les catégories suivantes :

- Catégorie 1 : lecture d'un caractère
- Catégorie 2 : émission d'un son
- Catégorie 3 : gestion des déplacements
- Catégorie 4 : gestion de l'édition

Lecture d'un caractère

Cette catégorie ne comporte qu'une seule procédure :

lirecar (var ch : integer; var fonction : boolean)

Cette procédure effectue la lecture d'un "caractère" introduit au clavier sans l'afficher à l'écran.

Elle est basée sur le fait qu'à chaque touche du clavier est associée une valeur numérique (également appelée "scan code"). Une valeur peut également être associée à une combinaison de touches enfoncées simultanément (par exemple : combinaison de la touche contrôle du clavier avec d'autres touches : "contrôle" et "A", ..., "contrôle" et "pagedown", ...)

Le clavier est composé de deux types de touches :

- les touches dites "non-fonctions",
- les touches dites "fonctions".

(De même une combinaison de touches est de type "non-fonction" ou de type "fonction")

L'introduction d'un touche (ou combinaison de touches) a pour effet de renvoyer une valeur numérique (appelé "scan code") :

- si ce code est différent de zéro, alors la touche (ou combinaison de touche) enfoncée est de type "non-fonction",
- s'il vaut zéro, alors la touche (ou combinaison de touches) enfoncée est de type "fonction" et une seconde valeur est renvoyée (appelée "extended scan code").

La procédure de lecture d'un caractère est basée sur cette distinction.

Lorsqu'une touche (ou combinaison de touches) "non-fonction" est enfoncée, la procédure **lirecar** renvoie les résultats suivants :

- fonction prend la valeur faux,
- ch contient le code numérique associé à la touche (ou combinaison de touches) "non-fonction" enfoncée.

Lorsqu'une touche (ou combinaison de touches) "fonction" est enfoncée, la procédure **lirecar** donne les résultats suivants :

- fonction vaut vrai
- ch contient le second code ("extended scan code") renvoyé par la touche (ou combinaison de touches) "fonction".

Emission d'un son

Cette catégorie présente une seule procédure qui émet un son d'une intensité donnée pendant un laps de temps déterminé.

Cette procédure peut notamment être utilisée pour signaler à l'utilisateur que l'opération qu'il désirait effectuer ne lui est pas permise ou qu'elle n'est pas réalisable, et qu'elle ne produit donc aucun effet.

Exemples d'utilisations :

-1- Dans le module gestion cadre : dans les déplacements, lorsque l'utilisateur désire effectuer un déplacement qui aurait pour effet de positionner le curseur en dehors des limites de la fiche, cette opération n'a aucun effet et l'utilisateur est averti par l'émission d'un son.

-2- Dans une simple gestion d'écran cette procédure pourrait être employée pour signaler à l'utilisateur que le déplacement qu'il désire effectuer n'est pas réalisable car cela positionnerait le curseur en dehors des limites de l'espace de déplacement du curseur autorisé.

Gestion des déplacements

Cette catégorie est constituée d'une procédure implémentant les déplacements du curseur définis dans le scénario d'exploitation du système proposé au chapitre 3.

Etant donné une valeur numérique (qui doit nécessairement correspondre au second code renvoyé par une touche "fonction"), la position du curseur est modifiée comme suit :

- Pour une valeur numérique associée à la touche "←" (ou "→") :

  Le curseur se déplacera d'un caractère vers la gauche (ou vers la droite).

- Pour une valeur numérique associée à la touche "↑" (ou "↓") :

  Le curseur se déplacera d'une ligne vers le haut (ou vers le bas).

Dans le cas où le déplacement demandé déplacerait le curseur en dehors de l'espace de travail -espace de déplacement du curseur-, la position du curseur restera inchangée et la procédure signalera le fait que le déplacement demandé n'a pas été effectué -par le biais d'une variable booléenne-.

- Pour une valeur numérique associée à la touche "HOME" :

  Le curseur se positionnera au début de la ligne (c'est-à-dire à la première colonne de l'espace de déplacement) dans laquelle il se trouve.

- Pour une valeur numérique associée à la touche "END" :

  Le curseur se positionnera à la fin de la ligne (c'est-à-dire à la dernière colonne de l'espace de déplacement) dans laquelle il se trouve.

- Pour une valeur numérique associée à la touche "PAGEUP" :

  Le curseur se positionnera sur la première ligne de l'espace de déplacement, sans changer de colonne.

- Pour une valeur numérique associée à la touche "PAGEDOWN" :

  Le curseur se positionnera sur la dernière ligne de l'espace de déplacement, sans changer de colonne.

Remarque

L'émission d'un son lorsque l'utilisateur désire effectuer un déplacement qui positionnerait le curseur en dehors des limites de l'espace de déplacement du curseur à l'écran n'est pas réalisée au niveau de la gestion même des déplacements.

En effet, ces procédures peuvent être utilisées dans différents contextes, et le soin de décider de signaler qu'un déplacement n'est pas autorisé par l'émission d'un son est laissé à l'appréciation des appelants.

Exemples

-1- Si les procédures de déplacement sont utilisées pour implémenter la gestion des déplacements à l'écran lors de l'édition d'un cadre, alors les fonctions de déplacement sont destinées à effectuer des déplacements sur la fiche et un déplacement qui positionnerait le curseur en dehors de l'espace de déplacement à l'écran (qui correspond au cadre) est peut être réalisable sur la fiche !

-2- Si les procédures sont employées pour implémenter les déplacements sur les éléments d'une liste, à nouveau, un déplacement peut être réalisable sur la liste alors qu'il aurait pour effet de positionner le curseur en dehors des limites de l'espace de déplacement.

Gestion de l'édition

Cette catégorie réunit les différentes procédures implémentant les effets à l'écran des différentes fonctions d'édition définies dans le scénario d'exploitation proposé au chapitre 3.

Ainsi, les implémentations suivantes sont-elles disponibles :

- **Une procédure implémentant à l'écran la suppression d'une ligne :**

  la ligne contenant le curseur est effacée de l'écran,

  le curseur est positionné dans la ligne au-dessus de celle qui est supprimée,

  les lignes se trouvant en-dessous sont décalées d'une ligne vers le haut,

  et la dernière ligne est mise à blanc.

- **Une procédure implémentant à l'écran l'effacement de la fin d'une ligne.**

- **Une procédure implémentant à l'écran l'effet d'un "return" :**

  le curseur va se positionner à la première colonne de l'espace de déplacement de la ligne suivant celle contenant le curseur avant l'appel.

- **Une procédure implémentant à l'écran l'insertion d'une ligne (vide) :**

  le curseur va se positionner dans la nouvelle ligne insérée,

  et les lignes sous cette dernière sont décalées d'une ligne vers le bas.

- **Une procédure implémentant à l'écran l'effacement du caractère sous le curseur.**

- **Une procédure implémentant à l'écran l'effacement du caractère à gauche du curseur.**

- **Une procédure affichant un caractère à l'écran.**

2.3. Remarque

Il est possible de rassembler toutes les implémentations de déplacement dans une seule procédure dont le fonctionnement pourra s'adapter pour l'édition d'un cadre, pour l'introduction d'un champ, et même pour effectuer un choix dans une liste.

Exemples

-1- L'édition d'un cadre : cfr l'implémentation réalisée dans le module gestion cadre.

-2- L'introduction d'un champ :

un champ est caractérisé par sa longueur et sa position à l'écran,

un espace de déplacement du curseur à l'écran peut donc être considéré comme suit,

- sa longueur : la longueur du champ,
- sa hauteur : un,

dans ce cas, seules les touches "←", "→", "HOME" et "END" pourront avoir un effet.

-3- Le choix d'un élément ou d'un identifiant dans une liste :

dans ce cas, l'espace de déplacement du curseur peut être déterminé comme suit,

- sa longueur : un,
- sa hauteur : le nombre d'éléments ou d'identifiants affichables simultanément à l'écran,

dans ce cas, seules les touches "↑", "↓", "PAGEUP" et "PAGEDOWN" pourront avoir un effet.

Cependant, il est nécessaire d'implémenter les fonctions d'édition dans des procédures distinctes.

En effet, cela n'a aucun sens de vouloir insérer une ligne dans un champ et la signification de l'emploi de la touche "return" sera différente dans l'édition d'un cadre (cette touche provoquera un passage à la ligne) ou lors de l'introduction d'un champ (cette touche marquera la fin de l'introduction d'un champ).

3. Gestion cadre

3.1. Présentation des types définis

Les notions de cadre et de fiche ont déjà été introduites dans la présentation d'un scénario d'exploitation du système au chapitre 3.

Une fiche est une structure qui est destinée à contenir la représentation d'un ensemble de lignes (une ligne étant une suite de caractères de taille maximale fixée).

Cependant, il ne sera pas toujours possible de visualiser l'entièreté de cette fiche à l'écran, d'où la nécessité du concept de "cadre", qui est une structure réunissant les caractéristiques de représentation de la fiche à l'écran.

Ainsi, une fiche est scindée en deux parties :

- **Partie 1** : la représentation interne

  Elle est constituée par un pointeur vers la représentation de la première ligne de la fiche,

  Une ligne étant représentée par un record comprenant les champs suivants :

  - Champ 1 : le numéro de la ligne dans la fiche.
  - Champ 2 : un pointeur vers la représentation de la ligne précédente de la fiche.
  - Champ 3 : un pointeur vers la représentation de la ligne suivante de la fiche.
  - Champ 4 : la longueur effective de la ligne (c'est-à-dire le nombre de caractères de la ligne représentée).
  - Champ 5 : un tableau de caractères contenant les caractères de la ligne même.

- **Partie 2** : la représentation externe -le cadre-

  Elle regroupe les différentes caractéristiques attachées à la fiche :

  -1- La position du cadre sur la fiche qui est donnée par la position du premier élément du cadre dans la fiche :

  - un pointeur vers la représentation de la ligne dans laquelle se trouve ce caractère et,
  - un entier indiquant la position du caractère dans le tableau contenant la représentation de la ligne (ou numéro de la colonne dans la fiche).

-2- La position du cadre à l'écran : elle est donnée par les coordonnées du premier caractère du cadre à l'écran.

-3- La position du curseur sur la fiche : elle est donnée par :

- un pointeur vers la représentation de la ligne dans laquelle se trouve le caractère courant et,
- un entier indiquant la position de ce caractère dans le tableau contenant la représentation de la ligne (ou numéro de la colonne dans la fiche).

-4- Les dimensions du cadre :

- sa longueur (correspond au nombre de caractères dans une ligne),
- sa hauteur (correspond au nombre de lignes).

-5- L'état du cadre : le cadre peut être dans deux états :

- ouvert : modifiable (ou en cours d'édition),
- fermé : non-modifiable.

-6- Le mode d'écriture de la fiche.

Les caractéristiques -1-, -2- et -4- permettent :

- de déterminer quelle est la partie de la fiche à afficher,
- de déterminer l'endroit à l'écran où il faut afficher le cadre.

La caractéristique -3- permet de mémoriser la position du curseur sur la fiche :

- ce qui permet de le replacer correctement après l'affichage, lors par exemple d'éditions successives de ce cadre,
- cette caractéristique peut également être exploitée dans la suite du développement du logiciel afin de repositionner le curseur à l'endroit où une erreur a été détectée au cours de l'analyse du contenu du cadre.

Modes d'écriture d'une fiche

Lors de l'édition d'un cadre, l'utilisateur dispose de deux manières d'écrire dans une fiche :

Le mode réécriture

Dans ce cas, l'introduction d'un caractère à simplement pour effet de ranger dans la fiche le caractère introduit, à l'endroit indiqué par la position courante du curseur sur la fiche.

Le mode insertion

Dans ce cas, l'introduction d'un caractère à pour effet d'insérer un caractère à la position courante du curseur sur la fiche. Si cette position était déjà occupée par un caractère de la ligne, alors ce caractère et tous les caractères se trouvant à sa droite seront décalés d'une position vers la droite.

Remarque

Le langage Pascal employé pour l'implémentation ne permet pas de définir des tableaux de longueur variable.

De ce fait, il est nécessaire de fixer une taille maximale unique pour les lignes d'un cadre. Tous les cadres déclarés dans un même programme seront composés de lignes de cette taille unique !

3.2. Manuel de l'utilisateur (utilisation des concepts)

L'objectif de l'implémentation des modules gestion curseur et gestion cadre est de permettre la définition d'un cadre à l'écran (cfr les caractéristiques de cette définition dans le point 3.1 du présent chapitre et dans l'annexe 4), et de pouvoir réaliser "l'édition d'une fiche".

Etant donné le scénario d'exploitation proposé au chapitre 3 et les types définis dans le module gestion cadre, lors de l'édition d'un cadre, l'utilisateur disposera de deux types de fonctionnalités :

- les fonctions de déplacement,
- les fonctions d'édition proprement dites.

Remarque

L'édition d'une fiche signifie :

- afficher le contenu du cadre attaché à la fiche, et
- modifier la fiche en fonction des touches enfoncées par l'utilisateur.

Les fonctions de déplacement

Pour effectuer les déplacements sur la fiche, l'utilisateur devra employer les touches de déplacement du curseur du clavier.

(Rappel : voir la distinction entre touche "fonction" et touche "non-fonction" au point 2 de ce chapitre)

L'utilisation de la touche "←" (ou de la touche "→") :

- Elle entraîne un déplacement du curseur d'un caractère vers la gauche (ou vers la droite) sur la fiche.

- Si lorsque l'utilisateur a enfoncé la touche "←" ("→"), le curseur était positionné sur la première (dernière) colonne de la fiche,

  alors l'utilisateur sera averti par l'émission d'un son et l'opération demandée n'aura aucun effet -ni modification de la position du curseur sur la fiche, ni modification de la position du curseur à l'écran-,

- Si lorsque l'utilisateur a enfoncé la touche "←" ("→"), le curseur était positionné sur la première (dernière) colonne de l'espace de déplacement du curseur à l'écran, mais pas sur la première colonne (juste après le dernier caractère de la ligne) de la fiche,

  alors le cadre sera déplacé d'une colonne vers la gauche (droite) sur la fiche et sont contenu sera réaffiché -cela aura pour effet de modifier la position du curseur sur la fiche mais pas sa position à l'écran-.

L'utilisation de la touche "↑" (ou de la touche "↓") :

- Elle entraîne un déplacement du curseur d'une ligne vers le haut (ou vers le bas) sur la fiche.

- Si lorsque l'utilisateur a enfoncé la touche "↑" ("↓"), le curseur était positionné sur la première (dernière) ligne de la fiche,

  alors l'utilisateur sera averti par l'émission d'un son et l'opération demandée n'aura aucun effet -ni modification de la position du curseur sur la fiche, ni modification de la position du curseur à l'écran-,

- Si lorsque l'utilisateur a enfoncé la touche "↑" ("↓"), le curseur était positionné sur la première (dernière) ligne de l'espace de déplacement du curseur à l'écran, mais pas sur la première (dernière) ligne de la fiche,

  alors le cadre sera déplacé d'une ligne vers la haut (bas) sur la fiche et sont contenu sera réaffiché.

L'utilisation de la touche "PAGE UP" (ou de la touche "PAGE DOWN") :

- Elle entraîne un déplacement du curseur d'un nombre de lignes vers la haut (ou vers la bas) sur la fiche égal au nombre de lignes du cadre.

- Si lorsque l'utilisateur a enfoncé la touche "PAGEUP" ("PAGEDOWN"), le curseur était positionné sur la première (dernière) ligne de la fiche,

  alors l'utilisateur sera averti par l'émission d'un son et l'opération demandée n'aura aucun effet -ni modification de la position du curseur sur la fiche, ni modification de la position du curseur à l'écran-.

Ajustement lors de l'utilisation des touches "↑", "↓", "PAGE UP" et "PAGE DOWN" :

- Si cette utilisation entraîne un positionnement du curseur après plus d'un caractère après la fin de la ligne de la fiche, alors le curseur sera positionné juste après la fin de cette ligne (et si nécessaire, on ajustera la position du cadre sur la fiche de sorte que la position du curseur se trouve dans la partie visible de la fiche, c'est-à-dire le cadre).

L'utilisation de la touche "HOME" (ou "END") :

- Elle entraîne le positionnement du curseur au début (à la fin, c'est-à-dire juste après le dernier caractère) de la ligne de la fiche dans laquelle se trouve le curseur.

- Si lorsque l'utilisateur a enfoncé la touche "HOME" ("END"), le curseur était positionné sur le premier (juste après le dernier) caractère de la ligne de la fiche,

  alors l'utilisateur sera averti par l'émission d'un son et l'opération demandée n'aura aucun effet -ni modification de la position du curseur sur la fiche, ni modification de la position du curseur à l'écran-.

Les fonctions d'édition

Remarques :

Les fonctions d'édition agissent :

- sur la fiche,
- à l'écran.

Lorsqu'une fonction d'édition a pour effet de positionner le curseur en dehors du cadre, alors la position du cadre est ajustée de sorte que la position du curseur soit dans le cadre de la fiche.

La combinaison de la touche "contrôle" avec une touche quelconque X sera notée : "CTRL-X".

Lorsque l'utilisateur introduit une fiche, il dispose des fonctions suivantes :

les touches "fonctions" :

L'utilisation de la touche "DEL"

Elle permet d'effacer le caractère sous le curseur :

- à l'écran et dans la fiche, les caractères à droite du curseur seront décalés d'une place vers la gauche,
- la longueur de la ligne de la fiche sera diminuée de un.

Après l'opération, la position du curseur est inchangée.

L'utilisation de la touche "INS"

Elle permet l'insertion d'une ligne vide juste avant la ligne contenant le curseur :

- à l'écran, cette ligne sera matérialisée par une ligne d'espaces et les lignes sous la ligne contenant le curseur avant l'insertion seront décalées d'une ligne vers le bas,
- sur la fiche, cela aura pour effet d'insérer une ligne de longueur zéro.

Après cette opération, le curseur sera positionné sur la première colonne de la ligne insérée dans la fiche.

les touches "non-fonctions" :

L'utilisation de la combinaison "CTRL-I" (ou "CTRL-O")

Elle permet de déterminer le mode d'écriture de la fiche : mode d'insertion (ou mode de réécriture).

Le mode d'écriture peut être modifié à volonté lors de l'édition d'un cadre.

L'utilisation de la touche "BACKSPACE"

Elle permet d'effacer le caractère à gauche du curseur :

- à l'écran et dans la fiche, le caractère sous le curseur et les caractères à droite du curseur seront décalés d'une position vers la gauche,
- la longueur de la ligne de la fiche sera diminuée de un.

Après l'opération, la position du curseur sera décalée d'une colonne vers la gauche sur la fiche et à l'écran.

L'utilisation de la touche "RETURN"

L'effet de cette utilisation sera différente selon le mode d'écriture actif de la fiche :

- Si la fiche se trouve en mode insertion alors,

  ajout d'une ligne dans la fiche juste après celle contenant le curseur avant l'opération,

  cette nouvelle ligne contiendra le caractère qui était sous le curseur et ceux qui étaient à droite du curseur,

  si le curseur était positionné juste après la fin de la ligne avant l'opération, alors la longueur de la nouvelle ligne sera zéro, sinon elle sera égale au nombre de caractères qui étaient à droite du curseur plus un.

- Si la fiche se trouve en mode réécriture alors,

  le caractère qui était sous le curseur et ceux qui étaient à droite du curseur sont reportés à la ligne suivante (si cette ligne était de longueur non nulle alors les caractères seront remplacés .....).

L'utilisation de la combinaison "CTRL-D"

Elle aura pour effet de détruire la ligne contenant le curseur :

- la ligne sera retirée de la fiche,
- elle sera effacée de l'écran, et les lignes en-dessous seront décalées d'une ligne vers le haut.

L'utilisation de la combinaison "CTRL-L"

Elle aura pour effet de supprimer la fin de la ligne à partir de la position du curseur :

- la fin de la ligne sera effacée de l'écran,
- la longueur de la ligne de la fiche sera égale à la position du curseur sur la fiche moins un.

L'utilisation de la combinaison "CTRL-Z"

Elle aura pour effet de terminer l'édition du cadre.

Le module gestion cadre est implémenté de sorte à respecter les spécifications données ci-dessus.

Remarque

Le choix des touches pour réaliser les fonctions de déplacement et d'édition n'est pas le résultat d'un pur hasard.

Afin de faciliter l'utilisation de ce système, nous avons choisi de reprendre les choix effectués dans l'éditeur Turbo utilisé pour l'édition de programmes. Cet éditeur est supposé être utilisé par les étudiants pour lesquels le système d'aide à la programmation est élaboré.

De plus, l'utilisation des touches de déplacement du curseur pour les fonctions de déplacement se justifie pleinement car :

- ces touches sont fréquemment utilisées à cet effet dans la plupart des éditeurs, et
- ces touches portent généralement des marques indiquant les fonctions reprises.

3.3. Commentaires sur les procédures offertes

Les procédures peuvent se classer dans les catégories suivantes :

- Catégorie 1 : affichage d'un cadre
- Catégorie 2 : initialisation d'une fiche
- Catégorie 3 : modifications de la structure de la fiche
- Catégorie 4 : gestion des déplacements
- Catégorie 5 : gestion de l'édition
- Catégorie 6 : sauvetage et chargement d'une fiche
- Catégorie 7 : modifications des dimensions d'un cadre

Affichage d'un cadre

Cette catégorie offre les procédures permettant d'afficher le contenu d'un cadre.

L'affichage est conçu en deux étapes (deux procédures) :

- l'affichage d'une ligne,
- l'affichage du contenu d'un cadre (basé sur l'affichage des différentes lignes).

Remarque :

Attention, lorsqu'on affiche le contenu d'un cadre, il faut au préalable déterminer la fenêtre dans laquelle il va s'afficher (Justification : cette procédure est utilisée au cours de l'édition pour ajuster l'affichage du contenu d'un cadre).

Initialisation d'une fiche

Cette catégorie ne comporte qu'une seule primitive permettant d'initialiser une fiche.

Avant la première utilisation d'une fiche, il est nécessaire de procéder à son initialisation.

Les valeurs déterminant la position du cadre à l'écran et ses dimensions devront être fournies à la procédure (elles peuvent être différentes pour chaque fiche, et en principe modifiables par la suite).

Par définition, lors de l'initialisation la fiche est vide, et ne comporte donc aucune ligne d'où :

- le pointeur vers la représentation de la première ligne de la fiche,
- le pointeur vers la première ligne du cadre dans la fiche,
- le pointeur vers la ligne de la fiche contenant le curseur,

sont initialisés à vide.

L'état d'une fiche est initialisé à faux, c'est-à-dire que après l'initialisation, une fiche n'est pas dans un état éditable.

Le mode d'écriture initial d'une fiche est le mode réécriture (le choix du mode insertion eut été tout aussi valable).

Modifications de la structure de la fiche

Une modification de la structure même d'une fiche signifie agir sur les lignes de celle-ci.

Cette catégorie présente quatre procédures de modification de la structure :

- ajouter une ligne vide à la fin de la fiche,
- ajouter une ligne vide avant la ligne contenant le curseur,
- supprimer la ligne contenant le curseur,
- supprimer toutes les lignes de la structure.

Gestion des déplacements

La gestion des déplacements est implémentée par une seule procédure.

Les déplacements implémentés sont ceux présentés dans le scénario d'exploitation du système proposé au chapitre 3 et dans le manuel de l'utilisateur présenté dans ce chapitre.

Gestion de l'édition

Chaque fonction d'édition décrite dans le manuel de l'utilisateur est implémentée dans une procédure.

Ensuite, toutes ces fonctions d'édition sont regroupées dans une procédures.

Enfin, une procédure d'édition globale d'un fiche regroupe les fonctions de déplacement et d'édition.

Remarque concernant l'affichage d'un caractère introduit

Lorsqu'un caractère est introduit au clavier :

-1- il est introduit dans la fiche à l'emplacement adéquat,

-2- il est affiché à l'écran.

Sauvetage et chargement d'une fiche

Le sauvetage du contenu d'une fiche peut s'effectuer dans un fichier de type texte.

Il suffit de reprendre les lignes de la fiche et de les enregistrer dans le fichier en les séparant par une marque de fin de ligne.

Lors du chargement, il suffira de définir le contenu de chaque ligne en fonction des marques de fin de ligne.

Remarques :

Il n'est pas nécessaire de sauvegarder les caractéristiques de la représentation externe, celles-ci seront choisies lors du chargement de la fiche.

Il faudrait peut-être envisager de sauver plusieurs cadres dans un même fichier afin de sauver tous les cadres relatifs à un même programme dans un même fichier.

Modifications des dimensions d'un cadre

(pas encore implémentées)

3.4. Extensions possibles

Il est possible d'ajouter

- des fonctions de déplacement,
- des fonctions d'édition.

Exemples de fonctions de déplacement

- Déplacement d'un mot vers la gauche (vers la droite)
- Positionnement au début de la fiche
- Positionnement en fin de la fiche

Exemples de fonctions d'édition

- Effacement du mot à gauche du curseur
- Effacement du mot à droite du curseur
- Introduction de tabulations

Des primitives permettant de cacher l'implémentation des fiches au module utilisant ces fonctions devraient être crées.

Exemples

- Ajout d'une procédure donnant le contenu de la ième ligne de la fiche.
- Ajout d'une procédure donnant la première ligne de la fiche.
- Ajout d'une procédure donnant la ligne suivante de la fiche.

4. Remarque : les procédures de contour sont extérieures

4.1. Justification

Les concepts de fiche et de cadre sont des concepts généraux. leur utilisation peut être variée et selon les emplois le contour d'un cadre peut être différent.

4.2. Exemples

Exemple 1

Dans le scénario proposé au chapitre 3, les cadres sont utilisés pour :

-1- saisir les conditions sur un segment,

-2- saisir les conditions sur un schéma,

-3- saisir les conditions sur une situation.

Dans -1-, le contour d'un cadre ne peut guère consister qu'en un simple trait,

alors que dans -2- et -3-, un contour de cadre plus fourni peut être adopté, en le bordant par exemple d'un nom dans la partie supérieure et d'un aide mémoire reprenant les fonctions d'édition dans la partie inférieure.

Exemple 2

Dans ce même scénario, les cadres sont utilisés pour effectuer la saisie :

- De conditions,
- D'instructions.

Selon le contenu de la fiche à laquelle est attaché le cadre, les cadres peuvent avoir des contours différenciés par :

- des couleurs différentes,
- des traits différents (trait simple, trait double, ...).

Références

[Fisette85] D. Fisette,
Définition d'un langage de programmation permettant l'expression d'assertions,
mémoire de maîtrise, Namur, 1985.

[LSD80] B. Le Charlier,
Définition du langage LSD80,
Institut d'Informatique, Namur, Septembre 1980.

[PROG] Notes du cours de B. Le Charlier,
Preuves de programmes,
Institut d'Informatique, Namur.

[SYS88] M. Derroitte et B. Le Charlier,
Un système d'aide à l'enseignement d'une méthode de programmation,
Institut d'Informatique, Namur, Mai 1988.

Contribution à l'implémentation d'un langage graphique de description d'assertions

(Annexes)

Jeannine Rulkin

Mémoire réalisé sous la
direction du Professeur B. Le Charlier
en vue de l'obtention du titre de
Licencié et Maître en Informatique.

Namur Juin 1989.

Annexe 1

"Un système d'aide à l'enseignement d'une méthode de programmation"

Marc Derroitte et Baudouin Le Charlier

Un système d'aide à l'enseignement d'une méthode de programmation

Marc Derroitte et Baudouin Le Charlier

Institut d'Informatique, Facultés Universitaires de Namur
Rue Grandgagnage, 21, B-5000 NAMUR (Belgium)
Tel (32) (81) 22 90 65

(e-mail mdr@fun-cs.uucp)

Mai 1988

Résumé

Cet article présente les objectifs et les concepts d'un système d'aide à l'apprentissage d'une méthodologie de construction de programmes. Ce système est en cours de réalisation à l'Institut d'Informatique des Facultés Universitaires de Namur où cette méthodologie est enseignée. Il permettra d'aider les étudiants à comprendre et à appliquer cette méthode.

La méthodologie se base sur la construction par invariant, c'est-à-dire la description d'une situation générale et la construction de suites d'instructions en fonction de cette situation générale. Les difficultés que peuvent rencontrer ceux qui utilisent la méthode concernent essentiellement le caractère peu intuitif du raisonnement en termes de situation, ainsi que la rigueur nécessaire à la formalisation mathématique des situations. Le système devrait permettre de résoudre ces difficultés.

Le système d'aide à l'apprentissage de cette méthode est basé d'une part sur un langage graphique de description de situations dont le concept fondamental est la notion de "segment", et d'autre part sur un langage mathématique d'expression d'assertions, lui-même basé sur les notions de "suite" et d'"ensemble". Parmi les fonctionnalités du système, mentionnons notamment la vérification par démonstration formelle, la recherche de contre-exemples, des vérifications de cohérence entre les deux types d'expression de situations, des vérifications de cohérence et de complétude par rapport à la méthode.

Dans cet article, nous illustrons également différents scénarios d'utilisation du système, ainsi que le type de remarques qu'il peut offrir. Le problème du "segment de somme maximale" est résolu complètement, selon la méthode supportée par le système.

1. Introduction

Il est généralement admis, actuellement, qu'un bon enseignement de la programmation doit reposer sur une méthode rigoureuse de construction de programmes, telle que proposée par J. Arsac [Arsac80][Arsac83], E. Dijkstra [Dijkstra76], D. Gries [Gries81], ... Ces méthodes, bien que les plus intéressantes d'un point de vue théorique, induisent cependant certaines difficultés d'enseignement. En effet, elles exigent de la part des étudiants une formation mathématique adéquate et correspondent peu à une approche intuitive. D'autre part, il existe une tendance actuelle à utiliser l'ordinateur et ses possibilités propres pour compléter bon nombre d'enseignements.

Notre objectif est donc d'utiliser l'ordinateur pour illustrer et supporter ces méthodes d'enseignement de la programmation et pour tenter de les rendre plus accessibles.
Idéalement l'ordinateur devrait permettre de montrer aux étudiants résolvant des problèmes de programmation ce qu'ils ne pourraient percevoir directement, si ce n'est en disposant d'un enseignant prêt à les orienter et à les conseiller au fur et à mesure de l'application pratique d'une de ces méthodes. Le système que nous présentons permet de *comprendre* une telle méthode de construction de programmes par la mise en évidence des éventuelles erreurs de raisonnement.

Il existe des systèmes du genre basés sur la détection et la présentation des erreurs de programmation [Johnson86]. Pour notre part, nous voudrions aborder le problème différemment, en élaborant un système capable d'assister l'étudiant à tous les stades de l'élaboration d'un programme, depuis la spécification jusqu'au programme final.

Dans cet article, nous préciserons d'abord nos objectifs pédagogiques et la méthode enseignée illustrée par un exemple. Nous présenterons ensuite un aperçu du système, puis nous développerons plus en détail les concepts les plus originaux : un langage de description de schémas associé à un langage mathématique d'expression d'assertions. Enfin, nous exposerons les différents scénarios possibles d'utilisation du système en les commentant sur des exemples.

2. Méthode enseignée

2.1. Cadre d'application et options pédagogiques

Comme nous l'avons signalé ci-dessus, la méthode de construction de programmes sur laquelle se basera le système s'apparente aux méthodes classiques de construction de programmes (Arsac, Dijkstra, Gries, ... voir aussi [LeCharlier85], [Leroy78]).

Cependant, ces méthodes ont été adaptées en vue d'objectifs pédagogiques précis. Le système que nous allons présenter devra supporter un cours d'introduction à la programmation donné en première année d'université. Le cours poursuit deux objectifs principaux:

1. faire comprendre à travers l'étude d'un sous-ensemble réduit du langage Pascal [Wirth75] la nature formelle et mécanique des langages de programmation excluant toute imprécision et toute approximation,
2. apprendre une méthode rigoureuse de raisonnement permettant de construire un programme sur des bases solides.

Ces objectifs nous ont conduit à limiter de manière radicale la classe des problèmes traités. Chaque "programme" à construire sera un "morceau" de programme Pascal comportant au plus une boucle et manipulant uniquement des constantes, des variables simples et des tableaux à une

dimension. Il sera spécifié de manière précise à la fois pour pouvoir être construit de manière rigoureuse et pour pouvoir servir de primitive dans la construction d'un autre programme. Un programme "compliqué" sera donc systématiquement décomposé en un ensemble de programmes "simples".

C'est à dessein que sont éliminés les problèmes d'entrées/sorties car ceux-ci sont plus ardus à spécifier de manière simple et rigoureuse en raison de leur caractère dynamique. D'autre part, les problèmes traités sont purement "techniques" sans référence aux applications "réelles". C'est aussi une option délibérée car le traitement rigoureux de problèmes "pratiques" exige une modélisation du réel qui augmente considérablement les difficultés d'une approche rigoureuse. Dans ce cours d'introduction, nous voulons tenter de motiver les étudiants par la démarche elle-même, conduisant directement à un programme correct, pour des raisons claires et logiques. Notre optique risque peut-être de frustrer l'étudiant en le privant de l'impression d'avoir réalisé un programme utile qui "tourne", mais nous estimons lui donner de la sorte de meilleures armes pour s'attaquer ensuite à des problèmes plus gros et plus "réels" exigeant un aspect de modélisation plus important. Car il saura déjà ce que veut dire qu'un programme est correct et comment le construire.

Le système présenté dans cet article devrait aussi, nous l'espérons, motiver l'étudiant à un autre niveau en l'accompagnant dans sa démarche de construction et en l'aidant à valider ses raisonnements.

2.2. Démarche proprement dite

A partir d'un énoncé déjà précis, en français, et éventuellement d'une idée intuitive de solution, les étapes décrites ci-dessous doivent être suivies.

2.2.1. Spécification

La spécification du programme, sur base de l'énoncé proposé, comporte trois parties :

1. La *liste des objets utilisés* (constantes, variables, tableaux) classés en :
   - objets principaux qui constituent les données et résultats du programme,
   - objets auxiliaires qui constituent principalement les variables de travail.(1)
2. La *précondition* du programme (notée *Pré*) ou *situation initiale* qui est l'ensemble des conditions que doivent respecter les valeurs des objets principaux pour que l'exécution du programme ait un sens.
3. La *postcondition* du programme (notée *Post*) ou *situation finale* qui est la description de l'état des objets principaux après l'exécution du programme (si la précondition était respectée avant l'exécution).

---

(1) Il peut paraître contraire à la notion de spécification d'y inclure la liste des objets auxiliaires. Celle-ci est cependant nécessaire si le programme est utilisé comme sous-problème dans la construction d'un autre programme, pour vérifier qu'un objet auxiliaire du sous-problème n'est pas utilisé par le programme "principal". En effet, un sous-problème ne constituant pas une procédure mais seulement un morceau de programme, il n'y a pas de notion de variable locale. Cette partie de la spécification n'est pas remplie a priori, mais au cours de l'élaboration du programme.

2.2.2. Situation générale et Invariant

Pour les programmes ayant une forme itérative, la situation générale est une description schématique de l'ensemble des conditions vérifiées par les objets utilisés à chaque itération.

On appellera Invariant (noté I_A) une situation générale décrite dans un langage mathématique.

2.2.3. Choix des instructions

Sur base de la situation générale, plusieurs suites d'instructions doivent être décrites. Chacune de ces suites d'instructions doit vérifier une condition du type {P} S {Q}, où P et Q sont des situations (ou assertions), S la suite d'instructions considérée, et qui signifie :
"Si P est vrai avant l'exécution de S, alors l'exécution de S se terminera et les valeurs finales des objets du programme vérifieront Q".

Ces suites d'instructions sont :

- l'initialisation (notée *Init*) qui est l'ensemble des instructions nécessaires à la vérification de la condition :

  {Pré} Init {I_A}.

- la condition de terminaison (notée *B*) de l'itération et la clôture (notée *Clôt*) qui est l'ensemble des instructions nécessaires à la vérification de la condition :

  {I_A et B} Clôt {Post}.

- l'itération (notée *Iter*) qui est l'ensemble des instructions nécessaires à la vérification de la condition :

  {I_A et non(B)} Iter {I_A}.

2.2.4. Vérification de la terminaison

La vérification de la terminaison se fait par la définition d'une fonction entière et bornée des valeurs des variables, qui croît strictement à chaque exécution de la suite d'instructions Iter.

2.2.5. Ecriture du programme

Toutes les suites d'instructions doivent être assemblées de manière à former un seul programme (ou morceau de programme) respectant la syntaxe du langage Pascal.

2.3. Exemple

Un problème entrant dans la catégorie des problèmes "simples" que nous prenons en considération est celui de la recherche dans un tableau du *segment de somme maximale*. Ce problème classique a notamment été proposé par Jacques Arsac dans son article intitulé "La conception des programmes" publié dans Pour la science [Arsac84].
Voici une solution complète de ce problème selon notre démarche. Le lecteur qui ne serait pas familiarisé avec une méthode de construction de programmes de ce type veillera à étudier soigneusement cet exemple en vue d'une compréhension plus aisée de ceux qui seront traités dans la suite.

2.3.1. Enoncé du problème

Soit a, un tableau initialisé de n éléments de type entier. Ces éléments, notés a[i] avec $1 \leq i \leq n$, sont positifs, négatifs ou nuls. Ecrire un programme pour déterminer les indices d et f tels que la somme des valeurs des éléments du segment a[d..f] soit maximale.

Contrainte: tout élément du tableau ne peut être consulté qu'une seule fois.

Convention: la somme des valeurs des éléments d'un segment vide est nulle.

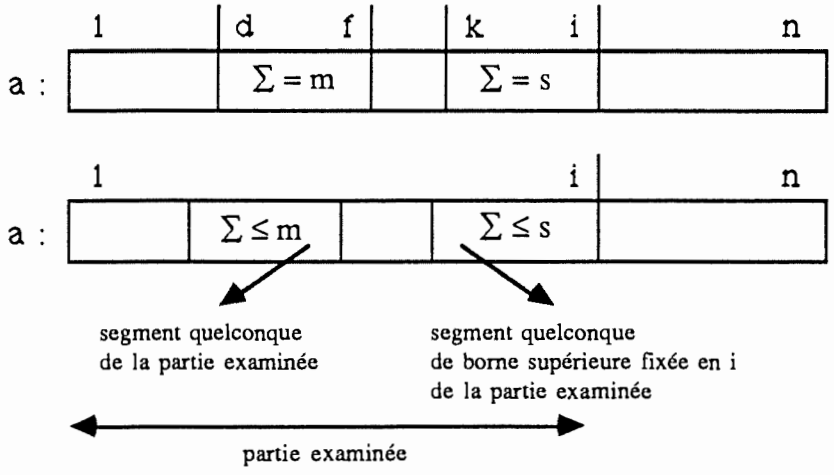
2.3.2. Solution

2.3.2.1. Spécification

- Objets utilisés :
  - objets principaux : a : array[1..n] of integer;
    $n \geq 0$, constante;
    d, f : integer;
  - objets auxiliaires : i, s, m, k : integer;
- Précondition :
  - a[1..n] initialisé;
- Postcondition :
  - $1 \leq d \leq f+1 \leq n+1$
  - $\forall i, j : 1 \leq i \leq j+1 \leq n+1 : \sum_{k=i}^{j} a[k] \leq \sum_{k=d}^{f} a[k]$
  - a inchangé

2.3.2.2. Description graphique de la situation générale

Supposons que l'on a déjà effectué une partie du travail : on a déjà examiné la partie a[1..i] du tableau. Pour espérer posséder suffisamment d'informations en fin d'exécution pour résoudre le problème posé, il faut que la situation suivante soit établie :



- Figure 1 -

2.3.2.3. Description mathématique de la situation générale

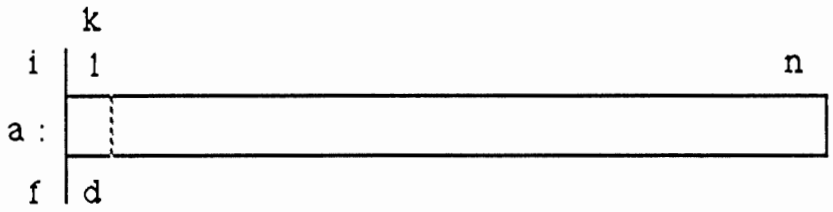
Invariant :

- $1 \le k \le i+1 \le n+1$
- $1 \le d \le f+1 \le i+1$
- $\sum_{p=d}^{f} a[p] = m$
- $\sum_{p=k}^{i} a[p] = s$
- $\forall\ d', f' : 1 \le d' \le f'+1 \le i+1 : \sum_{p=d'}^{f'} a[p] \le m$
- $\forall\ k' : 1 \le k' \le i+1 : \sum_{p=k'}^{i} a[p] \le s$

2.3.2.4. Choix des instructions

Le choix des instructions est fait ici en raisonnant sur la situation représentée graphiquement.

- Initialisations :
  - Pour établir la situation générale de la façon la plus simple, il suffit d'établir la situation suivante :



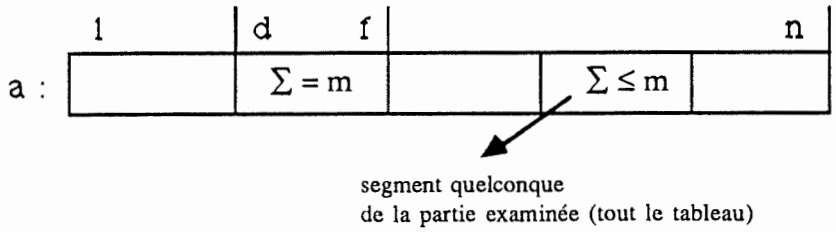
- Figure 2 -

En effet, si i = 0, le seul segment possible inclus dans a[1..i] est le segment vide dont la somme des éléments est nulle.

  - La suite d'instructions d'initialisation sera donc :

```
d := 1; f := 0; m := 0;
k := 1; i := 0; s := 0;
```

- Condition de terminaison et clôture :
  - La situation constituant un cas particulier de la situation générale, mais la plus proche de la situation finale, est la suivante :



- Figure 3 -

En effet, cette situation est vérifiée lorsque i = n, donc lorsque tout le tableau a été examiné.

D'autre part, on constate que la Postcondition est exactement la description de cette situation, donc qu'il n'y aura nul besoin d'instructions de clôture.

- La condition de terminaison sera :

  (i = n)

  et la suite des instructions de clôture sera vide.

- Itération :
  - Si la condition de terminaison n'est pas vérifiée, pour s'approcher de la situation finale, il est nécessaire d'augmenter la partie du tableau déjà examinée.

    Pour cela, nous aurons l'instruction :

    i := i + 1;

    Pour rétablir la situation générale, il faut d'abord déterminer le segment de somme maximale se terminant en i :

    s := s + a[i];
    si (s ≤ 0) alors k := i+1; s := 0 is

    Il faut ensuite comparer ce segment avec le segment de somme maximale déjà obtenu précédemment :

    si (m < s) alors d := k; f := i; m := s is

  - La suite d'instructions d'itération rétablissant la situation générale sera donc :

    i := i + 1;
    s := s + a[i];
    si (s ≤ 0) alors k := i+1; s := 0 is
    si (m < s) alors d := k; f := i; m := s is

2.3.2.5. Vérification de la terminaison

La valeur de la variable i est bornée par n et croît strictement à chaque itération.

2.3.2.6. Ecriture du programme

Le morceau de programme suivant respecte la syntaxe Pascal et résout le problème posé :

```
const   n = 10 {par exemple};

var     a : array[1..n] of integer;
            i, k, s, d, f, m : integer;

begin
        d := 1; f := 0; m := 0;
        k := 1; i := 0; s := 0;
        while (i <> n) do
        begin
                i := i + 1; s := s + a[i];
                if (s ≤ 0) then  begin k := i + 1; s := 0 end;
                if (m < s) then  begin m := s; d := k; f := i end
        end
end
```

2.4. Problèmes rencontrés

La méthode de construction de programmes telle que nous venons de la décrire est effectivement enseignée à l'Institut d'Informatique de Namur. Cet enseignement nous permet de cerner précisément les problèmes rencontrés par les étudiants dans l'application de la méthode.
Il semble possible de répartir de manière générale ces problèmes selon deux catégories que nous allons brièvement décrire.

2.4.1. Raisonner en termes de "situations"

La puissance de ce mode de construction de programmes repose sur le fait qu'il permet de structurer ceux-ci, non pas en termes de séquences d'instructions, mais en termes de *situations successives*. Il est ainsi possible de raisonner sur le nombre limité d'opérations qui permet de "passer" d'une situation à l'autre, plutôt que directement sur la globalité du problème, ce qui, même pour la classe des problèmes envisagée, dépasse bien souvent les capacités de l'esprit humain.
Malheureusement, cette manière de raisonner n'est pas intuitive. Les étudiants préfèrent, naturellement, simuler le déroulement d'un exemple et construire directement le programme, même s'ils sont contraints de tenter par la suite de justifier le résultat par la méthode proposée pour "contenter" l'enseignant.

2.4.2. Formulation "mathématique" des énoncés

La démarche proposée peut être appliquée avec deux niveaux de rigueur. Le premier permettant simplement de construire correctement le programme correspondant à une spécification, le second impliquant en plus de *démontrer* la correction de ce programme en justifiant de façon rigoureuse les différentes étapes de la démarche.

Dans la pratique, on constate une réelle difficulté de la part des étudiants pour franchir le cap du premier niveau. En effet, étant donné la classe des problèmes envisagés, il est presque toujours possible d'exprimer une situation sous la forme d'un schéma (voir exemple ci-dessus) assez simple et correspondant bien à une vue intuitive des choses. Cependant, même si l'étudiant s'acquitte correctement de cette tâche, il parvient généralement avec beaucoup plus de peines à fournir une description mathématique correcte de la même situation.

Il est alors en mesure de construire une solution correcte d'après la description "schématique", mais ne peut justifier rigoureusement son raisonnement.

3. Aperçu du système

L'outil que nous envisageons de réaliser, en tant qu'environnement d'aide à l'utilisateur, devrait apporter une solution concrète aux deux types de difficultés présentés ci-dessus.

En effet, le système sera doté de deux langages d'expression de spécifications et de situations :

- un langage graphique de représentation de schémas, basé sur la notion de "segment",
- un langage mathématique d'expression d'assertions, basé sur les notions de "suite" et d'"ensemble".

Le système disposera aussi de plusieurs fonctions de vérification :

- vérifications syntaxiques,
- vérification de la cohérence entre l'expression graphique et mathématique d'une même situation,
- vérification de la correction d'une séquence d'instructions par rapport à une situation initiale et une situation finale,
- vérifications de cohérence entre différentes étapes de la démarche,
- vérifications de complétude,
- ...

En fonction de toutes ces vérifications, le système pourra fournir à l'utilisateur, à chaque étape, des commentaires sur la manière dont il a réussi cette étape, allant du simple signal d'erreur à l'explication du problème de raisonnement qui a provoqué l'erreur, en passant par la présentation de contre-exemples.

Enfin, le système devra proposer plusieurs scénarios d'utilisation en fonction des besoins :

- un scénario standard consistant à suivre une à une toutes les étapes de la démarche, avec la possibilité de retours en arrière en fonction des commentaires du système,
- la possibilité d'effectuer isolément une étape particulière de la démarche,
- un scénario basé uniquement sur le langage graphique ou uniquement sur le langage mathématique,
- ...

Ainsi conçu, nous espérons que notre outil :

- aidera les étudiants à réfléchir sur la manière dont ils résolvent un problème, et à raisonner avec une plus grande précision, en utilisant le langage mathématique,
- leur apportera une assistance proche de celle qui pourrait leur être fournie par un instructeur, dès qu'elle s'avèrera nécessaire (plutôt qu'en fin d'exercice),
- leur fournira un environnement agréable, les engageant à utiliser la méthode telle que nous la proposons.

Les principales fonctionnalités du système tel que nous envisageons de le construire sont décrites plus en détail dans la suite de cet article.

4. Langage de description de schémas

Dans la plupart des cas, les situations à considérer peuvent être décrites sous forme d'un *ensemble de schémas* représentant chacun un contenu de tableau. Chaque schéma peut se caractériser par un *ensemble de segments contigus,* des conditions à vérifier par le contenu des segments et des relations entre ces segments.

Notre système mettra donc à la disposition de l'utilisateur un outil de description et de manipulation de schémas dont le concept fondamental sera la notion de segment.

Nous décrivons ci-dessous les principaux concepts et mécanismes de ce langage.

4.1. Le concept de segment

Un segment est une suite d'éléments consécutifs d'un tableau.

Il peut se caractériser par :

- la manière dont ses bornes sont fixées,
- les propriétés de son contenu,
- ses liens avec d'autres segments.

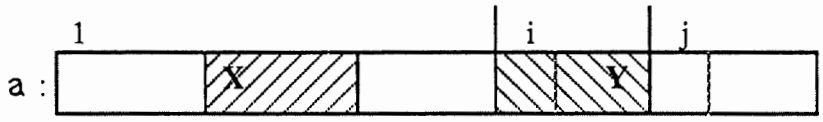
4.1.1. Règles relatives aux bornes des segments

- Un segment est repéré par ses bornes.

  La *borne* inférieure (supérieure) d'un segment est l'indice de son premier (dernier) élément.

  Une borne est fixe ou mobile. Elle est *fixe* si elle est liée à la valeur d'une variable c'est-à-dire si la valeur de cette variable est en permanence égale à la borne (ou, éventuellement, à l'indice précédant ou suivant la borne). Une borne est *mobile* si elle n'est pas liée à la valeur d'une variable. Sa valeur est alors quelconque

  Exemple :



- Figure 4 -

  Les bornes du segment X sont mobiles et celles du segment Y sont fixes puisque sa borne inférieure vaut i et sa borne supérieure vaut j-1.

- Un *segment* est *fixe* ou *mobile*. Il est fixe si ses deux bornes sont fixes. Il est mobile si au moins une de ses bornes est mobile. Un segment mobile représente en fait une famille de segments : tous les segments possibles dans l'espace de mobilité des bornes.

  Exemple : le segment X est mobile : il représente tous les segments possibles de a[1..i-1].

- Un segment peut être *nommé*. Dans cet article, nous nommerons les segments par des lettres majuscules (X, Y, ...).

- Un segment peut être *vide*. Il est vide si sa borne inférieure est égale à sa borne supérieure plus un.

4.1.2. Propriétés du contenu des segments

Le langage graphique devra permettre de décrire un grand nombre de propriétés du contenu d'un segment.

Pour cela, il faudra qu'il dispose d'un ensemble très large de primitives de sorte que l'utilisateur puisse se contenter de les combiner de manière simple sans rencontrer de problèmes de formulation mathématique et qu'ainsi la définition de schémas garde au maximum son caractère intuitif. Idéalement, le système devrait permettre de réaliser des schémas aussi simples à comprendre que ceux que l'on construirait de manière ad hoc, pour un problème déterminé.

Les propriétés du contenu d'un segment s'exprimeront par des conditions à indiquer dans la représentation graphique du segment. Une condition pourra comporter des éléments tels que constantes, variables, opérateurs arithmétiques et logiques, mais aussi des primitives plus puissantes et des notations simplificatrices dont voici quelques exemples :

- un élément quelconque d'un segment sera noté "•",
- l'indice d'un élément quelconque sera noté "i(•)",

Ainsi :

-	" $\bullet \leq 0$ "	signifie	tous les éléments du segment sont négatifs ou nuls.
-	" • pair "	signifie	tous les éléments du segment sont pairs
-	" $\Sigma = s$ "	signifie	la somme des éléments du segment est égale à la valeur de la variable s.
-	" croissant "	signifie	les éléments du segment sont triés de façon croissante.
-	" permuté "	signifie	le contenu du segment constitue une permutation de son contenu initial.

On peut également admettre que l'information que l'on veut exprimer décrit incomplètement une situation précise, ce qui permettra d'envisager des propriétés plus "floues" telles que la suivante :

-	" examiné "	signifie	l'algorithme a déjà passé en revue le contenu du segment.

Même si cette condition est imprécise, elle permet certaines vérifications.

4.1.3. Liens entre les segments

Pour exprimer au mieux l'ensemble des situations rencontrées dans la classe des problèmes traités, le langage devra aussi offrir la possibilité de décrire des relations entre segments.

Certaines de ces relations peuvent s'exprimer par la présence des segments sur un même schéma :

- Deux segments quelconques figurant sur un même schéma sont disjoints.
- Si deux segments sont contigus (sur le schéma), la borne supérieure du premier est égale à la borne inférieure du second moins un.
- Si un segment est avant un autre (sur le schéma), sa borne supérieure est inférieure à la borne inférieure de l'autre.

D'autres relations doivent s'exprimer en représentant les segments sur des schémas différents. Par exemple, pour exprimer que X et Y sont totalement indépendants, il suffit de les représenter sur des schémas différents.

Exemples de relations :

- " $\bullet \in Y$ " signifie l'ensemble des éléments du segment contenant cette propriété est inclus dans l'ensemble des éléments du segment nommé Y.
- " $\Sigma \leq \Sigma(Y)$ " signifie la somme des éléments du segment est inférieure ou égale à la somme des éléments du segment Y.

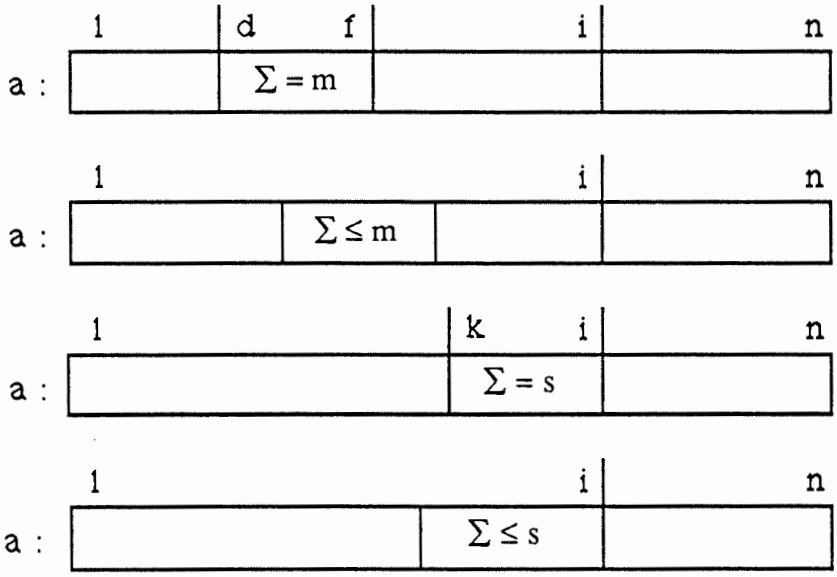
Et des relations plus compliquées peuvent être exprimées en imposant à deux segments des conditions dans lesquelles figurent des variables communes.

4.2. Application de ces concepts à la description de situations

Pour décrire une situation en utilisant les concepts définis ci-dessus, il sera souvent nécessaire de superposer plusieurs schémas représentant des situations compatibles.

Le système vérifiera cette compatibilité (si c'est possible) et affichera éventuellement un schéma de synthèse résumant la situation, mais ne respectant pas les règles d'interprétation stricte des schémas.

Par exemple, la description de la situation générale de l'exercice présenté au paragraphe 2.3 devra se faire à l'aide des quatre schémas suivants :



- Figure 5 -

Interprétation :

- schéma 1 :
  - le segment [d..f] est fixe et la somme de ses éléments est égale à m,
  - ce segment est inclus dans la partie [1..i] du tableau a,
  - ce segment peut être vide.

- schéma 2 :
  - la somme des éléments du segment mobile représenté est inférieure ou égale à m,
  - donc, la somme des éléments de tous les segments possibles de la partie [1..i] du tableau a est inférieure ou égale à m,
  - donc, le segment [d..f] est le segment de somme maximale de la partie [1..i] du tableau a.
- schéma 3 :
  - le segment [k..i] est fixe et la somme de ses éléments est égale à s,
  - ce segment peut être vide.
- schéma 4 :
  - la somme des éléments du segment mobile dont la borne supérieure est fixe (liée à la variable i) est inférieure ou égale à s,
  - donc, la somme des éléments de tous les segments possibles dont la borne supérieure est i est inférieure ou égale à s,
  - donc, le segment [k..i] est le segment de somme maximale dont la borne supérieure est i.

Ces quatre schémas sont superposables : ils représentent des situations compatibles.

Le système pourrait effectuer des superpositions pour afficher la même situation grâce aux deux schémas tels que présentés au paragraphe 2.3 (figure 1). Mais il faut noter que s'ils étaient directement introduits par l'utilisateur, les deux ensembles de schémas ne représenteraient pas la même situation et donneraient lieu à des interprétations différentes par le système.

En effet, face à la situation correspondant aux schémas de la figure 1, le système effectuerait notamment les interprétations suivantes :

- le segment [d..f] est avant le segment [k..i],
- $f < k$,
- le segment mobile caractérisé par " $\Sigma \leq m$" est avant le segment mobile caractérisé par " $\Sigma \leq s$ ".

Ces interprétations ne correspondent ni à l'interprétation intuitive de la situation générale, ni à l'interprétation par le système de la situation représentée par l'ensemble des quatre schémas ci-dessus.

5. Langage mathématique

5.1. Concepts

Pour exprimer de manière mathématique les situations auxquelles nous nous intéressons, il paraît suffisant de pouvoir construire des formules manipulant des quantificateurs et permettant de créer et manipuler des ensembles et des suites. Un langage de ce type est défini dans [Fisette85] [Wenzi87].

Le langage devra donc disposer d'une large gamme d'opérateurs puissants et de mécanismes de composition de ceux-ci. La syntaxe devra être aussi agréable que possible grâce à l'utilisation des symboles mathématiques habituels et d'opérateurs infixés.

Ainsi, l'invariant et les spécifications présentées dans l'exercice du segment de somme maximale pourront pratiquement être écrits tels quels dans notre langage.

5.1.1. Notions de suite et d'ensemble

De même que dans le langage graphique la notion de base était le segment, ce langage-ci peut être construit sur base des notions de suite et d'ensemble. Ces deux notions étant couramment utilisées en informatique, nous n'en donnerons pas ici une définition formelle. Une connaissance intuitive de ces notions est en effet suffisante pour la compréhension de cet article.

Par exemple, si, dans un schéma représentant le tableau a, un segment fixe dont les indices des bornes sont 1 et i est caractérisé par la propriété "# = s" (signifiant que le nombre d'éléments de valeurs différentes du segment est égal à la valeur de la variable s), alors, dans le langage mathématique, cette propriété pourra s'exprimer par "#{a[1..i]} = s" où les accolades servent à désigner l'ensemble des valeurs contenues dans a[1..i].

5.1.2. Opérateurs

Les opérateurs qui devront être mis à la disposition de l'utilisateur devront être suffisamment puissants pour que l'expression d'une situation ne constitue pas un effort important de programmation.

Exemples d'opérateurs : $\forall$, $\exists$, Σ , $\prod$, $\cap$, $\supset$, max, crois, ...

5.1.3. Définition de nouvelles primitives

Malgré la dotation de primitives puissantes, pour certains problèmes, le langage ne pourra que difficilement exprimer les situations relatives à ceux-ci. Il faudra donc prévoir des mécanismes permettant à l'utilisateur de définir de nouvelles primitives.

Cependant, la définition de ces primitives posera souvent des problèmes de programmation plus difficiles que ceux de la classe envisagée au départ. Cette tâche sera donc réservée aux enseignants désireux d'accroître la puissance d'expression du langage.

5.2. Problèmes de syntaxe

Comme nous l'avons dit ci-dessus, au point de vue de la syntaxe du langage mathématique, notre objectif est que celle-ci soit la plus proche de celle utilisée généralement sur papier, et donc la plus agréable possible pour l'utilisateur.

Cependant, cette syntaxe est impossible à mettre en oeuvre étant donné les limites liées à l'implémentation et aux outils utilisés (un clavier ne possède pas l'ensemble des caractères que nous voudrions voir apparaître dans la syntaxe). La solution que nous avons retenue est de définir deux niveaux de langage, le premier concernant l'introduction des expressions mathématiques et le second concernant leur affichage.

Ces remarques sont également valables, quoique dans une moindre mesure, pour le langage de description de schémas, nous adopterons donc pour celui-ci une solution du même type.

6. Scénarios d'utilisation du système

Dans cette partie, nous décrirons l'interface entre l'utilisateur et le système : quels services pourront être offerts par le système, quels choix seront laissés à l'utilisateur.

Signalons dès à présent la caractéristique essentielle de tous les scénarios d'utilisation du système : la possibilité permanente pour l'utilisateur d'effectuer autant de retours en arrière qu'il le

souhaite. Cette possibilité est rendue indispensable par le fait que d'une part, une erreur à une étape pourra en provoquer une autre à une étape suivante et que d'autre part, certains types d'erreurs ne pourront pas donner lieu à un diagnostic dans l'étape où elles ont été commises mais plus tard dans la démarche, le système se bornant à faire dans l'étape courante des vérifications de cohérence.

6.1. Scénario standard

Une façon d'utiliser le système sera d'appliquer la méthode telle que nous l'avons décrite, de fournir au système les résultats de chacune des étapes. Ces résultats seront alors évaluées et l'étape suivante de la démarche sera proposée lorsqu'ils seront jugés satisfaisants.

Imaginons un étudiant utilisant le système et résolvant l'exercice du segment de somme maximale présenté précédemment.

Pour chaque étape de la démarche, nous exposerons ci-dessous :

- des résultats possibles que cet étudiant pourrait fournir au système,
- les réactions du système face à ces résultats,
- des commentaires sur la façon dont le système pourrait s'y prendre pour engendrer ces réactions.

Le lecteur pourra comparer en permanence les résultats proposés avec ceux considérés comme corrects, présentés dans l'exemple.

6.1.1. Construction de la spécification

- Interventions de l'utilisateur :

L'utilisateur devra d'abord décrire les objets principaux. Ensuite, il décrira la précondition et la postcondition d'une part sous forme de schémas, d'autre part dans le langage formalisé.
Par contre, pour les objets auxiliaires, l'utilisateur ne sera pas tenu d'en fournir la liste à ce moment, il pourra les accumuler au fur et à mesure des étapes suivantes de la démarche, chaque fois qu'ils lui apparaîtront nécessaires.

- Vérifications du système :

Il est utopique d'espérer une vérification de la correction de spécifications par rapport à un énoncé intuitif puisque ce dernier est non formalisé. On ne pourra donc envisager, à cette étape, que des vérifications de cohérence.

Citons quelques vérifications possibles :

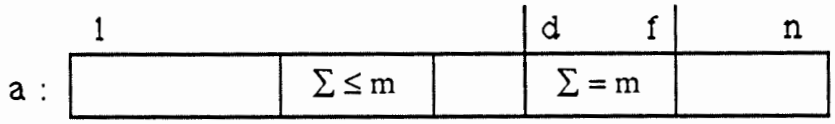
- vérifications syntaxiques par rapport aux langages graphique et mathématique,
- cohérence des schémas (ils expriment bien des situations non contradictoires ou réalisables),
- vérifications de complétude (tous les objets principaux apparaissent dans la précondition ou la postcondition, toutes les variables utilisées dans la précondition et la postcondition sont bien déclarées dans les objets principaux, la précondition ne contient pas d'objets qui ne sont plus référencés dans la postcondition, ...),
- ...

- Exemples :

1 U:(1) objets principaux :
a : array[1..n];
d , f : integer;

S: Les éléments du tableau a sont-ils de type quelconque ?
Le tableau a peut-il être vide ?

2 U: précondition : /
postcondition :

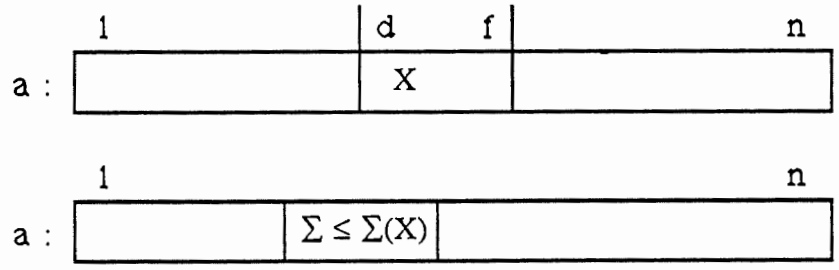


- Figure 6 -

S: 1. Le contenu du tableau ne devrait-il pas être initialisé ?
2. Interprétation du schéma :
- la somme des éléments du segment fixe [d..f] est égale à m,
- le segment mobile dont la somme est inférieure ou égale à m est avant le segment fixe [d..f],
- donc le segment [d..f] est de somme supérieure ou égale à celle de tous les segments qui le précèdent.

C: Nous espérons que la présentation de l'interprétation du schéma par le système permette à l'utilisateur de vérifier si cela correspond bien à son interprétation intuitive. Dans ce cas-ci, l'utilisateur constatera qu'il ne souhaitait pas représenter une telle situation et se rendra compte de la nécessité de deux schémas pour l'exprimer.

3 U: - Formulation graphique :



- Figure 7 -

- Formulation mathématique :

- $1 \le d \le f+1 \le n+1$
- $\forall\, i, j : 1 \le i \le j+1 \le n+1 : \quad \sum_{k=i}^{j} a[k] \le \sum_{k=d}^{f} a[k]$

S: O.K., Les deux formulations ne sont pas contradictoires.

---

(1) U = utilisateur, S = système, C = commentaires.

6.1.2. Description de la situation générale

- Interventions de l'utilisateur :

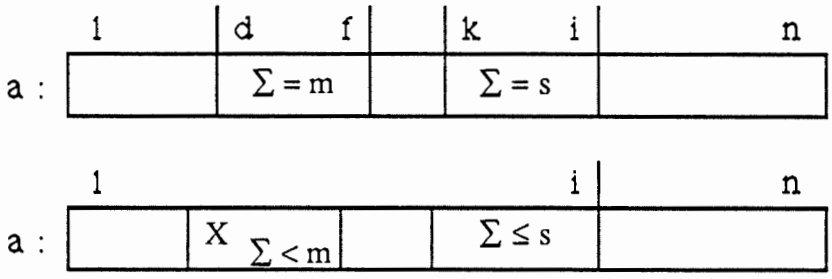
L'utilisateur devra d'abord décrire la situation générale à l'aide du langage graphique, et ensuite fournir un invariant équivalent en langage mathématique.

- Vérifications du système :

Le système pourra vérifier la cohérence de la description graphique : les schémas représentent des situations possibles, ils sont bien superposables, ... Il fera également le même genre de tests pour l'invariant.
Le système pourra ensuite effectuer des comparaisons et des tests de cohérence entre les deux descriptions. Remarquons cependant que leur équivalence logique ne pourra être démontrée formellement que pour des conditions simples comme le non-dépassement de bornes. Il faudra donc mettre en oeuvre d'autres techniques de vérification telles que des tests sur base d'exemples (le système génère des situations réelles correspondant à la situation générale, en donnant des valeurs aux variables et tableaux, teste si l'invariant est vérifié pour ces situations réelles et, le cas échéant, présente le contre-exemple à l'utilisateur).
Signalons enfin que d'autres vérifications concernant la situation générale pourront être effectuées dans les étapes suivantes.

- Exemples :

1 U: situation générale :



- Figure 8 -

S: L'interprétation du schéma conduit à une contradiction : le segment X et le segment [d..f] sont caractérisés par des propriétés incompatibles et peuvent coïncider.

C: En effet, le segment X est mobile, il représente donc un ensemble de segments dont une instance pourrait être [d..f], cela signifie donc que la somme des éléments de [d..f] devrait être en même temps strictement inférieure et égale à m !

2 U: situation générale : voir Figure 1 de l'exemple (§ 2.3.2.2),
Invariant : voir invariant de l'exemple (§ 2.3.2.3).

S: Incohérence entre le schéma et sa formulation mathématique : l'interprétation du schéma exprime que f < k, mais cette propriété ne se retrouve pas dans la formulation mathématique.

6.1.3. Choix des instructions

- Interventions de l'utilisateur :

L'utilisateur devra exprimer successivement les suites d'instructions suivantes: l'initialisation, la condition de terminaison, la clôture et l'itération.

- Vérifications du système :

Les principales vérifications se feront dans cette étape, grâce à des techniques de recherche de contre-exemples et à l'expertise du système.

Le système vérifiera la syntaxe des suites d'instructions, les propriétés liant chaque suite d'instructions à la situation générale et présentera les erreurs sous forme de schémas, contre-exemples, ...

- Exemples :

1 U: initialisations : d := 1; f := 1; m := a[1];
k := 1; s := a[i];
i := 2;

S: L'ordre des affectations est incorrect : lors de l'exécution de l'affectation " s := a[i] ", la valeur de i est indéterminée.
De plus, si n = 0, l'élément a[1] n'existe pas et donc l'affectation " m := a[1] " n'a pas de sens.

2 U: condition de terminaison :
i = n+1

S: contradiction avec la condition " i+1 ≤ n+1 " de l'invariant.

C: En effet, " i+1 ≤ n+1 " est équivalent à " i ≤ n ", donc la condition de terminaison ne sera jamais remplie.

3 U: clôture :
m := 0;

S: L'instruction de clôture est inutile puisqu'elle affecte une variable qui n'intervient pas dans la postcondition.

4 U: itération :
... ; i := i+2; ...

S: La condition " i+1 ≤ n+1 " de l'invariant n'est pas vérifiée après l'exécution de l'itération.

C: En effet, une situation permise avant l'itération est " i = n-1 ", donc, après celle-ci, on aura " i = n+1 ", situation non permise par l'invariant.

6.1.4. Vérification de la terminaison

- Interventions de l'utilisateur :

Expression d'une fonction des valeurs des variables bornée et strictement croissante.

- Vérifications du système :

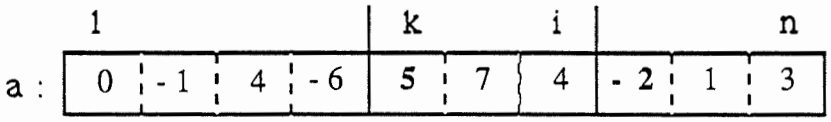
Le système vérifiera si cette fonction est bien bornée et croît bien à chaque exécution de l'itération.

- Exemples :

1 U: fonction bornée et strictement croissante :

$f = k$

S: Contre-exemple :



- Figure 9 -

Si on exécute les instructions d'itération dans cette situation, on constate que la valeur de k n'est pas modifiée.

6.1.5. Ecriture du programme

- Interventions de l'utilisateur :

Regrouper les différentes suites d'instructions et en faire un morceau de programme Pascal.

- Vérifications du système :

Les principales vérifications que le système pourra effectuer consisteront à vérifier que le programme reprend bien les suites d'instructions décrites précédemment et que la syntaxe Pascal est respectée.
Le système pourra également fournir un environnement de test à l'utilisateur en lui offrant un dialogue d'introduction des données de son programme, en exécutant celui-ci, en vérifiant les situations décrites lors de la construction et en présentant à l'utilisateur les résultats.

6.2. Autres scénarios

D'autres scénarios d'utilisation du système seront proposés :

- suivre la démarche selon le premier niveau de rigueur, c'est-à-dire en n'exprimant les situations que sous forme de schémas, sans utiliser le langage mathématique,
- inversement, ne fournir au système que l'expression mathématique des situations,
- exécuter une ou plusieurs étapes particulières de la démarche, par exemple la dernière pour vérifier qu'un programme quelconque est syntaxiquement correct,
- ...

6.3. Niveaux de vérification

Plusieurs niveaux de vérification peuvent être définis selon la complexité du problème.

1. Vérifications simples de cohérence et complétude avec messages d'erreur.
2. Présentation des erreurs détectées sous forme de contre-exemples.
3. Interprétation de ces erreurs sur base d'une certaine expertise (détection des erreurs de raisonnement classiques).
4. Démonstrations formelles de l'exactitude des raisonnements.

Dans les limites du possible, notre système tentera de fournir à l'utilisateur l'aide de niveau maximum.

7. Perspectives

Comme nous l'avons mentionné, la réalisation de ce système fait l'objet d'un projet de recherche à l'Institut d'Informatique de Namur.

Outre les points développés dans cet article, cette recherche porte notamment sur la constitution d'un répertoire d'exercices représentatifs abordés par les étudiants, l'analyse de la résolution de ces exercices afin d'étoffer l'expertise à intégrer dans le système, l'étude des techniques existantes de détection et d'interprétation des erreurs de programmation et des heuristiques de construction d'invariants.

Après implémentation, le système sera expérimenté avec la collaboration d'étudiants. Cette expérimentation devra permettre d'y apporter des améliorations au point de vue interface aussi bien qu'au point de vue efficacité de l'expertise.

8. Remerciements

Les auteurs remercient tous les membres de l'Institut d'Informatique qui ont lu et critiqué une version précédente de cet article: G. Barreto, J. Burnay, P. De Boeck, B. Delcourt, Y. Deville, F. Dubisy, J-P. Hogne, J-M. Jacquet et G. Thiry.

Références

[Arsac80] J. Arsac, *Premières leçons de programmation* , Nathan, Paris, 1980.

[Arsac83] J. Arsac, *Les bases de la programmation* , Dunod informatique, Paris, 1983.

[Arsac84] J. Arsac, *La conception des programmes* , In *Pour la science* , Novembre 1984.

[Dijkstra76] E.W. Dijkstra, *A discipline of programming* , Prentice Hall, 1973.

[Fisette85] D. Fisette, *Définition d'un langage de programmation permettant l'expression d'assertions* , mémoire de maîtrise, Namur, 1985.

[Gries81] D. Gries, *The Science of Programming* , Springer Verlag, 1981.

[Johnson86] W.L. Johnson, *Intention-Based Diagnosis of Novice Programming Errors* , Pitman, London, 1986.

[LeCharlier85] B. Le Charlier, *Réflexions sur le problème de la correction des programmes* , thèse de doctorat, Namur, Janvier 1985.

[Leroy75] H. Leroy, *La fiabilité des programmes* , Presses universitaires de Namur, 1975.

[Leroy78] H. Leroy, *La fiabilité des programmes (2)* , Ecole d'été de l'A.F.C.E.T., Namur, Juillet 1978.

[Wenzi87] M. Wenzi, *Implémentation des suites et ensembles pour un langage de programmation permettant l'expression d'assertions* , mémoire de maîtrise, Namur, 1987.

[Wirth75] N. Wirth, K. Jensen, *Pascal User Manual and Report* , Springer Verlag, New York, 1975.

Annexe 2

Langage pour l'expression de conditions

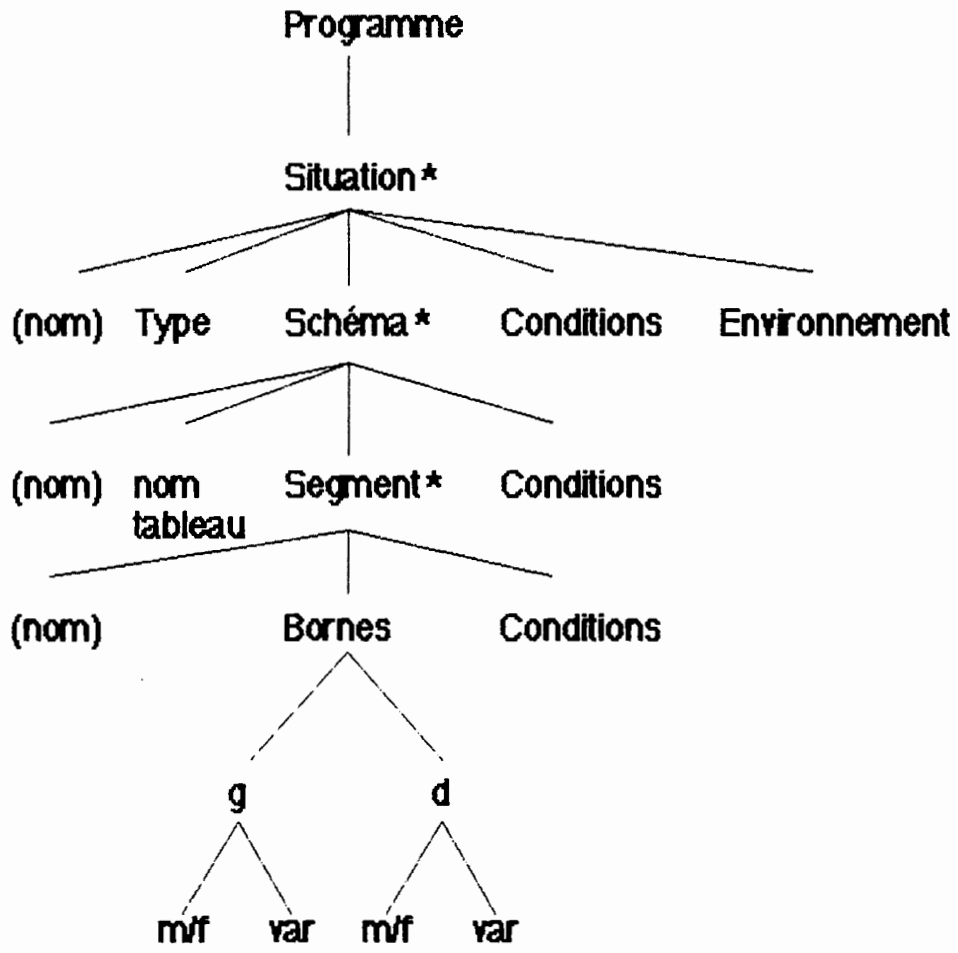
Table des matières

1. Introduction

Nous désirons creer un systeme permettant de mémoriser un ensemble de situations (=assertions).

La notion de situation est présentée dans le chapitre 1 de ce mémoire et clairement définie dans l'annexe 1, article écrit par Marc Derroitte et Baudouin Le Charlier.

Selon la définition donnée, une situation peut se modéliser comme suit :



Il est donc necessaire de définir un langage de conditions afin de pouvoir les exprimer.

Une condition peut porter

soit sur une situation,

soit sur un schéma,

soit sur un segment,

soit sur une borne.

2. Définition des types de valeurs

Dans le langage de conditions, deux types de valeurs sont permis :

2.1. le type ENTIER

Soit Z l'ensemble des entiers,

1. l'ensemble des objets de type entier : Z

2. les opérations primitives :

Addition : Z X Z -> Z : (a,b) --> a + b

Soustraction : Z X Z -> Z : (a,b) --> a - b

Multiplication : Z X Z -> Z : (a,b) --> a * b

Division entière: Z X Z -> Z : (a,b) --> a div b (b <> 0)

Reste: Z X Z -> Z : (a,b) --> a mod b (b <> 0)

Egalité: Z X Z -> B = {vrai,faux}

Inégalité < : Z X Z -> B = {vrai,faux}

(de même pour les autres types de comparaisons : >, >=, <=, <>)

2.2. le type BOOLEEN

1. l'ensemble des objets de type booléen : B = {vrai, faux}

2. les opérations primitives :

Conjonction : B X B -> B : (a,b) --> a and b

Disjonction : B X B -> B : (a,b) --> a or b

Négation : B -> B : a --> not a

3. Quelques définitions

3.1. Situation

Une **situation** est la description schématique d'un ensemble de conditions vérifiées par les objets utilisés à des moments déterminés de l'exécution du programme.

Les objets utilisés sont les constantes, les variables et les tableaux (qui appartiennent à l'environnement).

Une **situation** est caractérisée par :

- un nom éventuel,
- son type (précondition, postcondition ou situation générale),
- son environnement,
- un ensemble de conditions relatives aux objets utilisés,
- un ensemble de schémas.

Un **schéma**, ou un **ensemble de schémas**, est une manière particulière de décrire un tableau appartenant à l'environnement et les conditions relatives à ce tableau caractérisant la situation.

Un **schéma** définit une suite de **segments** consécutifs. Un **segment** étant caractérisé par une paire de bornes (borne inférieure et borne supérieure).

3.2. Situation particulière

Une **situation particulière** est une situation dont les valeurs de toutes les bornes de tous les segments sont fixées.

Soit Sit, une situation particulière caractérisée par un ensemble de n schémas. A chaque schéma i ($1 \leq i \leq n$) correspond un ensemble de bornes $\{bi_1, bs_1, bi_2, bs_2, \ldots, bi_{mi}, bs_{mi}\}$, où mi est le nombre de segments caractérisant le schéma i, bi_k et bs_k, sont respectivement la borne inférieure et la borne supérieure du segment k.

Une **situation particulière** de Sit est déterminée en associant à chaque borne bi_k et bs_k des valeurs sous les conditions suivantes :

- les valeurs des bornes fixes correspondent à l'évaluation des expressions qui y sont liées dans l'environnement,
- $bs_k = bi_{k+1} - 1$,
- $bi_k \leq bs_k - 1$.

Toutes les situations particulières de la situation Sit sont déterminées en faisant varier les valeurs des bornes sous les conditions ci-dessus.

3.3. Environnement

L'**environnement** d'une situation est l'ensemble des constantes, variables et tableaux utilisés dans le programme décrit par la situation.

Avant exécution du programme, les variables et tableaux sont initialisés avec les données du programme. Ensuite, on effectue une copie de cet environnement. Cette copie ne sera jamais modifiée et s'appelle l'**environnement initial** du programme.

3.4. Mémoires, variables et tableaux

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

- il est possible d'y enregistrer de l'information,
- l'information enregistrée peut être lue,
- 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 (entier ou booléen dans le cadre du langage de conditions restreint que nous définissons).

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***. 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*** ; le langage de conditions restreint n'admettant que des tableaux de type entier).

On appelle **VARIABLE INDICEE**, l'association d'un tableau et d'un indice de ce tableau (dans le langage de conditions restreint, une variable indicée doit toujours être de type entier).

3.5. Notion de segment

Un **segment** est constitué d'éléments d'indices successifs d'un tableau.

Un **segment** peut avoir un nom. Il est caractérisé par :

- une borne inférieure, associée à l'indice du premier élément du segment dans le tableau,
- une borne supérieure, associée à l'indice du dernier élément du segment dans le tableau.

4. Définition de la syntaxe et de la sémantique des conditions

4.1. Les symboles de base

‹chiffre› ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

‹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

‹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

‹valeur de vérité› ::= **vrai | faux**

‹entier› ::=

‹chiffre› |

‹entier› ‹chiffre›

‹constante› ::=

‹entier› |

‹valeur de vérité›

‹identificateur› ::=

‹lettre minuscule› |

‹identificateur› ‹lettre minuscule› |

‹identificateur› ‹chiffre›

‹symbole spécial› ::= * | = | - | (|) | _ | < | > | [|] | • | Σ | Π | **à | and | concat | concaténation | dans | de | dernier | div | égal | empty | et | examiné | faux | first | head | inchangé | last | long | max | min | mod | not | or | pair | permutation | permuté | pour | préfixe | premier | produit | queue | segment | somme | suffixe | tail | tête | tricrois | tridec | tristrcrois | tristrdec | variant | vide | vrai**

‹symbole de base› ::=

‹identificateur› |

‹constante› |

‹symbole spécial›

4.2. Expression de désignation

Une expression de désignation de variable (de tableau, ou de segment) est une construction du langage qui, étant donné un environnement E, identifie une et une seule variable (tableau, ou segment).

4.2.1. Syntaxe

‹nom de variable› ::= ‹identificateur›

‹nom de tableau› ::= ‹identificateur›

‹nom de segment› ::= ‹identificateur›

‹expression de désignation de tableau› ::= ‹nom de tableau›

‹expression de désignation de variable› ::=

‹nom de variable› |

‹expression de désignation de tableau› [‹expression arithmétique›]

‹borne› ::= ‹expression arithmétique›

‹expression de désignation de segment› ::=

‹expression de désignation de tableau› [‹borne› : ‹borne›] |

‹nom de segment›

4.2.2. Sémantique

Une expression de désignation possède un type.

Le type d'une expression de désignation de tableau (segment) est le type du tableau (segment) désigné ; le langage de conditions restreint n'admet que les tableaux (segments) de type entier.

Une borne est une expression arithmétique de type entier.

Une expression de désignation de variable est du type de la variable désignée, c'est-à-dire soit de type entier soit de type booléen, cependant, une expression de désignation de variable indicée est de type entier (car désigne un élément de tableau).

Sémantique d'une expression de désignation de tableau :

Soient :

- E, un environnement,
- t, un identificateur,

si t est un nom de tableau dans E alors l'expression de désignation t désigne ce tableau.

Sémantique d'une expression de désignation de variable :

Soient :

- E, un environnement,
- x et t, des identificateurs,

si x est un nom de variable dans E alors l'expression de désignation x désigne cette variable,

si t est un nom de tableau dans E alors l'expression de désignation t[expr], où expr est une expression arithmétique, désigne une variable indicée évaluée de la manière suivante :

1- on évalue l'expression arithmétique expr dans E,

2- si l'évaluation de expr est indéterminée alors l'évaluation de t[expr] est indéterminée, sinon soit i, la valeur obtenue lors de l'évaluation de expr dans E, si i est un des indices du tableau t, alors l'expression de désignation t[expr] désigne l'unique variable indicée t[i] sinon l'évaluation de t[i] est indéterminée.

Sémantique d'une expression de désignation de segment :

Soient :

- sit, une situation,
- E, un environnement de sit,
- t et s, des identificateurs,
- bi et bs, des bornes (ou expressions arithmétiques),

si s est un nom de segment dans E alors l'expression de désignation s désigne ce segment,

si t est un nom de tableau dans E alors l'expression de désignation t[bi:bs] désigne un segment du tableau t qui est caractérisé par les bornes bi et bs, bi étant la borne inférieure et bs la borne supérieure.

4.3. Appels de fonctions

4.3.1. Syntaxe

Pour la syntaxe des différents appels de fonction possibles, veuillez consulter la liste des fonctions prédéfinies.

‹appel de fonction› ::=

‹fonction somme› | ‹fonction produit› |
‹fonction minimum› | ‹fonction maximum› |
‹fonction long› |
‹fonction first› | ‹fonction last› |
‹fonction head› | ‹fonction tail› |
‹prédicat inchangé› | ‹prédicat empty› |
‹prédicat concat› | ‹prédicat égal› |
‹prédicat permutation› | ‹prédicat segment› |
‹prédicat préfixe› | ‹prédicat suffixe› |
‹prédicat tricrois› | ‹prédicat tridec› |
‹prédicat tristrcrois› | ‹prédicat tristrdec› |
‹prédicat pair› | ‹prédicat examiné› |
‹prédicat permuté›

4.3.2. Sémantique

Les fonctions du langage doivent appartenir à la liste des fonctions prédéfinies.

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

- soit de type entier, on a alors une fonction de type entier (ou fonction entière),
- soit de type booléen, on a alors une fonction de type booléen (ou fonction booléenne).

Un appel de fonction doit être conforme à la fonction appelée qui est décrite dans la liste des fonctions prédéfinies (cfr cette liste).

4.4. Expression

4.4.1. Syntaxe

‹expression› ::=

‹expression arithmétique› | ‹expression booléenne›

4.4.2. Syntaxe des expressions arithmétiques

‹opérateur multiplicatif› ::= ***** | **div** | **mod**

‹opérateur additif› ::= + | -

‹facteur› ::=

‹entier› | ‹expression de désignation de variable› |

• (‹expression de désignation de segment›) |

Λ (‹expression de désignation de segment›) |

‹appel de fonction de type entier›

‹terme› ::=

‹facteur› | ‹terme› ‹opérateur multiplicatif› ‹facteur›

‹expression arithmétique› ::=

‹terme› | - ‹terme›

‹expression arithmétique› ‹opérateur additif› ‹terme› |

4.4.3. Syntaxe des expressions booléennes

‹opérateur relationnel› ::= = | **‹=** | **›=** | **‹›** | **‹** | **›**

‹expression booléenne simple› ::=

‹expression de désignation de type booléen› |

‹valeur de vérité› | ‹appel de fonction de type booléen›

‹proposition atomique› ::=

‹expression booléenne simple› |

‹expression arithmétique› ‹opérateur relationnel› ‹expression arithmétique›

‹négation› ::=

‹proposition atomique› | **not** ‹proposition atomique›

‹conjonction› ::=

‹négation› | ‹négation› **and** ‹conjonction›

‹expression booléenne› ::=

‹conjonction› | ‹conjonction› **or** ‹expression booléenne›

4.4.4. Sémantique

Une expression est une construction du langage qui étant donné l'environnement E d'une situation sit, spécifie une suite d'opérations à effectuer pour calculer une valeur v. Cette suite d'opérations est appelée l'**évaluation de l'expression** dans l'environnement E et v, sa valeur.

(Il est toutefois possible que pour certaines valeurs de l'environnement, l'évaluation soit indéterminée ou n'existe pas. Dans ces deux cas l'évaluation de l'expression n'existe pas.)

Une **expression élémentaire** est une expression ne contenant ni symbole • (ce symbole désignant un élément "quelconque" d'un segment), ni symbole **Λ** (ce symbole désignant un "certain" élément d'un segment).

Pour plus de clarté, donnons des exemples d'expressions contenant des symboles de ce type et voyons quelle signification donner à ces expressions.

Soient,

- S, une expression de désignation de segment,
- bi et bs, respectivement la borne inférieure et la borne supérieure de S,
- x, une expression de désignation de variable de type entier.

L'expression booléenne

$$\bullet\,(S) = x$$

signifie

$$\forall\ i \text{ tel que } bi \leq i \leq bs : S[i] = x$$

(c'est-à-dire que tous les éléments du segment S égalent x).

L'expression booléenne

$$\Lambda\,(S) = x$$

signifie

$$\exists\ i \text{ tel que } bi \leq i \leq bs : S[i] = x$$

(c'est-à-dire qu'il existe au moins un élément de S égal à x).

L'évaluation d'une expression élémentaire est un algorithme qui, étant donné

- une situation particulière sit,
- l'environnement E de sit,
- l'environnement initial E_0 relatif à sit,
- une expression expr relative à cette situation sit,

calcule la valeur v de l'expression expr dans sit.

Il existe deux types d'expressions élémentaires :

- expression arithmétique élémentaire,
- expression booléenne élémentaire.

4.4.4.1. L'évaluation d'une expression arithmétique élémentaire

Soient :

- sit, une situation particulière,
- E, l'environnement de sit,
- term, un terme,
- expr, une expression arithmétique élémentaire,
- w_1, un opérateur multiplicatif,
- w_2, un opérateur additif,
- fact, un facteur.

Dans la situation particulière sit et l'environnement E, les expressions ci-dessous sont évaluées de la manière suivante :

fact :

Si **fact** est un entier, alors l'évaluation de fact donne comme valeur cet entier (l'évaluation d'une constante produit toujours la même valeur quelle que soit la situation et quel que soit l'environnement).

Si **fact** est une expression de désignation de variable (c'est-à-dire une variable ou une variable indicée de type entier),

alors l'évaluation de fact donne comme valeur la valeur de la variable désignée par l'expression de désignation dans la situation particulière sit et l'environnement E.

Si **fact** est un appel de fonction de type entier alors cfr la sémantique et la valeur de la fonction en question (les seules fonctions admises étant les fonctions prédéfinies).

term w_1 fact :

On effectue l'évaluation de **term**, puis l'évaluation de **fact**, dans la situation particulière sit et l'environnement E.

Soient v_1 et v_2, les valeurs obtenues, si elles ne sont pas de type entier alors la valeur de l'expression **term w_1 fact** n'existe pas, si elles sont de type entier alors la valeur de l'expression **term w_1 fact** est v_1 w_1 v_2 (c'est-à-dire la valeur de l'opération primitive w_1 appliquée à v_1 et v_2).

expr w_2 term :

On effectue l'évaluation de **expr**, puis l'évaluation de **term**, dans la situation particulière sit et l'environnement E.

Soient v_1 et v_2, les valeurs obtenues, si elles ne sont pas de type entier alors la valeur de l'expression **expr w_2 term** n'existe pas, si elles sont de type entier alors la valeur de l'expression **expr w_2 term** est v_1 w_2 v_2 (c'est-à-dire la valeur de l'opération primitive w_2 appliquée à v_1 et v_2).

-term :

On effectue l'évaluation de **term** dans la situation particulière sit et l'environnement E.

Soit v, la valeur obtenue, si elle n'est pas de type entier alors la valeur de l'expression **-term** n'existe pas, si v et de type entier alors la valeur de l'expression **-term** est -v.

L'évaluation d'une expression arithmétique élémentaire a donc pour effet de renvoyer une valeur de type entier.

4.4.4.2. L'évaluation d'une expression booléenne élémentaire

Soient :

- sit, une situation particulière,
- E, l'environnement de sit,
- E_0, l'environnement initial de sit,
- simplebexpr, une expression booléenne simple,
- $aexpr_1$ et $aexpr_2$, deux expressions arithmétiques,
- aprop, une proposition atomique,
- neg, une négation,
- conj, une conjonction,
- bexpr, une expression booléenne,
- w, un opérateur relationnel.

Etant donné une situation particulière sit, l'environnement initial E_0 et l'environnement E de sit, les expressions ci-dessous sont évaluées de la manière suivante :

simplebexpr :

Si **simplebexpr** est une valeur de vérité (soit **vrai**, soit **faux**) alors l'évaluation de **simplebexpr** donne comme valeur cette valeur de vérité (l'évaluation d'une constante produit toujours la même valeur quels que soient la situation et l'environnement).

Si **simplebexpr** est une expression de désignation de type booléen (c'est-à-dire une variable de type booléen), alors l'évaluation de **simplebexpr** donne comme valeur la valeur de la variable désignée dans la situation particulière sit et l'environnement E.

Si **simplebexpr** est un appel de fonction de type booléen alors cfr la sémantique et la valeur de la fonction en question (les seules fonctions admises étant les fonctions prédéfinies).

$aexpr_1$ w $aexpr_2$:

On effectue l'évaluation de **$aexpr_1$**, puis de **$aexpr_2$**, dans la situation particulière sit et l'environnement E.

Soient v_1 et v_2, les valeurs obtenues (elles doivent être de type entier). La valeur de l'expression booléenne **aexpr1 w aexpr2** est v_1 w v_2 (c'est-à-dire la valeur de l'opération primitive w appliquée à v_1 et v_2).

not aprop :

On effectue l'évaluation de **aprop** dans la situation particulière sit et l'environnement E.

Soit v, la valeur obtenue, elle doit être de type booléen, si v vaut **vrai** alors **not aprop** a pour valeur **faux**, si v vaut **faux** alors **not aprop** a pour valeur **vrai**.

neg and conj :

On effectue l'évaluation de **neg** dans la situation particulière sit et l'environnement E.

Soit v_1, la valeur obtenue, elle doit être de type booléen, si v_1 vaut **faux** alors la valeur de **neg and conj** est **faux**, si v_1 vaut **vrai** alors on effectue l'évaluation de **conj** dans la situation particulière sit et l'environnement E, soit v_2, la valeur obtenue, elle doit être de type booléen, v_2 est la valeur de l'expression **neg and conj**.

conj or bexpr :

On effectue l'évaluation de **conj** dans la situation particulière sit et l'environnement E.

Soit v_1, la valeur obtenue, elle doit être de type booléen, si v_1 vaut **vrai** alors la valeur de **conj or bexpr** est **vrai**, si v_1 vaut **faux** alors on effectue l'évaluation de **bexpr** dans la situation particulière sit et l'environnement E, soit v_2, la valeur obtenue, elle doit être de type booléen, v_2 est la valeur de l'expression **conj or bexpr**.

4.5. Condition

4.5.1. Syntaxe

‹condition› ::= ‹expression booléenne›

4.5.2. Sémantique

Une condition est une expression booléenne.

L'évaluation d'une condition a pour effet de renvoyer une valeur de type booléen (une condition étant soit vraie, soit fausse par rapport à une situation particulière et son environnement).

Il faut maintenant considérer l'évaluation d'une condition.

Soient :

- cond, une condition,
- sit, une situation particulière,
- E, l'environnement de sit,
- $t_1,...,t_n$, n tableaux de E (non nécessairement distincts),
- $X_1,...,X_n$, n segments (non nécessairement distincts) respectivement des tableaux $t_1,...,t_n$,
- $bi_1,...,bi_n$, les bornes inférieures respectivement des tableaux $t_1,...,t_n$,
- $bs_1,...,bs_n$, les bornes supérieures respectivement des tableaux $t_1,...,t_n$.

Dans la situation particulière sit et l'environnement E, la condition **cond** est évaluée de la manière suivante :

- si la condition **cond** ne contient ni symbole •, ni symbole **Λ**, alors **cond** est évaluée comme une expression booléenne élémentaire dans la situation particulière sit et l'environnement E,

- si la condition **cond** contient un (ou plusieurs) symboles •, alors :

  Supposons que **cond** contienne n symboles • portant sur les segments $X_1,...,X_n$: cond ‹ • (X_1),...,• (X_n) ›, alors l'évaluation

de **cond** a pour effet de renvoyer une valeur v de type booléen telle que :

$v =$

$(\forall\ i_1$ dans $[bi_1..bs_1]$ ... $\forall\ i_n$ dans $[bi_n..bs_n]$:

cond $\langle\ t_1[i_1],...,\ t_n[i_n]\ \rangle$),

- si la condition **cond** contient un (ou plusieurs) symboles **Λ**, alors :

  Supposons que **cond** contienne n symboles **Λ** portant sur les segments $X_1,...,X_n$: cond $\langle$ **Λ** $(X_1),...,$ **Λ** $(X_n)\ \rangle$, alors l'évaluation de **cond** a pour effet de renvoyer une valeur v de type booléen telle que :

  $v =$

  $(\exists\ i_1$ dans $[bi_1..bs_1]$ ... $\exists\ i_n$ dans $[bi_n..bs_n]$:

  cond $\langle\ t_1[i_1],...,\ t_n[i_n]\ \rangle$)

- si la condition **cond** contient un (ou plusieurs) symboles • et un (ou plusieurs) symboles **Λ**, alors :

  Supposons que **cond** contienne

  - n symboles • ou **Λ** portant sur les segments $X_1,...,X_n$:

    cond $\langle$ $symb_1$ $(X_1),...$, $symb_n$ $(X_n)\ \rangle$,

    où pour $1 \le j \le n$:

    - soit $symb_j$ = •,

    - soit $symb_j$ = **Λ**,

  alors l'évaluation de **cond** a pour effet de renvoyer une valeur v de type booléen telle que :

  $v =$

  $(Q_1\ i_1$ dans $[bi_1..bs_1]$ ... $Q_n\ i_n$ dans $[bi_{n1}..bs_n]$:

  cond $\langle\ t_1[i_1],...,\ t_n[i_n]\ \rangle$)

  où pour $1 \le j \le n$:

  - $Q_j = \forall$ si $symb_j$ = •,

  - $Q_j = \exists$ si $symb_j$ = **Λ**.

Notons que dans ce dernier cas, l'ordre d'apparition des symboles • et des symboles **Λ** dans l'expression est important.

En effet, montrons cela sur un exemple simple comportant seulement un symbole • et un symbole **Λ** :

Soient les conditions suivantes :

• (X_1) = **Λ** (X_2)

Λ (X_1) = • (X_2)

La première signifie :

$\forall\ i_1$ tel que $bi_1 \leq i_1 \leq bs_1$, $\exists\ i_2$ tel que $bi_2 \leq i_2 \leq bs_2$:

$$X_1\ [i_1] = X_2\ [i_2]$$

La seconde signifie :

$\exists\ i_1$ tel que $bi_1 \leq i_1 \leq bs_1$, $\forall\ i_2$ tel que $bi_2 \leq i_2 \leq bs_2$:

$$X_1\ [i_1] = X_2\ [i_2]$$

Ainsi, supposons :

- un segment X_1 = [1, 2, 3, 4, 5],
- un segment X_2 = [5, 4, 3, 2, 1],

dans ce cas, la première condition sera évaluée à vrai, alors que la deuxième sera évaluée à faux.

Par contre, si:

- X_1 = [1, 2, 3, 4, 5], et
- X_2 = [2, 2],

alors, la première condition vaudra faux, et la deuxième sera évaluée à vrai.

5. Fonctions prédéfinies du langage

Soient ,

- $t_1,\ldots, t_n$, des tableaux,
- $S_1,\ldots, S_n$, des segments (non nécessairement distincts) respectivement de $t_1,\ldots, t_n$,
- $bi_1,\ldots,bi_n$, les bornes inférieures respectivement des segments $S_1,\ldots, S_n$,
- $bs_1,\ldots,bs_n$, les bornes supérieures respectivement des segments $S_1,\ldots, S_n$.

Remarques :

- notation :

  SEG représente l'ensemble des segments.

- syntaxe d'une suite :

  ‹suite› ::=

  ‹expression de désignation de segment› |

  ‹expression de désignation de tableau›

5.1. La fonction SOMME : SEG --› Z

5.1.1. Syntaxe

‹fonction somme› ::=

somme (‹suite›) |

Σ (‹suite›) |

somme pour ‹expression de désignation de variable de type entier› **variant de** ‹expression de désignation de borne› **à** ‹expression de désignation de borne› **dans** ‹suite›

5.1.2. Sémantique

La fonction SOMME appliquée au segment S_1 renvoie une valeur v de type entier telle que v est égale à la somme de tous les éléments du segment S_1 de t_1 :

$$v = t_1[bi_1] + t_1[bi_1 + 1] + \ldots + t[bs_1]$$

5.2. La fonction PRODUIT : SEG --> Z

5.2.1. Syntaxe

‹fonction produit› ::=

produit (‹suite›) |

Π (‹suite›) |

produit pour ‹expression de désignation de variable de type entier› **variant de** ‹expression de désignation de borne› **à** ‹expression de désignation de borne› **dans** ‹suite›

5.2.2. Sémantique

La fonction PRODUIT appliquée au segment S_l renvoie une valeur v de type entier telle que v est égale au produit de tous les éléments du segment S_l de t_l :

$$v = t_l[bi_l] * t_l[bi_l + 1] * \ldots * t[bs_l]$$

5.3. La fonction MINIMUM (MAXIMUM) : SEG --> Z

5.3.1. Syntaxe

‹fonction minimum› ::=

min (‹suite›) |

min pour ‹expression de désignation de variable de type entier› **variant de** ‹expression de désignation de borne› **à** ‹expression de désignation de borne› **dans** ‹suite›

‹fonction maximum› ::=

max (‹suite›) |

max pour ‹expression de désignation de variable de type entier› **variant de** ‹expression de désignation de borne› **à** ‹expression de désignation de borne› **dans** ‹suite›

5.3.2. Sémantique

La fonction MINIMUM (MAXIMUM) appliquée au segment S_l renvoie une valeur v de type entier telle que v est égale à la valeur de l'élément de valeur minimale (maximale) du segment S_l de t_l :

$$v = \min (\max)\ t_l[i] \quad \text{pour } bi_l \leq i \leq bs_l$$

5.4. La fonction LONG : SEG --> Z

5.4.1. Syntaxe

‹fonction long› ::= **long** (‹suite›)

5.4.2. Sémantique

La fonction LONG appliquée au segment S_1 renvoie une valeur v de type entier telle que v est égale au nombre d'éléments du segment S_1 de t_1 :

$$v = bs_1 - bi_1 + 1$$

5.5. La fonction FIRST (LAST) : SEG --> Z

5.5.1. Syntaxe

‹fonction first› ::=

premier (‹suite›) |

first (‹suite›)

‹fonction last› ::=

dernier (‹suite›) |

last (‹suite›)

5.5.2. Sémantique

La fonction FIRST (LAST) appliquée au segment S_1 renvoie une valeur v de type entier telle que v est égale à la valeur du premier (dernier) élément du segment S_1 de t_1 :

$$v = t_1[bi_1]$$

$$(v = t[bs_1])$$

5.6. La fonction HEAD (TAIL) : SEG --> SEG

5.6.1. Syntaxe

‹fonction head› ::=

tête (‹suite›) |

head (‹suite›)

‹fonction tail› ::=

queue (‹suite›) |

tail (‹suite›)

5.6.2. Sémantique

La fonction HEAD (TAIL) appliquée au segment S_1 renvoie un segment S' tel que S' est le même segment que S_1 tronqué de son dernier (premier) élément :

$S' = t_1[bi_1 : bs_1 - 1]$

$(S' = t_1[bi_1 + 1 : bs_1])$

5.7. Le prédicat INCHANGE : SEG --> B

5.7.1. Syntaxe

‹prédicat inchangé› ::=

inchangé (‹suite›) |

(‹suite›) **inchangé**

5.7.2. Sémantique

Dans la situation particulière sit et l'environnement E,

le prédicat INCHANGE appliqué au segment S_1 renvoie une valeur v de type booléen telle que

v vaut **vrai** ssi les valeurs enregistrées dans les mémoires associées aux éléments du segment S_1 dans l'environnement initial E_0 égalent les valeurs enregistrées dans les mémoires associées aux éléments de S_1 dans l'environnement E

et sinon v vaut **faux**.

5.8. Le prédicat EMPTY : SEG --> B

5.8.1. Syntaxe

‹prédicat empty› ::=

vide (‹suite›) |

‹suite› **vide** |

empty (‹suite›) |

‹suite› **empty**

5.8.2. Sémantique

Le prédicat EMPTY appliqué au segment S_1 renvoie une valeur v de type booléen telle que v vaut **vrai** si le segment S_1 est vide et **faux** sinon.

5.9. Le prédicat CONCAT : SEG X SEG X SEG --> B

5.9.1. Syntaxe

‹prédicat concat› ::=

concat (‹suite›, ‹suite›, ‹suite›) |

‹suite› **concaténation de** ‹suite› **et** ‹suite›

5.9.2. Sémantique

Le prédicat CONCAT appliqué aux segments S_1, S_2 et S_3, renvoie une valeur v de type booléen telle que v vaut **vrai** ssi

$(t_1\ [bi_1],\ldots,\ t_1\ [bs_1],\ t_2\ [bi_2],\ldots,\ t_1[bs_2]) = (t_3\ [bi_3],\ldots,\ t_3\ [bs_3])$

5.10. Le prédicat EGAL : SEG X SEG--> B

5.10.1. Syntaxe

‹prédicat égal› ::=

égal (‹suite›, ‹suite›) |

‹suite› **égal** ‹suite› |

‹suite› = ‹suite›

5.10.2. Sémantique

Le prédicat EGAL appliqué aux segments S_1 et S_2 renvoie une valeur v de type booléen telle que

v vaut **vrai** si le segment S_1 égale le segment S_2, c'est-à-dire si

$t_1\ [bi_1] = t_2\ [bi_2],\ldots,\ t_1\ [bs_1] = t_2\ [bs_2]$,

et **faux** sinon.

5.11. Le prédicat PERMUTATION : SEG X SEG --> B

5.11.1. Syntaxe

‹prédicat permutation› ::=

permutation (‹suite›, ‹suite›) |

‹suite› **permutation de** ‹suite›

5.11.2. Sémantique

Le prédicat PERMUTATION appliqué aux segments S_1 et S_2, renvoie une valeur v de type booléen telle que v vaut **vrai** si les éléments du segment S_2 représentent une permutation des éléments du segment S_1, et **faux** sinon.

5.12. Le prédicat SEGMENT : SEG X SEG --> B

5.12.1. Syntaxe

‹prédicat segment› ::=

segment (‹suite›, ‹suite›) |

‹suite› **segment de** ‹suite›

5.12.2. Sémantique

Le prédicat SEGMENT appliqué aux segments S_1 et S_2, renvoie une valeur v de type booléen telle que v vaut **vrai** si le segment S_2 est un sous-segment de S_1 (c'est-à-dire, $bi_1 \leq bi_2 \leq bs_2 \leq bs_1$) et **faux** sinon.

5.13. Le prédicat PREFIXE (SUFFIXE) : SEG X SEG --> Z

5.13.1. Syntaxe

‹prédicat préfixe› ::=

préfixe (‹suite›, ‹suite›) |

‹suite› **préfixe de** ‹suite›

‹prédicat suffixe› ::=

suffixe (‹suite›, ‹suite›) |

‹suite› **suffixe de** ‹suite›

5.13.2. Sémantique

Le prédicat PREFIXE (SUFFIXE) appliqué aux segments S_1 et S_2, renvoie une valeur v de type booléen telle que v vaut **vrai** si le segment S_2 est un préfixe (suffixe) du segment S_1,

c'est-à-dire $bi_1 = bi_2 \leq bs_2 \leq bs_1$ ($bi_1 \leq bi_2 \leq bs_2 = bs_1$) et **faux** sinon.

5.14. Le prédicat TRICROIS (TRIDEC) : SEG --› B

5.14.1. Syntaxe

‹prédicat tricrois› ::=

tricrois (‹suite›) |

‹suite› **tricrois**

‹prédicat tridec› ::=

tridec (‹suite›) |

‹suite› **tridec**

5.14.2. Sémantique

Le prédicat TRICROIS (TRIDEC) appliqué au segment S_1 renvoie une valeur v de type booléen telle que

v vaut **vrai** ssi les éléments du segment sont rangés dans le segment par ordre croissant (décroissant), c'est-à-dire

$t\,[bi_1] \leq t\,[bi_1 + 1] \leq \ldots \leq t\,[bs_1]$ $\quad (t\,[bi_1] \geq t\,[bi_1 + 1] \geq \ldots \geq t\,[bs_1])$,

et **faux** sinon.

5.15. Le prédicat TRISTRCROIS (TRISTRDEC) : SEG --› B

5.15.1. Syntaxe

‹prédicat tristrcrois› ::=

tristrcrois (‹suite›) |

‹suite› **tristrcrois**

‹prédicat tristrdec› ::=

tristrdec (‹suite›) |

‹suite› **tristrdec**

5.15.2. Sémantique

Le prédicat TRISTRCROIS (TRISTRDEC) appliqué au segment S_1 renvoie une valeur v de type booléen telle que

v vaut **vrai** ssi les éléments du segment sont rangés dans le segment par ordre strictement croissant (décroissant), c'est-à-dire

$t[bi_1] < t[bi_1 + 1] < \ldots < t[bs_1]$ $\quad (t[bi_1] > t[bi_1 + 1] > \ldots > t[bs_1])$,

et **faux** sinon.

5.16. Le prédicat PAIR : SEG --> B

5.16.1. Syntaxe

‹prédicat pair› ::=

pair (‹suite›) |

‹suite› **pair**

5.16.2. Sémantique

Le prédicat PAIR appliqué au segment S_1 renvoie une valeur v de type booléen telle que v vaut **vrai** si tous les éléments du segment S_1 sont de valeur paire et **faux** sinon.

5.17. Le prédicat EXAMINE : SEG --> B

5.17.1. Syntaxe

‹prédicat examiné› ::=

examiné (‹suite›) |

‹suite› **examiné**

5.17.2. Sémantique

Le prédicat EXAMINE appliqué au segment S_1 renvoie une valeur v de type booléen telle que v vaut **vrai** si le segment S_1 a été examiné et **faux** sinon.

5.18. Le prédicat PERMUTE : SEG --> B

5.18.1. Syntaxe

‹prédicat permuté› ::=

permuté (‹suite›) |

‹suite› **permuté**

5.18.2. Sémantique

Soient E_0, l'environnement initial de la situation sit et E, l'environnement de sit.

Le prédicat PERMUTE appliqué au segment S_1 renvoie une valeur v de type booléen telle que v vaut **vrai** si les éléments du segment S_1 dans l'environnement E représentent une permutation des éléments du segment S_1 dans l'environnement E_0, et **faux** sinon.

6. Situation vérifiée dans un environnement E

Soient , Sit une situation et E, l'environnement de Sit.

La situation Sit est vérifiée dans l'environnement E si toutes les situations particulières de sit sont vérifiées dans cet environnement E.

La situation particulière sit_1 est vérifiée dans l'environnement E si toutes les conditions qui figurent dans sit_1 sont vraies dans E (c'est-à-dire, si l'évaluation de toute condition de sit_1 dans E a pour effet de renvoyer la valeur booléenne **vrai**).

Annexe 3

Découpe en modules

Table des matières

1. Introduction

Un module implémentera généralement un ou plusieurs types. Il offrira également un ensemble de primitives, notamment les primitives permettant de manipuler les types définis dans ce module. Chaque module contiendra également les déclarations des structures de données utilisées pour implémenter ce module.

Remarque

Les structures de données déclarées dans un module ne correspondent pas nécessairement à des structures des types spécifiés dans le module. Par exemple, des déclarations de structures de type fiche seront effectuées dans le module scénario ou dans le module gestion structure.

La description des modules qui suit est sommaire. Les types et les primitives à implémenter sont seulement énumérés afin de voir si l'architecture proposée peut supporter la réalisation du scénario envisagé. L'analyse de l'architecture n'est pas suffisamment approfondies que pour pouvoir énumérer les structures à déclarer.

2. Contenu des différents modules

2.1. Module scénario

2.1.1. Types à définir

Afin d'implémenter le scénario d'exploitation, il faut définir les types suivants :

- Table des objets du programme introduit
- Fichiers contenant les différents programmes enregistrés

2.1.2. Primitives à implémenter

Pour réaliser le scénario, ce module doit offrir les primitives implémentant les différentes parties du système :

- Le programme principal
- Introduction d'un programme
- Introduction des objets
- Introduction de la précondition
- Introduction de la situation générale

- Introduction de la postcondition
- Introduction des instructions
- Analyse d'un écran édité (analyse de ses différents composants : champs et cadres)
- Exécution du programme
- Sauvetage des données d'un programme introduit (ou modifié)
- Lecture des données d'un programme dans le fichier
- Initialisation de la liste des identificateur de programme déjà enregistrés
- Initialisation de la liste des schémas attachés à la description d'une situation
- Initialisation des données du programme à exécuter
- Production des résultats du programme exécuté
- Affichage du contour d'un cadre (et effacement d'un cadre et de son contour)

2.2. Module analyse

2.2.1. Types à définir

Ce module doit implémenter les types définis dans le langage de conditions repris à l'annexe 2 :

- Représentation d'une condition

Ce module implémentera également :

- Représentation d'une instruction

2.2.2. Primitives à implémenter

- Analyse du contenu d'une fiche (ou analyse d'un cadre)
- Analyse d'un objet
- Analyse d'appartenance à une liste (liste d'identifiants ou liste de données)
- Analyse d'une condition
- Analyse d'une instruction
- Analyse d'un identificateur

- Analyse d'une expression booléenne
- Analyse d'une expression arithmétique
- Analyse d'un terme
- Analyse d'un facteur
- Analyse d'un appel de fonction
- Analyse d'une expression booléenne simple
- Analyse d'une proposition atomique
- Analyse d'une négation
- Analyse d'une conjonction

2.3. Module gestion structures

2.3.1. Types à définir

Ce module implémentera les structures suivantes :

- Représentation d'un programme introduit par l'utilisateur
- Représentation d'un objet
- Représentation d'une situation
- Représentation d'un schéma
- Représentation d'un segment
- Représentation d'un champ

2.3.2. Primitives à implémenter

Ce module offrira les primitives suivantes :

- Affichage des différents écrans (de 1 à 7)
- Affichage de messages
- Affichage d'un schéma
- Affichage d'un segment
- Création d'un segment
- Destruction d'un segment
- Affichage d'un champ
- Saisie d'un champ

- Saisie d'un objet
- Saisie de la valeur d'une constante du programme
- Saisie de la valeur d'une variable du programme introduit
- Saisie des éléments d'un tableau du programme introduit
- Choix dans un menu
- Choix d'un cadre à éditer ou d'un champ à introduire dans un écran
- Choix d'un segment dans un schéma (pour supprimer ce dernier ou pour insérer un autre segment entre ce dernier et le précédent)
- Affichage d'une variable du programme introduit et de sa valeur
- Affichage d'un tableau du programme introduit et des valeurs de ses éléments

2.4. Module gestion cadre

2.4.1. Types à définir

Etant donné les concepts de base définis dans la présentation du scénario, ce module implémentera les types suivants :

- Type fiche
- Type cadre

2.4.2. Primitives à implémenter

Ce module doit offrir les primitives de manipulations des structures de type fiche et de type cadre :

- Initialisation d'une fiche
- Affichage d'une ligne
- Affichage du contenu d'un cadre
- Ajout d'une ligne vide à la fin d'une fiche
- Insertion d'une ligne vide dans une fiche
- Suppression d'une ligne dans une fiche
- Gestion des déplacements du curseur
- Fonctions d'édition

- Effacement du caractère sous le curseur
- Insertion d'une ligne vide à la position du curseur
- Effacement du caractère à gauche du curseur
- Implémentation du "return" sur une fiche
- Effacement de la ligne contenant le curseur
- Effacement de la ligne à partir de la position du curseur
- Ecriture d'un caractère sur une fiche
- Gestion des fonctions d'édition sur une fiche
- Edition globale d'une fiche
- Sauvetage du contenu d'une fiche dans un fichier
- Chargement d'une fiche à partir d'un fichier de type texte
- Destruction d'une fiche
- Modification des dimensions d'un cadre

2.5. Module gestion liste

2.5.1. Type à définir

Ce module implémentera un seul type :

- Type liste

2.5.2. Primitives à implémenter

Ce module doit offrir les primitives de manipulation des structures de type liste :

- Saisie du choix d'une opération sur une liste
- Affichage du contour d'une liste
- Affichage du contenu d'une liste
- Ajout d'un élément en dernière position
- Suppression d'un élément
- Remplacement d'un élément

2.6. Module gestion curseur

2.6.1. Types à définir

- structure enregistrant la position du curseur à l'écran
- structure enregistrant les dimensions de l'espace de déplacement du curseur

2.6.2. Primitives à implémenter

Ce module offrira les primitives suivantes :

- Lecture d'un caractère introduit au clavier
- Emission d'un son
- Déplacements du curseur à l'écran
- Fonctions d'édition
- Effacement du caractère sous le curseur
- Insertion d'une ligne à la position du curseur
- Effacement du caractère à gauche du curseur
- Implémentation du "return" (passage à la ligne)
- Effacement de la ligne contenant le curseur
- Effacement de la fin de la ligne contenant le curseur
- Ecriture d'un caractère à la position du curseur
- Edition globale d'une fiche

2.7. Module exécution programme

2.7.1. Types à définir

Afin de mémoriser les valeurs des variables manipulées au cours de l'exécution du programme introduit par l'utilisateur, ce module implémentera la structure suivante :

- Table de mémoire pour enregistrer les valeurs des variables et tableaux

2.7.2. Primitives à implémenter

Ce module offrira les primitives suivantes :

- Exécution d'une instruction
- Evaluation d'une constante
- Evaluation d'une variable (variable simple ou variable indicée)
- Evaluation d'une condition
- Evaluation d'une situation
- Création des emplacements des variables et tableaux du programme à exécuter
- Calcul d'adresse d'une variable (lors de la création des places mémoires des variables ...)

Annexe 4.1

Spécifications des primitives du module gestion curseur du module gestion cadre

Table des matières

1. Introduction

Ce document regroupe

- les types définis dans les modules gestion curseur et gestion cadre,
- les spécifications des procédures des modules gestion curseur et gestion cadre.

Les spécifications des procédures sont présentées comme suit :

En-tête :

Argument :

Résultat :

Spécifications :

La partie "en-tête" reprend le nom de la procédure et la liste de ses paramètres formels (représente la forme de l'appel de la procédure).

Ensuite, les paramètres sont regroupés en deux ensembles : arguments et résultats. Un paramètre peut apparaître dans les deux ensembles.

Les arguments sont les paramètres fournis à la procédure et les résultats sont les paramètres produits ou modifiés par la procédure.

La partie "spécifications" donne une brève description des effets de la procédure.

2. Spécifications des primitives du module gestion curseur

2.1. Types implémentés

TYPES

POSECRAN = record

posex : integer ;

posey : integer ;

{position du curseur à l'écran}

end ; {end du record posecran}

{mémorise la position du curseur à l'écran en coordonnées relatives à l'espace de déplacement du curseur}

DEPLACEMENTCURSEUR = record

long : integer ; {longueur du déplacement horizontal permis du curseur}

haut : integer ; {longueur du déplacement vertical permis du curseur}

end ; {end du record deplacementcurseur}

{donne l'espace de déplacement du curseur à l'écran}

Remarque

Le concept d'espace de déplacement du curseur est développé au chapitre 5.

2.2. Primitives offertes

2.2.1. Lecture du code d'une touche clavier

En-tête :

lirecar (ch, fonction)

Argument : -

Résultats :

ch : char ;

fonction : boolean ;

Spécifications :

La procédure LIRECAR lit un caractère introduit au clavier :

ch contient le code numérique correspondant à la touche du clavier enfoncée par l'utilisateur,

fonction vaut vrai si ce code correspond au code numérique d'une touche fonction, et vaut faux s'il correspond au code numérique d'une touche non-fonction.

Remarque

La distinction entre touches fonctions et touches non-fonctions est introduite au chapitre 5.

2.2.2. Emission d'un son

En-tête :

bip (x, y)

Arguments :

x : integer ;

y : integer ;

Résultat : -

Spécification :

La procédure BIP émet un son d'une intensité **x** pendant un laps de temps **y**.

2.2.3. Déplacements du curseur à l'écran

En-tête :

deplacement (ch, ccurseur, espacecurseur, deborde)

Arguments :

ch : char ;

ccurseur : posecran ;

espacecurseur : deplacementcurseur ;

Résultats :

ccurseur : posecran ;

deborde : boolean ;

Spécifications :

La procédure DEPLACEMENT gère la position du curseur à l'écran selon le code de la touche fonction (de déplacement) **ch** qui lui est transmis :

étant donné l'espace de déplacement du curseur **espacecurseur** et la position du curseur **ccurseur** avant l'appel de la procédure,

si le déplacement demandé entraîne le curseur en dehors de l'espace autorisé,

alors **deborde** vaut vrai et la position du curseur est inchangée,

sinon **deborde** vaut faux et la position du curseur à l'écran est modifiée et enregistrée dans la structure **ccurseur**.

Remarque

les déplacements suivants sont implémentés :

- déplacement d'un caractère vers la gauche : touche "←",
- déplacement d'un caractère vers la droite : touche "→",
- déplacement d'une ligne vers le haut : touche "↑",
- déplacement d'une ligne vers le bas : touche "↓",
- positionnement du curseur en début de ligne : touche "HOME",
- positionnement du curseur en fin de ligne : touche "END",
- positionnement du curseur en haut de l'espace de déplacement du curseur: touche "PAGEUP",
- positionnement du curseur en bas de l'espace de déplacement du curseur : touche "PAGEDOWN".

2.2.4. Fonctions d'édition

2.2.4.1. Effacement du caractère sous le curseur

En-tête :

delcar (curs)

Argument :

curs : posecran ;

Résultat : -

Spécifications :

La procédure DELCAR efface à l'écran le caractère à la position enregistrée dans la structure **curs**.

La position du curseur est inchangée.

2.2.4.2. Insertion d'une ligne à la position du curseur

En-tête :

ligneinsertion (curs)

Argument :

curs : posecran ;

Résultat :

curs : posecran ;

Spécifications :

La procédure LIGNEINSERTION insère une ligne vide (c'est-à-dire une ligne d'espaces) à l'écran avant la ligne contenant le curseur :

la ligne contenant le curseur (dont la position est enregistrée dans **curs**), et les lignes sous celle-ci sont décalées d'une ligne vers le bas,

ensuite, le curseur est positionné sur la première colonne de la ligne insérée et sa nouvelle position est enregistrée dans **curs**.

2.2.4.3. Effacement du caractère à gauche du curseur

En-tête :

bscar (curs, deborde)

Argument :

curs : posecran ;

Résultats :

curs : posecran ;

deborde : boolean ;

Spécifications :

(La procédure BSCAR a pour effet d'effacer à l'écran le caractère positionné à gauche de la position curs -s'il y en a un)

Si la position enregistrée dans **curs** ne correspond pas à la première colonne de l'espace de déplacement à l'écran,

alors **deborde** vaut faux, le caractère à gauche du curseur est effacé de l'écran, le curseur est déplacé d'une colonne vers la gauche et sa nouvelle position est enregistrée dans la structure **curs**,

sinon **deborde** vaut vrai et la position **curs** du curseur est inchangée.

2.2.4.4. Implémentation du "return" (passage à la ligne)

En-tête :

effetreturn (**curs**, **deborde**, **espacecurseur**)

Arguments :

curs : posecran ;

espacecurseur : deplacementcurseur ;

Résultats :

curs : posecran ;

deborde : boolean ;

Spécifications :

(La procédure EFFETRETURN implémente l'effet d'un "return" à l'écran : c'est-à-dire le passage à la ligne)

Si la position du curseur enregistrée dans **curs** correspond à la dernière ligne de l'espace de déplacement du curseur,

alors **deborde** vaut vrai et la position du curseur est inchangée,

sinon **deborde** vaut faux, le curseur est positionné sur la première colonne de la ligne suivante et sa nouvelle position est enregistrée dans **curs**.

2.2.4.5. Effacement de la ligne contenant le curseur

En-tête :

efligne (curs)

Argument :

curs : posecran ;

Résultats :

curs : posecran ;

Spécifications :

(La procédure EFLIGNE efface à l'écran la ligne contenant le curseur dont la position est enregistrée dans la structure **curs**.)

Si la position enregistrée dans **curs** correspond à la première ligne de l'espace de déplacement du curseur,

alors cette ligne est effacée et le curseur est positionné sur le premier emplacement de la première ligne de l'espace de déplacement,

sinon la ligne contenant le curseur est effacée et le curseur est positionné sur le premier emplacement de la ligne précédant la ligne effacée.

La nouvelle position du curseur est enregistrée dans la structure **curs**.

2.2.4.6. Effacement de la fin de la ligne contenant le curseur

En-tête :

effinligne

Argument :

Résultat :

Spécifications :

La procédure EFFINLIGNE a pour effet d'effacer la fin de la ligne contenant le curseur à partir de la position de celui-ci.

La position du curseur est inchangée.

2.2.4.7. Ecriture d'un caractère à la position du curseur

En-tête :

ecrirecar (c, curs, dep, deborde, fonction)

Arguments :

c : char ;

curs : posecran ;

dep : deplacementcurseur ;

fonction : boolean;

Résultats :

curs : posecran ;

deborde : boolean ;

Spécifications :

Si **fonction** vaut faux

alors la procédure ECRIRECAR n'aura aucun effet,

sinon

- le caractère **c** sera affiché à la position de l'écran enregistrée dans la structure **curs**,
- si le curseur n'est pas positionné sur la dernière colonne de l'espace de déplacement **dep** :

  alors il sera déplacé d'une colonne vers la gauche, sa nouvelle position sera enregistrée dans la structure **curs**, et **deborde** vaudra faux,

  sinon la position du curseur restera inchangée et **deborde** vaudra vrai.

2.2.4.8. Edition d'un code introduit au clavier

En-tête :

editcurs (ch, fonct, ccurseur, espacecurseur, deborde)

Arguments :

ch : char ;

fonct : boolean;

ccurseur : posecran ;

espacecurseur : deplacementcurseur ;

Résultats :

ccurseur : posecran ;

deborde : boolean ;

Spécifications :

Etant donné un caractère **ch**, une variable booléenne **fonct** indiquant si le caractère correspond à une touche fonction ou à une touche non-fonction,

étant donné un espace de déplacement du curseur **espacecurseur** et la position du curseur enregistrée dans la structure **ccurseur** relative à cet espace de déplacement,

la procédure EDITCURS effectue le traitement correspondant au code numérique **ch** (appelle la fonction d'édition appropriée), renvoie la position du curseur dans la structure **ccurseur** et signale si la fonction d'édition réalisée entraînait le curseur en dehors de son espace de déplacement en mettant la variable booléenne **deborde** à vrai (sinon cette dernière prendra la valeur faux).

3. Spécifications des primitives du module gestion cadre

3.1. Types implémentés

CONSTANTE

lmligne : longueur d'une ligne

TYPES

PREPLIGNE : ↑repligne

REPLIGNE = record

numligne : integer ; {numéro de ligne}

prec : prepligne ; {pointeur vers la représentation de la ligne précédente}

suiv : prepligne ; {pointeur vers la représentation de la ligne suivante}

long : integer ; {longueur effective de la ligne représentée}

ligne : array [1..lmligne] of char ; {contenu de la ligne}

end ; {end du record repligne}

CADRE = record

noligne : prepligne ;

nocolonne : integer ;

{position du premier élément du cadre dans la fiche}

posx, posy : integer ;

{position du premier élément du cadre à l'écran}

posfx : prepligne ;

posfy : integer ;

{position du curseur sur la fiche}

espace : deplacementcurseur ;

{dimensions du cadre}

état : boolean ;

{indique si le cadre est en cours d'édition ou non}

insertmode : boolean ;

{indique le mode d'écriture de la fiche}

end ; {end du record cadre}

FICHE = record

repinterne : prepligne ;

{pointeur vers la représentation de la première ligne de la fiche}

repexterne : cadre ;

{représentation externe de la fiche}

end ; {end du record fiche}

3.2. Primitives offertes

3.2.1. Initialisation d'une fiche

En-tête :

initfiche (f, coordx, coordy, h, l)

Arguments :

f : fiche ;

coordx, coordy : integer;

h, l : integer ;

Résultat :

f : fiche ;

Spécifications :

La procédure INITFICHE initialise une structure de type fiche **f** de la manière suivante :

- les coordonnées du premier élément du cadre à l'écran sont initialisées à

  (f.repexterne.posx, f.repexterne.posy) := (**coordx**, **coordy**),

- les dimensions du cadre :

  (f.repexterne.espace.haut, f.repexterne.espace.long) := (**h**, **l**).

Les initialisations suivantes sont également effectuées :

- la position du premier élément du cadre dans la fiche :

  f.repexterne.noligne := nil,

  f.repexterne.nocolonne := 1,

- la position du curseur sur la fiche :

  f.repexterne.posfx := nil,

  f.repexterne.posfy := 1,

- l'état d'un cadre :

  f.repexterne.etat := false,

- le mode d'écriture d'une fiche : (par défaut : réécriture)

  f.repexterne.insertmode := false.

3.2.2. Affichage d'une ligne

En-tête :

affichageligne (**f**, **n**, **x**, **y**, **numcar**)

Arguments :

f : fiche ;

n : prepligne;

x, y, numcar : integer ;

Résultat : -

Spécifications :

La procédure AFFICHAGELIGNE affiche la ligne de la structure **f** pointée par **n** à partir du caractère de position **numcar** dans la ligne pointée, l'affichage à l'écran s'effectue à la ligne **x** et à partir de la colonne **y** de l'espace de déplacement du curseur.

3.2.3. Affichage du contenu d'un cadre

En-tête :

affichagecontenucadre (**f**, **curs**)

Arguments :

f : fiche ;

curs : posecran;

Résultat :

curs : posecran ;

Spécifications :

La procédure AFFICHAGECONTENUCADRE affiche le cadre attaché à la structure **f** et positionne le curseur à la position **ccurseur** (cette position correspond à la positon du curseur sur la fiche enregistrée dans le cadre attaché à la fiche).

(Remarque : le cadre sera affiché conformément aux données enregistrées dans la structure de type cadre, f.repexterne, attachée à la fiche f)

3.2.4. Ajout d'une ligne vide à la fin d'une fiche

En-tête :

ajoutligne (f, l)

Argument :

f : fiche ;

Résultats :

f : fiche ;

l : prepligne ;

Spécifications :

La procédure AJOUTLIGNE ajoute une ligne de longueur 0 à la fin de la fiche **f**, et cette ligne sera pointée par **l**.

3.2.5. Insertion d'une ligne vide dans une fiche

En-tête :

insertligne (f, pligne)

Arguments :

f : fiche ;

pligne : prepligne;

Résultats :

f : fiche ;

pligne : prepligne ;

Spécifications :

La procédure INSERTLIGNE insère une ligne de longueur 0 avant la ligne pointée par **pligne** dans la fiche **f**, après insertion **pligne** pointera vers la ligne vide insérée.

3.2.6. Suppression d'une ligne dans une fiche

En-tête :

supligne (f, pligne)

Arguments :

f : fiche ;

pligne : prepligne;

Résultats :

f : fiche ;

pligne : prepligne ;

Spécifications :

La procédure SUPLIGNE supprime la ligne pointée par **pligne** dans la fiche **f**,

si la ligne supprimée était la dernière de la fiche,

alors **pligne** est mis à nil,

sinon si la ligne supprimée était la première ligne de la fiche,

alors après suppression, **pligne** pointe vers la ligne suivante,

sinon après suppression, **pligne** pointe vers la ligne précédente.

3.2.7. Gestion des déplacements du curseur

En-tête :

gestdeplacement (ch, f, curs)

Arguments :

ch : char ;

f : fiche ;

curs : posecran;

Résultats :

f : fiche ;

curs : posecran ;

Spécifications :

La procédure GESTDEPLACEMENT gère la position du curseur à l'écran et sur la fiche **f** selon le code de la touche fonction (de déplacement) **ch** qui lui est transmis.

La position du curseur à l'écran est enregistrée dans la structure **ccurseur**, et

la position du curseur sur la fiche est enregistrée dans la structure de type cadre attachée à la fiche.

Remarque

Les déplacements implémentés sont ceux implémentés dans la gestion de l'écran.

Ces déplacements sont également explicités dans le chapitre 5, au point "4.2. Manuel de l'utilisateur (utilisation des concepts)".

3.2.8. Fonctions d'édition

3.2.8.1. Effacement du caractère sous le curseur

En-tête :

efcarsouscurs (f, curs)

Arguments :

f : fiche ;

curs : posecran;

Résultats :

f : fiche ;

curs : posecran ;

Spécifications :

La procédure EFCARSOUSCURS efface le caractère de la fiche **f** correspondant à la position du curseur à l'écran enregistrée dans la structure **curs**.

(Remarque : les caractères à droite de la position du curseur dans la fiche sont décalés d'un caractère vers la gauche.)

Cette procédure prend également en charge l'ajustement de l'affichage de la ligne à l'écran si nécessaire.

3.2.8.2. Insertion d'une ligne vide à la position du curseur

En-tête :

insertficheligne (f, curs,ajustaffichage)

Arguments :

f : fiche ;

curs : posecran ;

Résultats :

f : fiche ;

curs : posecran ;

ajustaffichage : boolean ;

Spécifications :

La procédure INSERTFICHELIGNE insère une ligne vide dans la fiche **f** avant la ligne dont la position à l'écran est enregistrée dans la structure **curs**.

La variable booléenne **ajustaffichage** vaut vrai si le cadre attaché à la fiche doit être réaffiché, et faux sinon.

3.2.8.3. Effacement du caractère à gauche du curseur

En-tête :

efcargauche (f, curs, limite, ajustaffichage)

Arguments :

f : fiche ;

curs : posecran;

limite : boolean ;

Résultats :

f : fiche ;

curs : posecran ;

ajustaffichage : boolean ;

Spécifications :

La procédure EFCARGAUCHE efface le caractère de la fiche **f** situé à gauche de la position du curseur à l'écran enregistrée dans la structure **curs**.

(Remarque : le caractère sous le curseur et les caractères à droite de la position du curseur dans la fiche sont décalés d'un caractère vers la gauche.)

Si **limite** vaut vrai, alors la variable booléenne **ajustaffichage** prendra la valeur vrai car le cadre attaché à la fiche devra être réaffiché.

3.2.8.4. Implémentation du "return" sur une fiche

En-tête :

fichereturn (f, limite, ajustaffichage, curs)

Arguments :

f : fiche ;

limite : boolean ;

curs : posecran ;

Résultats :

f : fiche ;

ajustaffichage : boolean ;

curs : posecran ;

Spécifications :

La procédure FICHERETURN implémente l'effet d'un "return" sur la fiche **f**, lorsque le curseur est positionné à l'écran à la position enregistrée dans la structure **curs**.

Si **limite** vaut vrai, alors la variable booléenne **ajustaffichage** prendra la valeur vrai car le cadre attaché à la fiche devra être réaffiché.

(Remarque : les effets d'un "return" sont différents selon le mode d'écriture de la fiche f, cfr chapitre 5)

3.2.8.5. Effacement de la ligne contenant le curseur

En-tête :

supficheligne (f, curs, ajustaffichage)

Arguments :

f : fiche ;

curs : posecran;

Résultats :

f : fiche ;

curs : posecran ;

ajustaffichage : boolean ;

Spécifications :

La procédure SUPFICHELIGNE supprime de la fiche **f** la ligne dont la position à l'écran est enregistrée dans la structure **curs**.

La variable booléenne **ajustaffichage** vaut vrai si le cadre attaché à la fiche doit être réaffiché et faux sinon.

3.2.8.6. Effacement de la ligne à partir de la position du curseur

En-tête :

finlignefichesup (f)

Argument :

f : fiche ;

Résultat :

f : fiche ;

Spécifications :

La procédure FINLIGNEFICHESUP supprime de la fiche **f** la fin de la ligne contenant la position courante à partir de celle-ci, si **f** n'est pas vide, sinon émission d'un son.

(Remarque : la position courante sur la fiche est inchangée)

3.2.8.7. Ecriture d'un caractère sur une fiche

En-tête :

ecrirefiche (f, curs, limite, ajustaffichage, ch, fonct)

Arguments :

f : fiche ;

curs : posecran ;

limite : boolean ;

ch : char ;

fonct : boolean ;

Résultats :

f : fiche ;

curs : boolean ;

ajustaffichage : boolean ;

Spécifications :

Si **fonct** vaut vrai

alors la procédure ECRIREFICHE n'aura aucun effet,

sinon le caractère **ch** sera inséré dans la fiche f à la position correspondant à la position à l'écran enregistrée dans la structure **curs**.

(Remarque : les effets seront différents selon le mode d'écriture de la fiche, cfr chapitre 5)

La variable booléenne **ajustaffichage** vaut vrai si le cadre attaché à la fiche doit être réaffiché et faux sinon.

3.2.8.8. Gestion des fonctions d'édition sur une fiche

En-tête :

gestfiche (ch, f, fonct, ccurseur)

Arguments :

ch : char ;

f : fiche ;

fonct : boolean;

ccurseur : posecran ;

Résultats :

f : fiche ;

ccurseur : posecran ;

Spécifications :

Etant donné un caractère **ch**, une variable booléenne **fonct** indiquant si le caractère correspond à une touche fonction ou à une touche non-fonction,

étant donné une fiche **f** et la position du curseur à l'écran enregistrée dans la structure **ccurseur**,

la procédure GESTFICHE coordonne les différentes procédures d'édition à l'écran et sur la fiche, et renvoie la position du curseur dans la structure **ccurseur**.

3.2.9. Edition globale d'une fiche

En-tête :

editfiche (f)

Argument :

f : fiche ;

Résultat :

f : fiche ;

Spécifications :

La procédure EDITFICHE implémente l'édition et la saisie d'une fiche **f**.

Cette procédure affiche le contenu de la fiche **f** selon les caractéristiques de la représentation externe de cette fiche enregistrée dans **f.repexterne**,

ensuite, l'introduction de la fiche est implémentée par une boucle d'acquisition des codes des touches enfoncées par l'utilisateur (cfr le manuel de l'utilisateur défini au point 4.2. du chapitre 5), et de la transformation de la fiche en fonction des codes introduits.

3.2.10. Sauvetage du contenu d'une fiche dans un fichier

En-tête :

sauvefiche (f, nomf)

Argument :

f : fiche ;

nom f : string ;

Résultat : -

Spécifications :

La procédure SAUVEFICHE effectue le sauvetage d'une structure de type fiche **f** dans un fichier de type texte de nom **nomf**.

3.2.11. Chargement d'une fiche à partir d'un fichier de type texte

En-tête :

chargerfiche (**f**, **nomf**)

Arguments :

f : fiche ;

nomf : string;

Résultat : -

Spécifications :

La procédure CHARGERFICHE charge le contenu d'un fichier de type texte de nom **nomf** dans une structure de type fiche **f**.

(Remarque : avant d'effectuer l'appel de cette procédure, il est nécessaire de procéder à l'initialisation de la fiche f à l'aide de la procédure initfiche)

3.2.12. Destruction d'une fiche

En-tête :

delfiche (**f**)

Argument :

f : fiche ;

Résultat :

f : fiche ;

Spécifications :

La procédure DELFICHE libère l'espace occupé par les représentations des lignes constituant la fiche **f**, et met le pointeur vers la première ligne de la fiche à nil.

4. Spécifications de primitives externes à gestion cadre

4.1. Affichage contour cadre

En-tête :

contourcadre (repcadre)

Argument :

repcadre : cadre ;

Résultat : -

Spécifications :

La procédure CONTOURCADRE affiche à l'écran le contour du cadre **repcadre** (Le contour est un simple trait).

4.2. Effacement d'un cadre

En-tête :

effacercadre (repcadre)

Argument :

repcadre : cadre ;

Résultat : -

Spécifications :

La procédure EFFACERCADRE efface le contour et le contenu du cadre **repcadre**.

Annexe 4.2

Implémentation de deux modules :

gestion curseur

gestion cadre

```
program essai ;
uses CRT ;

{*****************************************************************************}
{ DECLARATION DE CONSTANTES                                                   }
{*****************************************************************************}
const
      {constantes pour le module gestion écran}
      {pour les fonctions de déplacement}

            flechehaut = #72;
            flechegauche = #75;
            flechedroite = #77;
            flechebas = #80;

            home = #71;
            pageup = #73;
            toucheend = #79;
            pagedown = #81;

      {pour les fonctions d'édition : touches "non-fonction"}

            espace = #32;
            backspace = #8;
            return = #13;
            effaceligne = ^d;
            effacefinligne = ^l;

      {pour les fonctions d'édition : touches "fonction"}

            delete = #83;
            insert = #82;

      {constantes pour le module gestion cadre}

      {touches non-fonctions pour indiquer le mode d'édition}

            insmode = ^i;
            overmode = ^o;

      {pour le module gestion cadre}

            lmligne = 80; {longueur maximale d'une ligne à saisir}
```

```
{*****************************************************************************}
{ DECLARATION DE TYPES                                                        }
{*****************************************************************************}
type
    {déclaration de type pour le module gestion écran}

    posecran = record
            poscx : integer; {numéro de ligne}
            poscy : integer; {numéro de colonne}
    end;
    {enregistre la position du curseur à l'écran en coordonnées relatives à l'espace de
       déplacement du curseur}

    deplacementcurseur = record
            long : integer; {longueur de l'espace de déplacement}
            haut : integer; {hauteur de l'espace de déplacement}
    end;
    {donne l'espace de déplacement du curseur à l'écran}

    {déclaration de type pour le module gestion cadre}

    prepligne = ^repligne;

    repligne = record
            numligne : integer; {numéro de ligne}
            prec : prepligne; {pointeur vers la ligne précédente}
            suiv : prepligne; {pointeur vers la ligne suivante}
            long : integer; {longueur effective de la ligne représentée}
            ligne : array [1..lmligne] of char; {représentation de la ligne}
    end; {end du record repligne}

    cadre = record
            noligne : prepligne;
            nocolonne : integer;
                {position du premier élément du cadre dans la fiche}
            posx, posy : integer;
                {position du premier élément du cadre à l'écran}
            posfx : prepligne;
            posfy : integer;
                {position du curseur sur la fiche}
            espace : deplacementcurseur;
                {dimensions du cadre}
            etat : boolean;
                {indique si le cadre est en cours d'édition ou non}
            insertmode : boolean;
                {indique le mode d'édition du cadre}
    end; {end du record cadre}

    fiche = record
            repinterne : prepligne; {pointeur vers première ligne de la fiche}
            repexterne : cadre; {représentation externe de la fiche}
    end; {end du record fiche}
```

```
{*****************************************************************************}
{ DECLARATION DES VARIABLES DU PROGRAMME  (de test)                           }
{*****************************************************************************}
var
      car : char ;
      curseur : posecran;
      f : fiche;
      s : string;
{*****************************************************************************}
{ DECLARATIONS DES PROCEDURES                                                 }
{*****************************************************************************}
{*****************************************************************************}
{ Procédure dessinant le contour d'un cadre                                   }
{*****************************************************************************}
procedure contourcadre (repcadre : cadre);
const
      coinsupgauche = #218;
      barhorizon    = #196;
      coinsupdroit  = #191;
      barverticale  = #179;
      coininfgauche = #192;
      coininfdroit  = #217;
var
      x1, x2, y1, y2, i : integer;
begin
      x1 := repcadre.posy - 1;
      y1 := repcadre.posx - 1;
            {coin supérieur gauche du cadre}
      x2 := repcadre.posy + repcadre.espace.long + 1;
      y2 := repcadre.posx + repcadre.espace.haut + 1;
            {coin inférieur droit du cadre}
      window (x1, y1, x2, y2);      clrscr;
      window ( 1, 1, 80, 25);
      gotoxy (x1, y1);  write (coinsupgauche);
      for i := (x1 + 1) to x2 - 1 do write (barhorizon);
      write (coinsupdroit);
      for i := (y1 + 1) to (y2 - 2) do
      begin
            gotoxy (x1, i); write (barverticale);
            gotoxy (x2, i); write (barverticale);
      end;
      gotoxy (x1, y2 - 1);     write (coininfgauche);
      for i := (x1 + 1) to (x2 - 1) do write (barhorizon);
      write (coininfdroit);
      window ( 1,1,80,25);
end; {end de la procédure contourcadre}

{*****************************************************************************}
{ Procédure effaçant le contour et le contenu d'un cadre                      }
{*****************************************************************************}
procedure effacercadre (repcadre : cadre);
var x1, x2, y1, y2 : integer;
begin
      x1 := repcadre.posy - 1;       y1 := repcadre.posx - 1;
            {coin supérieur gauche du cadre}
      x2 := repcadre.posy + repcadre.espace.long + 1;
      y2 := repcadre.posx + repcadre.espace.haut + 1;
            {coin inférieur droit du cadre}
```

```
      window (x1, y1, x2, y2);      clrscr;
      window ( 1,1,80,25);
end; {end de la procédure effacercadre}
{*****************************************************************************}
{ DECLARATION DES PROCEDURES DE "GESTION CURSEUR"                             }
{*****************************************************************************}

{*****************************************************************************}
{ Procédure de lecture d'un caractère au clavier                              }
{*****************************************************************************}
procedure lirecar (var ch : char ; var fonction : boolean );
begin
      ch := readkey ;
      if ch <> #0
      then fonction := false
      else
      begin
            ch := readkey ;
            fonction := true ;
      end ;
end ;

{*****************************************************************************}
{ Procédure d'émission d'un son                                               }
{*****************************************************************************}
procedure bip (x : integer; y : integer);
begin
      sound (x);
      delay (y);
      nosound;
end;

{*****************************************************************************}
{ Procédure de déplacement du curseur                                         }
{*****************************************************************************}
procedure deplacement (ch : char; var ccurseur : posecran ;
                              espacecurseur : deplacementcurseur ;
                              var deborde : boolean);
begin
      deborde := false;
      case ch of
      flechehaut :
            if (ccurseur.poscx = 1)
            then deborde := true
            else
            begin
                dec (ccurseur.poscx);
                gotoxy (ccurseur.poscy,ccurseur.poscx);
            end;
      flechegauche :
            if (ccurseur.poscy = 1)
            then deborde := true
            else
            begin
                dec (ccurseur.poscy);
                gotoxy (ccurseur.poscy,ccurseur.poscx);
            end;
      flechedroite :
```

```
                if (ccurseur.poscy = espacecurseur.long)
                then deborde := true
                else
                begin
                    inc (ccurseur.poscy);
                    gotoxy (ccurseur.poscy,ccurseur.poscx);
                end;
        flechebas :
                if (ccurseur.poscx = espacecurseur.haut)
                then deborde := true
                else
                begin
                    inc (ccurseur.poscx);
                    gotoxy (ccurseur.poscy,ccurseur.poscx);
                end;
        home :
                begin
                    deborde := true;
                    ccurseur.poscy := 1;
                    gotoxy (ccurseur.poscy, ccurseur.poscx);
                end;
        toucheend :
                begin
                    deborde := true;
                    ccurseur.poscy := espacecurseur.long;
                    gotoxy (ccurseur.poscy, ccurseur.poscx);
                end;
        pageup :
                begin
                    deborde := true;
                    ccurseur.poscx := 1;
                    gotoxy (ccurseur.poscy, ccurseur.poscx);
                end;
        pagedown :
                begin
                    deborde := true;
                    ccurseur.poscx := espacecurseur.haut;
                    gotoxy (ccurseur.poscy, ccurseur.poscx);
                end;
        end; {end du case}
end;

{*****************************************************************************}
{ Implémentations des fonctions d'édition à l'écran                           }
{*****************************************************************************}


{*****************************************************************************}
{ Procédure effaçant le caractère sous le curseur à l'écran                   }
{*****************************************************************************}
procedure delcar (curs : posecran);
begin
        write (espace);
        gotoxy (curs.poscy, curs.poscx);
end;
```

```
{*****************************************************************************}
{ Procédure insérant une ligne la position du curseur                         }
{*****************************************************************************}
procedure ligneinsertion (var curs : posecran);
begin
      gotoxy( 1, curs.poscx);
      insline;
      curs.poscy := 1;
end;

{*****************************************************************************}
{ Procédure effaçant le caractère à gauche du curseur                         }
{*****************************************************************************}
procedure bscar (var curs : posecran ; var deborde : boolean);
begin
      if (curs.poscy = 1)
      then deborde := true
      else
      begin
            dec (curs.poscy);
            gotoxy (curs.poscy, curs.poscx);
            write (espace);
            gotoxy (curs.poscy, curs.poscx);
      end;
end;

{*****************************************************************************}
{ Procédure implémentant le return à l'écran                                  }
{*****************************************************************************}
procedure effetreturn (var curs : posecran; var deborde : boolean;
                       espacecurseur : deplacementcurseur);
begin
      if (curs.poscx = espacecurseur.haut)
      then deborde := true
      else
      begin
            deborde := false;
            clreol ;
            inc (curs.poscx);
            curs.poscy := 1;
      end;
end;

{*****************************************************************************}
{ Procédure effaçant la ligne contenant le curseur                            }
{*****************************************************************************}
procedure efligne (var curs : posecran);
begin
      delline;
      if (wherey > 1) then curs.poscx := wherey - 1 ;
      curs.poscy := 1;
      gotoxy (curs.poscy, curs.poscx);
end;
```

```
{*****************************************************************************}
{ Procédure effaçant la fin de la ligne contenant le curseur                  }
{*****************************************************************************}
procedure effinligne ;
begin
      clreol ;
end;

{*****************************************************************************}
{ Procédure écrivant un caractère à la position du curseur                    }
{*****************************************************************************}
procedure ecrirecar (c : char; var curs : posecran; dep : deplacementcurseur;
                           var deborde : boolean; fonction : boolean);
begin
      deborde := false;
      if (fonction <> true) and (c >= espace)
      then
      begin
            if (curs.poscy < dep.long)
            then
            begin
                write (c);
                inc (curs.poscy);
            end
            else if (curs.poscy = dep.long)
            then
            begin
                deborde := true;
                write (c);
                gotoxy (curs.poscy, curs.poscx);
            end;
      end;
end;

{*****************************************************************************}
{ Procédure implémentant l'ensemble des fonctions d'édition à l'écran         }
{*****************************************************************************}
procedure editcurs (ch : char; fonct : boolean; var ccurseur : posecran;
                  espacecurseur : deplacementcurseur ; var deborde : boolean);
begin
      deborde := false;
      if fonct = true
   then
      case ch of
      delete : delcar (ccurseur);
            {effacement du caractère sous le curseur}
      insert : ligneinsertion (ccurseur);
            {insertion d'une ligne vide à la position du curseur}
      end {end du case}
   else {c'est-à-dire fonct <> true}
      case ch of
      backspace : bscar (ccurseur,deborde);      {effacement du caractère à gauche du curseur}
      return : effetreturn (ccurseur,deborde,espacecurseur);      {passage à la ligne}
      effaceligne : efligne (ccurseur);      {effacement de la ligne contenant le curseur}
      effacefinligne : effinligne;      {effacement à partir de la position du curseur}
      end; {end du case}
      ecrirecar (ch,ccurseur,espacecurseur,deborde,fonct);
end;
```

```
{*****************************************************************************}
{ Procédure gérant les déplacements et l'édition                              }
{*****************************************************************************}
procedure gestcurseur (c: char; f : boolean; var curs: posecran;
                       depespace : deplacementcurseur; var limite : boolean);
begin
      if f = true then deplacement(c, curs, depespace,limite);
      editcurs (c,f,curs,depespace,limite);
end;

{*****************************************************************************}
{ DECLARATION DES PROCEDURES DE GESTION CADRE                                 }
{*****************************************************************************}

{*****************************************************************************}
{ Procédure d'initialisation d'une fiche                                      }
{*****************************************************************************}
procedure initfiche (var f : fiche ; coordx, coordy : integer ;
                     h : integer ; l : integer);
begin
      f.repinterne := nil;

      f.repexterne.noligne := nil;
      f.repexterne.nocolonne := 1;
            {position du premier élément du cadre dans la fiche (vide)}
      f.repexterne.posx := coordx;
      f.repexterne.posy := coordy;
            {position du premier élément du cadre à l'écran}
      f.repexterne.posfx := nil;
      f.repexterne.posfy := 1;
            {position du curseur sur la fiche}
      f.repexterne.espace.haut := h;
      f.repexterne.espace.long := l;
            {dimensions du cadre}
      f.repexterne.etat := false;
      f.repexterne.insertmode := false;
end; {end de la procedure initfiche}

{*****************************************************************************}
{ Procédure affichant le contenu d'une ligne du cadre                         }
{*****************************************************************************}
procedure affichageligne (f: fiche; n:prepligne; x:integer ; y:integer ;
                          numcar : integer);
var i,j : integer;
begin
      gotoxy (y,x);
      clreol;
      j:=y;
      i:=numcar;
      while (i<=n^.long) and (j<=f.repexterne.espace.long) and (i<=lmligne) do
      begin
            write (n^.ligne[i]);
            inc (i);
            inc (j);
      end;
end;
```

```
{*****************************************************************************}
{ Procédure affichant le contenu d'un cadre                                   }
{*****************************************************************************}
procedure affichagecontenucadre (f: fiche; var curs : posecran);
var
      i : integer;
      fin : boolean;
      l : prepligne;
begin
      gotoxy ( 1,1);
      clrscr;
      if f.repinterne <> nil
      then
      begin
            fin := false;
            l := f.repexterne.noligne;
      end
      else fin := true ;
      i := 1;
      while ( i<=f.repexterne.espace.haut) and ( not fin) do
      begin
            affichageligne (f,l, i,1,f.repexterne.nocolonne);
            if l^.suiv = nil
               then fin := true
               else l := l^.suiv;
            inc (i);
      end;
      curs.poscy := f.repexterne.posfy - f.repexterne.nocolonne + 1;
      curs.poscx := f.repexterne.posfx^.numligne -
                        f.repexterne.noligne^.numligne + 1;
      gotoxy (curs.poscy,curs.poscx);
end;

{*****************************************************************************}
{ Procédure ajoutant une ligne vide à la fin de la structure f                }
{*****************************************************************************}
procedure ajoutligne (var f : fiche; var l :prepligne );
var dern : prepligne;
begin
      new (l);
      dern := f.repinterne;
      if dern <> nil
     then
      begin
            while (dern^.suiv <> nil) do dern := dern^.suiv;
            l^.numligne := dern^.numligne + 1;
            dern^.suiv := l;
      end
     else
      begin
            l^.numligne := 1;
            f.repinterne := l;
      end;
      l^.suiv := nil;
      l^.prec := dern;
      l^.long := 0;
end;
```

```
{*****************************************************************************}
{ Procédure insérant une ligne vide dans la structure f                       }
{*****************************************************************************}
procedure insertligne (var f : fiche; var pligne : prepligne);
var l : prepligne;
begin
      if (f.repinterne = nil)
    then
      begin
            ajoutligne (f,l);
            pligne := f.repinterne;
      end
    else
      begin
            new (l);
            l^.numligne := pligne^.numligne;
            l^.suiv := pligne;
            l^.prec := pligne^.prec;
            l^.long := 0;
            if (f.repinterne <> pligne)
                then pligne^.prec^.suiv := l
                else f.repinterne := l;
            pligne^.prec := l;
            pligne := l;

            {changement des numéros}

            l := l^.suiv;
            while (l <> nil) do
            begin
                l^.numligne := l^.numligne + 1;
                l := l^.suiv;
            end;
      end;
end;

{*****************************************************************************}
{ Procédure supprimant une ligne de la structure f                            }
{*****************************************************************************}
procedure supligne (var f : fiche ; var pligne : prepligne);
var l : prepligne;
begin
      l := pligne;
      if (l^.prec = nil) and (l^.suiv = nil)
   then {supprimer la seule ligne de la fiche}
      begin
            f.repinterne := nil;
            f.repexterne.noligne := nil;
            pligne := nil;
      end
   else
      if (l^.prec = nil)
      then {suppression de la première ligne de la fiche}
      begin
            pligne := l^.suiv;
            f.repinterne := pligne;
            pligne^.prec := nil;
            pligne^.numligne := 1;
```

```
        end
        else if (l^.suiv = nil)
        then {suppression de la dernière ligne de la fiche}
        begin
                pligne := l^.prec;
                pligne^.suiv := nil;
        end
        else {c'est-à-dire l^.suiv et l^. prec <> nil}
        begin
                pligne := l^.prec;
                pligne^.suiv := l^.suiv;
                l^.suiv^.prec := l^.prec;
        end;
        dispose (l);

        {renumérotation des lignes}
        if (pligne <> nil) and (pligne^.suiv <> nil)
                then l := pligne^.suiv else l := nil;
        while (l <> nil) do
        begin
                l^.numligne := l^.numligne - 1;
                l := l^.suiv;
        end;
end;

{*****************************************************************************}
{ Procédure gérant les fonctions de déplacement du curseur sur la fiche       }
{*****************************************************************************}
procedure gestdeplacement (ch : char; var f : fiche; var curs : posecran);
var
        horscadre,reafficher : boolean;
        i : integer;
begin
        deplacement (ch, curs, f.repexterne.espace, horscadre);
        if (f.repinterne = nil)
    then
        begin
                bip (200,150);
                curs.poscx := 1;
                curs.poscy := 1;
                gotoxy (curs.poscy, curs.poscx);
        end
    else
        if (horscadre = false)
        then
        begin
                reafficher := false;
                case ch of
                flechehaut : f.repexterne.posfx := f.repexterne.posfx^.prec;
                flechegauche : dec (f.repexterne.posfy);
                flechedroite : inc (f.repexterne.posfy);
                flechebas :
                    if (f.repexterne.posfx^.suiv <> nil)
                    then f.repexterne.posfx := f.repexterne.posfx^.suiv
                    else {ne peut descendre plus bas dans la fiche}
                    begin
                        bip (200,150);
                        dec (curs.poscx);
```

```
                    gotoxy (curs.poscy, curs.poscx);
                end;
            end; {end du case}
    end
    else {c'est-à-dire horscadre = true}
    begin
            reafficher := true;
            case ch of
            flechehaut :
                if (f.repexterne.noligne^.prec = nil)
                then
                begin
                    bip (200, 150);
                    reafficher := false;
                end
                else
                begin
                    f.repexterne.noligne := f.repexterne.noligne^.prec;
                        {déplace le cadre sur la fiche}
                    f.repexterne.posfx := f.repexterne.posfx^.prec;
                        {déplace le curseur sur la fiche}
                end;
            flechegauche :
                if (f.repexterne.posfy = 1)
                then
                begin
                    bip (200, 150);
                    reafficher :=false;
                end
                else
                begin
                    dec (f.repexterne.nocolonne);
                        {déplace le cadre sur la fiche}
                    dec (f.repexterne.posfy);
                        {déplace le curseur sur la fiche}
                end;
            flechedroite :
                if (f.repexterne.posfy = lmligne)
                then
                begin
                    bip (200, 150);
                    reafficher := false;
                end
                else
                if (f.repexterne.posfy < f.repexterne.posfx^.long)
                then
                begin
                    inc (f.repexterne.nocolonne);
                        {déplace le cadre sur la fiche}
                    inc (f.repexterne.posfy);
                        {déplace le curseur sur la fiche}
                end
                else
                reafficher := false;
            flechebas :
                if (f.repexterne.posfx^.suiv = nil)
                then
                begin
```

```
                bip (200, 150);
                reafficher := false;
            end
            else
            begin
                f.repexterne.noligne := f.repexterne.noligne^.suiv;
                        {déplace le cadre sur la fiche}
                f.repexterne.posfx := f.repexterne.posfx^.suiv;
                        {déplace le curseur sur la fiche}
            end;
        home :
            if (f.repexterne.posfy = 1)
            then
            begin
                bip (200, 150);
                reafficher := false;
            end
            else
            begin
                if (f.repexterne.nocolonne = 1)
                then
                begin
                        reafficher := false ;
                        f.repexterne.posfy := 1
                end
                else
                begin
                        f.repexterne.nocolonne := 1;
                        f.repexterne.posfy := 1;
                end;
            end;
        toucheend :
            if (f.repexterne.posfy >= lmligne)
            then
            begin
                bip (200, 150);
                reafficher := false;
            end
            else
            begin
                f.repexterne.posfy := f.repexterne.posfx^.long + 1;
                if (f.repexterne.posfy > lmligne + 1)
                        then dec (f.repexterne.posfy);
                {positionnement du curseur en fin de ligne sur fiche}
            end;
        pageup :
            begin
               if (f.repexterne.posfx = f.repinterne)
               then
               begin
                 bip (200, 150);
                 reafficher := false;
               end
               else
               begin
                 i := 1;
                 while (i <= f.repexterne.espace.haut) and
                        (f.repexterne.posfx^.prec <> nil) do
```

```
                    begin
                        f.repexterne.posfx := f.repexterne.posfx^.prec;
                        inc (i);
                    end;
                f.repexterne.noligne := f.repexterne.posfx;
                end;
            end;
        pagedown :
            begin
                if (f.repexterne.posfx^.suiv = nil)
                then
                begin
                  bip (200, 150);
                  reafficher := false;
                  curs.poscy := f.repexterne.posfy
                                - f.repexterne.nocolonne + 1;
                  curs.poscx := f.repexterne.posfx^.numligne -
                                f.repexterne.noligne^.numligne + 1;
                  gotoxy (curs.poscy,curs.poscx);
                end
                else
                begin
                  i := 1;
                  while (i <= f.repexterne.espace.haut) and
                        (f.repexterne.posfx^.suiv <> nil) do
                  begin
                        f.repexterne.posfx := f.repexterne.posfx^.suiv;
                        inc (i);
                  end;
                f.repexterne.noligne := f.repexterne.posfx;
                end;
            end;
        end; {end du case}
end; {end du else}

{vérifie que le curseur est positionné dans les limites d'une ligne}

if (f.repexterne.posfy > f.repexterne.posfx^.long + 1)
    then f.repexterne.posfy := f.repexterne.posfx^.long + 1;

{voir si la position courante appartient toujours au cadre}

if (f.repexterne.posfy <=
    f.repexterne.nocolonne + f.repexterne.espace.long)
    and (f.repexterne.posfy >= f.repexterne.nocolonne)
then
begin
        if (horscadre = false)
        then
        begin
            reafficher := false;
            curs.poscy := f.repexterne.posfy
                          - f.repexterne.nocolonne + 1;
            gotoxy (curs.poscy, curs.poscx);
        end;
end
else
begin
```

```
                reafficher := true;
                f.repexterne.nocolonne := f.repexterne.posfy;
                if (f.repexterne.nocolonne >1) then dec (f.repexterne.nocolonne);
                if (f.repexterne.nocolonne > lmligne - f.repexterne.espace.long)
                    then f.repexterne.nocolonne := lmligne
                                                    - f.repexterne.espace.long + 1;
        end;

        {ajustement de l'affichage si nécessaire}

        if reafficher = true then affichagecontenucadre (f,curs);
end;

{*****************************************************************************}
{ Implémentations des fonctions d'édition sur une fiche                       }
{*****************************************************************************}

{*****************************************************************************}
{ Procédure effaçant le caractère sous le curseur sur la fiche                }
{*****************************************************************************}
procedure efcarsouscurs (var f : fiche; var curs : posecran);
var i : integer;
begin
        if (f.repinterne = nil)
        then bip (200,150)
        else if (f.repexterne.posfy <= f.repexterne.posfx^.long)
                then
                begin
                    i := f.repexterne.posfy;
                    while (i <= f.repexterne.posfx^.long) do
                    begin
                        f.repexterne.posfx^.ligne[i] :=
                        f.repexterne.posfx^.ligne [i+1];
                        i := i + 1;
                    end;
                    if (f.repexterne.posfx^.long > 0)
                        then dec (f.repexterne.posfx^.long);
                    affichageligne (f, f.repexterne.posfx,
                        f.repexterne.posfx^.numligne mod f.repexterne.espace.haut,
                        f.repexterne.posfy - f.repexterne.nocolonne + 1,
                        f.repexterne.posfy);
                    gotoxy (curs.poscy, curs.poscx);
                end;
end;

{*****************************************************************************}
{ Procédure insérant une ligne dans la fiche                                  }
{*****************************************************************************}
procedure insertficheligne (var f : fiche; var curs : posecran;
                                    var ajustaffichage : boolean);
begin
        if (f.repexterne.nocolonne > 1)
        then
        begin
                ajustaffichage := true;
                f.repexterne.nocolonne := 1;
        end;
        insertligne (f, f.repexterne.posfx);
```

```
        f.repexterne.posfy := 1;
end;

{*****************************************************************************}
{ Procédure effaçant le caractère à gauche du curseur                         }
{*****************************************************************************}
procedure efcargauche (var f : fiche; var curs : posecran;
                                limite : boolean; var ajustaffichage : boolean);
var i : integer;
begin
        if (f.repinterne = nil) then bip (200,150)
        else
        if (f.repexterne.posfy = 1) then bip (200,150)
        else
        begin
                i := f.repexterne.posfy - 1;
                while (i <= f.repexterne.posfx^.long) and (i>0) do
                begin
                    f.repexterne.posfx^.ligne [i] :=
                    f.repexterne.posfx^.ligne [i+1];
                    inc (i);
                end;
                if (f.repexterne.posfx^.long > 0)
                    then dec (f.repexterne.posfx^.long);
                dec (f.repexterne.posfy);
                if (limite =true)
                then
                begin
                    dec (f.repexterne.nocolonne);
                    ajustaffichage := true;
                end
                else
                begin
                    affichageligne (f, f.repexterne.posfx,
                          f.repexterne.posfx^.numligne mod f.repexterne.espace.haut,
                          f.repexterne.posfy - f.repexterne.nocolonne + 1,
                          f.repexterne.posfy);
                    gotoxy (curs.poscy, curs.poscx);
                end;
        end;
end;

{*****************************************************************************}
{ Procédure implémentant le return sur la fiche                               }
{*****************************************************************************}
procedure fichereturn (var f : fiche; limite : boolean; var ajustaffichage :
                                boolean; var curs : posecran);
var
        i,j : integer;
        nouvl : prepligne;
begin
        if (f.repinterne = nil) then
        begin
                ajoutligne (f,nouvl);
                f.repexterne.posfx := f.repinterne;
                f.repexterne.noligne := f.repinterne;
        end;
        if (f.repexterne.posfx^.suiv = nil)
```

```
        then ajoutligne (f,nouvl)
        else
        begin
                nouvl := f.repexterne.posfx^.suiv;
                if (f.repexterne.insertmode = true) then insertligne (f,nouvl);
        end;
        i:=f.repexterne.posfy;
        j:=1;
        while (i<=f.repexterne.posfx^.long) do
        begin
                nouvl^.ligne [j] := f.repexterne.posfx^.ligne [i];
                inc (i);
                inc (j);
        end;
        f.repexterne.posfx^.long := f.repexterne.posfy - 1 ;
        if (nouvl^.long < j - 1) then nouvl^.long := j - 1;
        f.repexterne.posfx := nouvl;
        f.repexterne.posfy := 1;
        if (f.repexterne.posfy < f.repexterne.nocolonne)
        then
        begin
                ajustaffichage := true;
                f.repexterne.nocolonne := 1;
        end;
        if (limite = true) then
        begin
                ajustaffichage := true;
                f.repexterne.noligne := f.repexterne.noligne^.suiv;
        end;
        if (ajustaffichage = false)
        then
        begin
                if (f.repexterne.insertmode = true)
                then
                begin
                    gotoxy (curs.poscy,curs.poscx);
                    insline;
                end;
                affichageligne (f, nouvl,curs.poscx,1,1);
                gotoxy (curs.poscy, curs.poscx);
        end;
end;

{*****************************************************************************}
{ Procédure supprimant une ligne de la fiche                                  }
{*****************************************************************************}
procedure supficheligne (var f : fiche; var curs : posecran;
                                 var ajustaffichage : boolean);
var
        i : integer;
        nouvl : prepligne;
begin
        if (f.repinterne = nil)
        then bip (200,150)
        else
        begin
           supligne (f,f.repexterne.posfx);
           nouvl := f.repexterne.posfx;
```

```
        if (nouvl = nil) then
        begin
             curs.poscy := 1;
             gotoxy (curs.poscy,curs.poscx);
        end
        else
        begin
             if (f.repexterne.posfy > f.repexterne.posfx^.long+1)
                 then f.repexterne.posfy := f.repexterne.posfx^.long + 1;
             if (f.repexterne.posfx^.numligne<f.repexterne.noligne^.numligne)
             then
             begin
                 ajustaffichage := true;
                 f.repexterne.noligne := f.repexterne.posfx;
             end;
             if (f.repexterne.posfy < f.repexterne.nocolonne)
             then
             begin
                 ajustaffichage := true;
                 f.repexterne.nocolonne := f.repexterne.posfy;
                 if (f.repexterne.nocolonne > 1)
                 then dec (f.repexterne.nocolonne);
                 if(f.repexterne.nocolonne > lmligne - f.repexterne.espace.long)
                     then f.repexterne.nocolonne := lmligne
                                                    - f.repexterne.espace.long + 1;
             end;
             if (ajustaffichage = false)
             then
             begin
                 i:=f.repexterne.posfx^.numligne mod f.repexterne.espace.haut;
                 while (i<=f.repexterne.espace.haut - 1) and
                          (nouvl^.suiv <> nil) do
                 begin
                      nouvl := nouvl^.suiv;
                      inc (i);
                 end;
                 if (i>=f.repexterne.espace.haut)
                 then
                     affichageligne (f,nouvl,f.repexterne.espace.haut,1,1);
             end;
             curs.poscy := f.repexterne.posfy - f.repexterne.nocolonne + 1;
             gotoxy (curs.poscy, curs.poscx);
        end;
    end;
end;

{*****************************************************************************}
{ Procédure supprimant la fin de la ligne courante de la fiche                }
{*****************************************************************************}
procedure finlignefichesup (var f : fiche);
begin
    if (f.repinterne = nil)
    then bip (200,150)
    else f.repexterne.posfx^.long := f.repexterne.posfy - 1;
end;
```

```
{*************************************************************************}
{ Procédure implémentant l'écriture sur une fiche                         }
{*************************************************************************}
procedure ecrirefiche (var f : fiche; var curs : posecran; limite : boolean;
                       var ajustaffichage : boolean; ch : char; fonct : boolean);
var
      i : integer;
      nouvl : prepligne;
begin
      if (fonct<>true) and (ch >= espace) then
      begin
            if (f.repinterne = nil) then
            begin
                ajoutligne (f,nouvl);
                f.repexterne.noligne := f.repinterne;
                f.repexterne.posfx := f.repinterne;
                f.repexterne.posfy := 1;
            end;
            if (f.repexterne.posfy > lmligne) or (f.repexterne.posfx^.long = lmligne)
            then
            begin
                affichageligne (f,f.repexterne.posfx,curs.poscx,
                                      1,f.repexterne.nocolonne);
                bip (200,150);
                if (f.repexterne.posfy < lmligne)
                then
                begin
                      if (curs.poscy > 1) then dec (curs.poscy);
                      gotoxy (curs.poscy, curs.poscx);
                end;
            end
            else if (f.repexterne.insertmode = false)
            then
            begin
                f.repexterne.posfx^.ligne [f.repexterne.posfy] := ch;
                if (f.repexterne.posfx^.long < f.repexterne.posfy) then
                   f.repexterne.posfx^.long := f.repexterne.posfy;
                inc (f.repexterne.posfy);
                if (limite = true)
                then
                begin
                      inc (f.repexterne.nocolonne);
                      ajustaffichage := true;
                end;
                end
                else
                begin {c'est-à-dire insertmode = true }
                   i:= f.repexterne.posfx^.long;
                   while (i>= f.repexterne.posfy) do
                   begin
                     f.repexterne.posfx^.ligne [i+1] :=
                           f.repexterne.posfx^.ligne [i];
                     dec (i);
                   end;
                   f.repexterne.posfx^.ligne [f.repexterne.posfy] := ch;
                   inc (f.repexterne.posfy);
                   inc (f.repexterne.posfx^.long);
                   if (limite = false)
```

```
                    then
                      begin
                            affichageligne (f,f.repexterne.posfx,curs.poscx,
                                                curs.poscy,f.repexterne.posfy);
                            gotoxy (curs.poscy, curs.poscx);
                      end
                      else
                      begin
                            inc (f.repexterne.nocolonne);
                            ajustaffichage := true;
                      end;
              end;
      end;
end;

{*****************************************************************************}
{ Procédure implémentant les fonctions d'édition sur une fiche                }
{*****************************************************************************}
procedure gestfiche (ch : char;var f : fiche; fonct : boolean;
                          var ccurseur : posecran );
var
      horscadre, reafficher : boolean;
begin
      reafficher := false;
      if (fonct = true)
      then
      case ch of
      delete : {effacement du caractère sous le curseur}
              begin
                  delcar (ccurseur);
                  efcarsouscurs (f, ccurseur);
              end;
      insert : {insertion d'une ligne vide à la position du curseur}
              begin
                  ligneinsertion (ccurseur);
                  insertficheligne (f, ccurseur,reafficher);
              end;
      end {end du case}
      else {c'est-à-dire fonct <> true}
      begin
              case ch of
              insmode :
                  if (f.repexterne.insertmode = true)
                     then bip (300,100)
                     else f.repexterne.insertmode := true;
              overmode :
                  if (f.repexterne.insertmode = false)
                     then bip (300,100)
                     else f.repexterne.insertmode := false;
              backspace : {effacement du caractère à gauche du curseur}
                  begin
                       bscar(ccurseur,horscadre);
                       efcargauche (f, ccurseur,horscadre,reafficher);
                  end;
              return : {passage à la ligne}
                  begin
                       effetreturn (ccurseur, horscadre, f.repexterne.espace);
                       fichereturn (f, horscadre, reafficher, ccurseur);
```

```
                end;
            effaceligne : {effacement de la ligne contenant le curseur}
                begin
                    efligne (ccurseur);
                    supficheligne (f, ccurseur, reafficher);
                end;
            effacefinligne : {effacement à partir de la position du curseur}
                begin
                    effinligne;
                    finlignefichesup (f);
                end;
            end; {end du case}
            ecrirecar (ch, ccurseur, f.repexterne.espace, horscadre, fonct);
            ecrirefiche (f, ccurseur, horscadre, reafficher, ch, fonct);
    end;
    if reafficher = true then affichagecontenucadre (f,ccurseur);
end;

{*****************************************************************************}
{ Procédure d'édition globale d'une fiche                                     }
{*****************************************************************************}
procedure editfiche (var f : fiche);
var
    finedit, fonctiontest : boolean;
    car : char;
    curseur : posecran;
begin
    window (f.repexterne.posy,f.repexterne.posx,
            f.repexterne.posy  + f.repexterne.espace.long ,
            f.repexterne.posx  + f.repexterne.espace.haut - 1);
    f.repexterne.etat := true;
    affichagecontenucadre (f,curseur);
    finedit := false;
    while not finedit do
    begin
            lirecar (car,fonctiontest);
            if fonctiontest = true then gestdeplacement (car,f,curseur);
            gestfiche (car,f,fonctiontest,curseur);
            if (car = ^z) then finedit := true;
    end;
    f.repexterne.etat := false;
    window (1,1,80,25);
end;

{*****************************************************************************}
{ Procédure de sauvetage d'une fiche dans un fichier de type texte            }
{*****************************************************************************}
procedure sauvefiche (var f : fiche; var nomf : string);
var
    fichier : text;
    pligne : prepligne;
    str : string [lmligne];
    i : integer;
begin
    assign (fichier, nomf);
    rewrite (fichier);
    if (f.repinterne <> nil)
    then
```

```
        begin
            pligne := f.repinterne;
            while (pligne <> nil) do
            begin
                i := 1;    str := '';
                while (i<=pligne^.long) do
                begin
                    str := str + pligne^.ligne[i];
                    inc (i);
                end;
                write (fichier, str);
                writeln (fichier);
                pligne := pligne^.suiv;
            end;
        end;
        write (fichier,^z);
        close (fichier);
end;

{*****************************************************************************}
{ Procédure de chargement d'une fiche d'un fichier de type texte              }
{*****************************************************************************}
procedure chargerfiche (var f : fiche; nomf : string);
var
        fichier : text;
        c : char;
        pligne, lignecour : prepligne;
        i : integer;
        t : array [1..lmligne] of char;
begin
        f.repexterne.nocolonne := 1;
        assign (fichier, nomf);
        reset (fichier);
        while (not eof (fichier) ) do
        begin
                ajoutligne (f,f.repexterne.posfx);
                i:=1;
                while (c <>#13) do
                begin
                    read (fichier,c);
                    f.repexterne.posfx^.ligne[i] := c;
                    inc(i);
                end;
                read (fichier,c);
                f.repexterne.posfx^.long := i-2;
        end;
        if (f.repinterne <> nil) then f.repexterne.noligne := f.repinterne;
end;

{*****************************************************************************}
{ Procédure de détruisant une fiche                                           }
{*****************************************************************************}
procedure delfiche (var f : fiche);
var
        pligne, lignedel : prepligne;
begin
        pligne := f.repinterne;
        if (pligne <> nil)
```

```
        then
        begin
                while (pligne^.suiv <> nil) do pligne := pligne^.suiv;
                lignedel := pligne;
                pligne := pligne^.prec;
                while (pligne <> nil) do
                begin
                    lignedel^.suiv := nil;
                    lignedel^.prec := nil;
                    dispose (lignedel);
                    lignedel := pligne;
                    pligne := pligne^.prec;
                end;
                dispose (lignedel);
                f.repinterne := nil;
        end;
        f.repexterne.noligne := nil;
        f.repexterne.nocolonne := 1;
end;
```

```
{*****************************************************************************}
{ CORPS DU PROGRAMME PRINCIPAL                                                }
{*****************************************************************************}

{ Programme permettant de tester les primitives des deux modules              }

begin
      clrscr;

      {initialisation de la fiche}

      initfiche (f,5,18,11,50);

      {fin de l'initialisation de la fiche}

      contourcadre (f.repexterne);

      window (f.repexterne.posy,f.repexterne.posx,
               f.repexterne.posy  + f.repexterne.espace.long - 1,
               f.repexterne.posx  + f.repexterne.espace.haut - 1);
      affichagecontenucadre (f,curseur);
      editfiche (f);
      effacercadre (f.repexterne);
      writeln ('donner un nom de fichier');
      read(s);
      sauvefiche (f,s);
      delfiche (f);
      initfiche (f,5,18,11,50);
      chargerfiche (f,s);
      if f.repinterne <> nil
      then
         f.repexterne.noligne := f.repinterne;
         f.repexterne.posfx := f.repinterne;
      end;
      contourcadre (f.repexterne);
      window (f.repexterne.posy,f.repexterne.posx,
               f.repexterne.posy  + f.repexterne.espace.long - 1,
               f.repexterne.posx  + f.repexterne.espace.haut - 1);
      affichagecontenucadre(f,curseur);
      editfiche (f);
      window (f.repexterne.posy,f.repexterne.posx,
               f.repexterne.posy  + f.repexterne.espace.long - 1,
               f.repexterne.posx  + f.repexterne.espace.haut - 1);
      affichagecontenucadre(f,curseur);
      repeat until keypressed;
end .□
```

Références

[Fisette85] D. Fisette,
Définition d'un langage de programmation permettant l'expression d'assertions,
mémoire de maîtrise, Namur, 1985.

[LSD80] B. Le Charlier,
Définition du langage LSD80,
Institut d'Informatique, Namur, Septembre 1980.

[PROG] Notes du cours de B. Le Charlier,
Preuves de programmes,
Institut d'Informatique, Namur.

[SYS88] M. Derroitte et B. Le Charlier,
Un système d'aide à l'enseignement d'une méthode de programmation,
Institut d'Informatique, Namur, Mai 1988.