MASTER EN SCIENCES INFORMATIQUES

Spécification, construction et validation d'un analyseur fortran interactif

Deville, Yves

Award date:
1983

Awarding institution:
Universite de Namur

Facultés Universitaires N.D. de la Paix - NAMUR

INSTITUT D'INFORMATIQUE

SPECIFICATION, CONSTRUCTION ET VALIDATION D'UN ANALYSEUR FORTRAN INTERACTIF

YVES DEVILLE

Mémoire présenté
en vue de l'obtention du grade de
LICENCIE ET MAITRE EN INFORMATIQUE

année académique 1982-1983

Je tiens à exprimer mes plus vifs remerciements à Baudouin LE CHARLIER qui a accepté de diriger ce travail. Ses critiques et conseils ont grandement contribué à la réalisation de ce mémoire. Je lui suis également très reconnaissant pour l'esprit ouvert et amical dont il a fait preuve lors de sa précieuse collaboration.

Je tiens également à remercier Monsieur A. VAN LAMSWEERDE pour le stage qu'il m'a proposé, de même que Messieurs P.N. ROBILLARD et R. PLAMONDON pour leur accueil lors de ce stage à l'Ecole Polytechnique de Montréal.

PLAN

TABLE DES MATIERES

2ème PARTIE : A P P L I C A T I O N

chapitre 0 : SCHEMACODE

chapitre I : SPECIFICATION

chapitre II : CONSTRUCTION ET VALIDATION

3ème PARTIE : C R I T I Q U E

chapitre I : SPECIFICATION

chapitre II : CONSTRUCTION ET VALIDATION

I N T R O D U C T I O N

Dans la littérature, les méthodes de programmation sont généralement illustrées par quelques exemples qui sont presque toujours et pour des raisons faciles à comprendre, de petits problèmes, aboutissant à des programmes très courts, écrits dans des langages idéalisés. Dans ce travail, nous voulons montrer qu'il est possible d'utiliser des méthodes rigoureuses de spécification et de construction de programmes pour une application de taille "réelle".

On admet qu'en moyenne, un programmeur passe au moins la moitié de son temps à corriger les erreurs qu'il a commises pendant l'autre moitié. Ceci résulte de l'attitude traditionnelle où, face à la résolution d'un problème, on s'efforce d'obtenir le plus rapidement possible un programme. On se dit alors,que s'il y a des erreurs, on s'en apercevra toujours au cours des tests et il suffira de modifier le programme à chaque erreur rencontrée, en espérant qu'il deviendra ainsi moins incorrect. Cette attitude, nous le savons, conduit à des résultats désastreux; tant au point de vue de la qualité des programmes produits que de celui du temps total de réalisation. C'est évidemment l'attitude inverse qu'il faut avoir : faire, dès le début, tout ce qui est humainement possible pour éviter de commettre des erreurs et obtenir un programme sûrement correct. Cela signifie qu'il faut s'efforcer de raisonner avec une rigueur parfaite.

Nous présenterons dans ce travail une démarche rigoureuse, mais non formelle, qui nous a permis de mener à bien une application, en augmentant la confiance accordée aux programmes, en limitant les tests empiriques à un rôle d'appoint et en réduisant la durée de réalisation.

Ce travail se compose de trois parties intitulées respectivement méthode, application et critique. Dans la première partie, sans vouloir donner LA bonne méthode de programmation, nous exposerons les idées fondamentales et principes généraux de notre démarche, de même qu'un certain nombre de méthodes qu'elle utilise. La deuxième partie montrera son application à la construction d'un analyseur Fortran interactif. Dans la troisième partie, nous comparerons, de façon critique, les principes exposés dans la première partie à certains points de vue exposés dans la littérature.

Chacune de ces trois parties s'articule selon deux axes : les spécifications, d'une part, la construction et la validation des programmes d'autre part. Ces deux axes ne sont pas indépendants, mais complémentaires. La découpe choisie nous paraît rendre l'exposé plus clair et systématique et favorise le parallélisme et les comparaisons entre les trois parties de ce travail.

Nous allons maintenant synthétiser le contenu des différentes parties. Dans le chapitre "spécification" de la première partie, nous développerons le rôle des spécifications, la nécessité de définir une "théorie" et des concepts adéquats, l'importance de la notion de représentation et l'importance de généraliser la notion de primitive. Ensuite, nous exposerons la démarche de spécification appliquée dans la deuxième partie de ce travail.

Au chapitre "construction et validation", nous préciserons d'abord le but de la validation en remarquant qu'aucune forme de raisonnement ne permet d'obtenir une certitude absolue. Ensuite, nous décrirons brièvement la notion de test. Après quoi, nous aborderons l'exécution symbolique et le calcul d'assertions, bases des démonstrations de correction. Nous exposerons plus complètement deux types de méthodes de démonstration de programmes : la méthode des assertions inductives et des raisonnements sur les données. Nous donnerons les principes de la méthode des assertions inductives, les aspects constructifs de celle-ci et l'appliquerons à un exemple élémentaire. Concernant les raisonnements sur les données, nous distinguerons les méthodes descendante et ascendante. Après avoir exposé le principe sur lequel elles se basent (induction complète), nous étudierons leurs différences, et les appliquerons au même exemple. Nous comparerons finalement la méthode des assertions inductives aux raisonnements sur les données, et nous expliquerons pourquoi nous préférerons généralement la méthode de raisonnement descendant sur les données, pour construire un programme.

L'application développée dans la seconde partie a été réalisée à l'école polytechnique de Montréal, dans le cadre du projet Schémacode dont nous décrirons brièvement les objectifs et les principes pour montrer ensuite l'intérêt d'y inclure un analyseur Fortran interactif.

Le premier chapitre, consacré aux spécifications de cet analyseur, suivra la démarche de spécification proposée dans la première partie. Nous donnerons tout d'abord une spécification "naturelle". Ensuite, nous développerons un ensemble de concepts qui nous permettront d'énoncer les spécifications "techniques" des différentes parties de l'analyseur. Enfin, nous établirons la compatibilité de la spécification naturelle et des spécifications techniques.

Dans le second chapitre, nous développerons complètement la construction et la validation d'une partie de l'analyseur : l'analyse syntaxique et sémantique des expressions. Nous y appliquerons notamment les méthodes de démonstration et de construction exposées dans la première partie de ce travail. Les autres parties de l'analyseur, quoique complètement réalisées ne seront pas décrites. Sans cela, il nous aurait été impossible de développer en profondeur la première et la troisième partie de ce travail. Le but poursuivi étant de montrer qu'il est possible d'appliquer la méthode exposée pour réaliser une application de taille "réelle", ce chapitre, malgré son volume, fait partie intégrante du travail et n'a donc pas été porté en annexe.

Le chapitre "spécification" de la troisième partie contiendra un bref exposé de la méthode de modularisation d'un système, proposée par Parnas, suivi d'une critique et d'une comparaison de celle-ci avec les principes exposés dans la première partie. Nous examinerons ensuite si notre démarche permet d'obtenir les qualités généralement exigées d'un "bon" logiciel. Après cela, nous aborderons le problème important du choix d'un langage de spécification. Nous exposerons tout d'abord le cercle vicieux de la vérité par définition. Ensuite, nous étudierons les faiblesses des langages formels de spécification et nous montrerons pourquoi leur utilisation conduit à ce cercle vicieux. Nous terminerons en exposant la différence entre formalisme et notations, ainsi que les avantages du langage naturel pour exprimer les spécifications.

Dans le chapitre "construction et validation" de cette troisième partie, nous étudierons d'abord la doctrine de la "goto less programming", née suite à un article de Dijkstra de 1968 et qui, depuis lors, a beaucoup d'adeptes. Nous la critiquerons et nous montrerons pourquoi le retrait ou l'interdiction des

branchements ne peuvent constituer une méthode de programmation. Ensuite, nous examinerons brièvement ce que recouvre le concept de programmation structurée et comment les idées exposées dans la première partie rencontrent ce concept. Ce chapitre se terminera par une critique du formalisme pour la construction et la validation des programmes. Nous étudierons successivement le formalisme de Hoare, l'idée de définir "mathématiquement" la sémantique des langages de programmation, et le "calcul des programmes" de Dijkstra.

1ERE PARTIE

METHODE

Chapitre I : S P E C I F I C A T I O N

1. : PRINCIPES GENERAUX

1.1. : Rôle des spécifications

Nous allons essayer de montrer que la spécification d'un programme n'est pas seulement une relation liant les données de ce programme à ses résultats , mais surtout que cette relation exprime exactement ce qu'il faut connaître d'un programme pour pouvoir l'utiliser.

Soit x, l'ensemble des données d'un programme et y, l'ensemble des résultats. Si l'on considère que la spécification est la relation déterminant les résultats en fonction des données, alors le texte du programme définit lui-même sa spécification.
Ainsi on peut écrire y = F(x) où F est la fonction partielle calculée par le programme. Mais cette fonction F n'est définie que par le texte du programme.

Remarquons que les renseignements apportés par la formulation y = F(x) sont d'une nature différente de la relation y = sinus(x) qui, elle, constitue une vraie spécification. Pourquoi ?
Parce que la fonction sinus est connue indépendamment du programme écrit pour la calculer. Si quelqu'un désire utiliser ce programme, il doit connaître autre chose sur la notion de sinus que le texte du programme et bien plus qu'une définition de cette fonction : il doit en connaître suffisamment de propriétés pour résoudre un autre problème pour lequel la notion de sinus est utile.

La spécification d'un programme est donc l'énoncé d'une propriété de ce programme qu'il faut savoir pour utiliser intelligemment ses résultats ou pour faire un raisonnement permettant d'utiliser ce programme pour en construire d'autres. C'est pourquoi une spécification doit avant tout être simple et compréhensible.

1.2. : Nécessité de concepts adéquats

Dire qu'une spécification d'un programme quelconque doit être simple, précise et compréhensible, c'est dire qu'elle doit l'être autant que

celle d'une opération primitive d'un langage de programmation telle que +,*,... Ainsi, si une spécification se réduit parfois à un concept familier (sinus, racine carrée, ...), le plus souvent, on devra créer des concepts propres au problème à résoudre et en faire une "théorie" qui rende ces concepts aussi familiers que ceux relatifs aux nombres entiers, à l'addition,...
Une telle théorie devra avoir un sens en dehors de tout souci de programmation.

La notion de spécification compréhensible est étroitement liée au contexte culturel; dire qu'un programme fournit le résultat sinus(x) ne sera compréhensible que pour quelqu'un possédant des notions de trigonométrie. Ce programme ne pourra être utilisé avec profit que si on s'intéresse à certains domaines où cette notion trouve des applications. Le plus souvent, ce contexte devra être créé, du moins partiellement, de toutes pièces.
La rédaction de spécifications devra donc être précédée ou accompagnée de la définition de concepts ayant des propriétés utiles pour le problème à résoudre et qui permettront d'en construire une solution.

1.3. : Importance de la notion de représentation

Il y a apparemment une objection importante à émettre au point de vue exposé ci-dessus : la variété des concepts que l'on peut imaginer pour énoncer des spécifications contraste violemment avec l'extrême pauvreté des objets disponibles dans les langages de programmation. La notion de représentation répond à cette objection. Un programme manipulera des types de données qui ont très peu de signification en eux-mêmes, mais qui représentent par la volonté du programmeur, des objets réels, concrets ou abstraits, relatifs aux concepts introduits, et d'une nature généralement étrangère à la programmation. La spécification d'un programme revient alors à citer une propriété de ces objets.

Si l'on a, par exemple, deux catégories d'objets X et Y, et une certaine relation R entre ces objets. La spécification d'un programme pourrait être :

> Si x^* est la représentation d'un objet $x \in X$,
> Alors s'il existe un objet $y \in Y$ tel que x R y, le programme fournira pour résultat une représentation y^* de y.

Pour qu'une telle spécification soit effectivement simple et compréhensible, il faut et il suffit que la relation R soit bien connue. Si R n'est connue que par une définition, le programme sera probablement inutile tant que quelqu'un n'aura pas établi une théorie mettant en lumière d'autres propriétés des objets appartenant à X ou Y, et de la relation R.

La notion de représentation permettra, une fois les relations liant les x aux x^* et les y aux y^* établies, de raisonner sur les objets x et y plutôt que directement sur leur représentation.

1.4. : Généralisation de la notion de primitive

Dans un langage de programmation, une primitive est une unité élémentaire avec laquelle on construit un programme. Une primitive est donc l'équivalent d'une brique. Elle permet d'affirmer à priori, sans démonstration, que toutes les exécutions de cette primitive ont une propriété en commun. C'est à cette propriété que se réduit la définition de la primitive, ou encore la connaissance que l'on en a. Il n'est pas nécessaire de connaître l'"algorithme" pour utiliser cette primitive. On peut supposer qu'il existe un oracle qui l'exécute et donne le résultat. Une primitive est donc une spécification sans algorithme.

Nous généraliserons cette notion de primitive; ainsi une spécification sera jugée bonne si l'on peut utiliser le programme associé à cette spécification comme une primitive lors de la construction d'un autre programme.

2. : PROPOSITION D'UNE DEMARCHE DE SPECIFICATION

2.1. : Spécification naturelle

La première phase de la résolution d'un problème consiste évidemment à identifier le problème. Dans le cas d'un problème de programmation, ce travail d'identification débouchera sur la spécification d'un programme, appelée spécification "naturelle", qui résout ce problème. Cette spécification devra donc être simple, claire, précise et devra permettre d'utiliser correctement le programme.

De plus, il est souhaitable que cette spécification soit formulée en termes facilement compréhensibles par les utilisateurs potentiels du programme, donc au moyen de concepts facilement assimilables par ces utilisateurs. Elle devra finalement refléter les exigences du demandeur sans faire pour autant de concessions à l'exigence de précision.

2.2. : Choix des concepts

Cette phase pourrait s'appeler l' "analyse" du problème. Elle consiste essentiellement à faire une "théorie", à découvrir des concepts et des propriétés de ceux-ci qui seront utiles pour résoudre le problème considéré.

Il n'existe pas de méthode générale permettant de découvrir des concepts à partir d'un énoncé de problème. En effet, la nature des problèmes est tellement diverse, que les concepts associés à ces problèmes peuvent également être de natures très différentes. La découverte des concepts et des propriétés associées est véritablement une oeuvre d'imagination et constitue la principale difficulté du problème; seules l'intuition et l'expérience pourront guider cette recherche.

Il n'existe également aucun critère absolu pour déterminer si un concept est adéquat pour résoudre un problème donné. Néanmoins, nous pourrons dire qu'un concept ou une notion est adéquat si on le comprend bien et si l'on a pu mettre en lumière certaines de ses propriétés fondamentales pour le problème à résoudre. Ces propriétés doivent servir de base aux raisonnements de construction des programmes.

Les concepts introduits reflètent la démarche du spécificateur. Ces concepts devront donc être communicables, et celui qui les reçoit devrait être capable de refaire lui-même cette démarche pour atteindre le même niveau de compréhension.

2.3. : Spécification technique

Les concepts définis, ainsi que leurs propriétés, forment une "théorie" qui va permettre de spécifier un ensemble de programmes qui seront utilisés comme primitives pour la réalisation du programme initial. Nous appellerons ces spécifications "techniques", d'une part pour les différencier des spécifications naturelles, et

d'autre part parce qu'elles font partie intégrante du processus de résolution du problème et n'ont en général pas d'intérêt pour l'utilisateur.

Remarquons qu' une fois ces primitives spécifiées, il faudra pour les résoudre refaire la même démarche : définir de nouveaux concepts pour spécifier de nouvelles primitives et ainsi de suite. L'ensemble des concepts définis aux différentes étapes détermine la structure générale des programmes dont la seule relation est : "est considéré comme primitive". C'est cette relation qui permettra de construire les programmes de façon simple, en divisant les difficultés et en raisonnant sur les objets eux-même plutôt que sur leur représentation.

2.4. : Liaison entre spécification naturelle et technique

Pour que le programme résultat respecte la spécification naturelle, il faut que l'ensemble des spécifications techniques soit compatible avec la spécification naturelle du problème. Il faudra donc établir, par un raisonnement explicite, qu'un programme respectant la spécification technique respecte aussi la spécification naturelle initiale. Ceci nécessitera la démonstration d'un certain nombre de propriétés reliant les concepts "techniques" aux concepts "naturels" proches de l'utilisateur.

2.5. : Processus d'essai et d'erreur

La démarche exposée ici est idéalisante; rien ne nous assure a priori de l'adéquation des concepts définis avec le problème à résoudre. Ces concepts pourront donc être remis en question soit lors de la spécification de primitives, soit lors de la construction de programmes.

Lorsqu'on définit un concept, c'est parce qu'on a une idée intuitive de l'utilité de celui-ci pour la suite des raisonnements. Il est donc possible que lors de la spécification de primitives, on se rende compte que certains concepts devraient être abandonnés au profit d'autres qui faciliteraient les raisonnements. De même, lorsqu'on spécifie une primitive, on pense intuitivement que l'on pourra construire un programme respectant cette spécification. Or il est possible que l'on arrive à la conclusion qu'il est

impossible de construire un tel programme, ce qui conduira à modifier la spécification, voire un concept.

On remarque donc que le choix de concepts ne peut généralement se faire sans de nombreux retours en arrière. Ce n'est qu'après un processus d'essais et d'erreurs que l'on obtiendra un ensemble de concepts satisfaisants qui permettent de résoudre le problème. En définitive, seule la connaissance intuitive du problème et l'expérience guideront la démarche effectuée et en faciliteront l'aboutissement. Cela ne veut pas dire qu'un manque d'expérience et une méconnaissance intuitive du problème empêcheront cette démarche d'aboutir, mais il sera sans doute nécessaire,dans ce cas, d'effectuer un plus grand nombre de retours en arrière avant d'obtenir les concepts adéquats.

Chapitre II : CONSTRUCTION ET VALIDATION

1. : BUT DE LA VALIDATION

1.1. : Objectif

L'objectif poursuivi est de savoir s'il existe des techniques permettant de se persuader qu'un programme que l'on a écrit a de bonnes chances d'être correct. Ceci permettrait de limiter l'importance des tests empiriques et de réduire ceux-ci à un rôle d'appoint. Nous ne développerons pas ici la notion de programme correct car ce concept est assez ambigu [LER,80]. Néanmoins, nous prendrons le point de vue suivant : un programme sera correct s'il respecte ses spécifications. Ceci nous ramène à la question de savoir ce qu'est une spécification, ce que nous avons développé précédemment.

Démontrer un programme sera donc démontrer l'absence de contradictions avec les spécifications.

1.2. : Limites

Peut-on établir avec une certitude absolue la correction d'un programme ? Non, aucun raisonnement, aucune démonstration ne permet d'obtenir une certitude absolue au sujet d'une hypothèse. En effet, si nous donnons une démonstration qui prétend qu'un programme donné est correct, encore faut-il être certain que cette démonstration est correcte. Nous devons donc fournir une preuve de la correction de cette démonstration. Ensuite, pour être tout à fait sûr, il faudrait démontrer que cette preuve est correcte, et ainsi de suite ... Il n'est pas possible de sortir de ce cercle vicieux, même avec un système formel, car si l'on arrive à démontrer un théorème dans ce système, encore faut-il démontrer que cette formule correspond bien à ce que l'on voulait démontrer et cette démonstration elle, ne sera pas formelle.

Nous pouvons comparer celui qui écrit une démonstration à quelqu'un qui cherche un objet dans une pièce [LEC,83]. Celui-ci soulève les fauteuils, fouille les armoires, regarde partout et, après avoir recommencé plusieurs fois déclare : "l'objet n'est pas ici". Mais peut-être a-t-il été seulement incapable de le

découvrir ! Cela ne veut pas dire qu'une démonstration est sans valeur, mais qu'elle est imparfaite et qu'il faut en être conscient.

Le but d'une démonstration, formelle ou non formelle, ne peut donc pas être d'établir une certitude absolue, toujours hors de portée, mais seulement d'essayer de se donner de meilleures raisons de croire à la correction d'un programme.

1.3. : Méthode de validation

Comment prouver qu'un programme est correct; ou mieux encore, comment construire un programme correct ? Nous étudierons plusieurs méthodes de validation. Tout d'abord, la méthode la plus utilisée qui consiste à tester un programme. Ensuite, nous aborderons les méthodes de démonstration de programmes basées sur deux principes : l'exécution symbolique et la démonstration par induction, sous sa forme la plus générale : l'induction complète. Selon que l'on raisonne sur un ensemble de segments de l'exécution ou sur un ensemble de données, on obtient deux familles de méthodes dont la méthode des assertions inductives et l'induction sur les données sont des représentants.

Dans tous les cas, une démonstration ne sera que l'explicitation du raisonnement que l'on ferait pour construire ou pour essayer de comprendre le programme.

2. : NOTION DE TEST

2.1. : Principe

Le but des tests est d'établir la présence d'erreurs dans un programme. Le processus de test se décompose en 3 phases.
La première consiste à choisir judicieusement un ou plusieurs jeux de données de test; ensuite il faut exécuter le programme avec ces jeux de données, et enfin comparer les résultats obtenus avec les résultats attendus pour en déduire la présence d'erreurs éventuelles.

2.2. : Qu'est-ce qu'un bon jeu de tests?

Nous n'avons pas la prétention de répondre à cette question,

mais de donner quelques éléments de réponse. Remarquons tout d'abord qu'il est impossible d'effectuer des tests exhaustifs. Le problème consiste à partitionner l'ensemble des données possibles en classes. Ces classes doivent être choisies de telle façon que l'on puisse trouver un représentant de chaque classe et que si l'exécution du programme fournit des résultats corrects pour le représentant d'une classe, on puisse avoir de bonnes raisons de croire que le programme est correct pour toutes les données de cette classe. La notion de bon jeu de tests est donc opposée à celle de jeu de tests aléatoire.

La détermination des données tests peut se faire à 3 moments distincts de la conception d'un ensemble de programmes : spécification, algorithme et intégration.

2.3. : Tests construits à partir des spécifications

A partir des spécifications, on peut définir un premier ensemble de jeux de données tests. Pour ce faire, on essaye de discerner les classes qui apparaissent dans les spécifications. Il suffira alors de déterminer un jeu de tests qui couvre chacune des classes identifiées, de même que les "cas limites" des différentes classes. Enfin, il faudra définir les résultats attendus pour les jeux de tests choisis.

Les jeux de tests obtenus ici sont généralement appelés tests "black box" [VAN,81]; montrant ainsi que l'on ne se préoccupe pas de la stratégie de résolution du problème, mais que l'on s'attache uniquement aux spécifications de celui-ci.

2.4. : Tests construits à partir de l'algorithme

On peut définir un second ensemble de jeux de données tests à partir de la structure algorithmique d'un programme. Le principe est d'associer une classe à chaque chemin d'exécution. Malheureusement, cela devient rapidement complexe, surtout si le programme contient des boucles. Ainsi, on se contentera de couvrir un ensemble "représentatif " de chemins. Une méthode consiste à déterminer un ensemble de chemins appelés élémentaires couvrant toutes les combinaisons possibles des issues des tests de l'algorithme ainsi que 0 ou 1 itération de chaque boucle.

Ensuite, il faudra déterminer les conditions qui doivent être initialement vraies en début de programme pour que ces chemins soient parcourus. Ces conditions permettent ainsi de choisir des données tests pour chaque chemin déterminé.

Les jeux de tests obtenus ici sont généralement appelés tests "white box" [VAN,81]; montrant ainsi que l'on se préoccupe de la stratégie de résolution du problème.

2.5. : Test d'intégration

Les jeux de tests définis jusqu'à présent ne permettent de détecter des erreurs qu'au sein d'un programme et non d'un ensemble de programmes. Il reste donc à tester la compatibilité des interfaces, les séquences d'invocation, la coopération entre programmes,...

Nous retiendrons ici la méthode de test incrémental [VAN,81], appelée également test"bottom up" dans la mesure où l'on teste un programme et ses combinaisons avec tous les programmes qui ont déjà été testés. Mais par quel programme commencer ?
Généralement, on commencera par les programmes de niveau le plus bas et par ceux considérés comme les plus critiques dans la phase d'intégration.

2.6. : Autres tests

Les tests explicités ci-dessus ne constituent qu'une petite partie des tests que l'on peut effectuer. En effet, nous pourrions encore citer les tests de performances relatifs à la complexité du programme, aux données, temps réel,...
Signalons aussi les tests d'acceptation, d'utilisabilité, d'installation,...

3. : EXECUTION SYMBOLIQUE

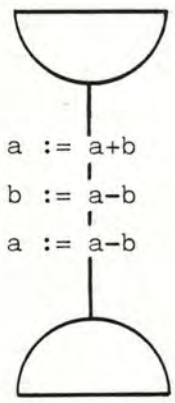
3.1. : Principe

Cette méthode ressemble à l'exécution du programme à la main pour établir une propriété de celui-ci. Mais on établit une propriété valable pour une infinité de valeurs initiales des variables et non pour une valeur spécifique.

Pour exécuter symboliquement un organigramme sans boucle, il faut :

- représenter les valeurs initiales des variables par des symboles, éventuellement soumis à certaines conditions initiales.
- exécuter symboliquement l'organigramme, dans tous les cas de figures possibles (branches de condition) en tenant compte des conditions initiales. Les calculs sont faits sur les symboles et non sur des valeurs numériques.
- vérifier que, dans tous les cas et en tenant compte des conditions initiales, les valeurs finales des variables vérifient la spécification.

3.2. : Exemple



Spécification : ce programme a pour effet de permuter les valeurs des variables a et b, de type entier (pourvu que celles-ci aient reçu préalablement une valeur).

démonstration :

Soit a_o, b_o, les valeurs initiales de a et b.

initialement nous avons : $a = a_o$ et $b = b_o$

exécutons la 1° instruction $a = a_o+b_o$ et $b = b_o$

2° instruction $a = a_o+b_o$ et $b = (a_o+b_o)-b_o = a_o$

3° instruction $a=(a_o+b_o)-a_o$ et $b = a_o$

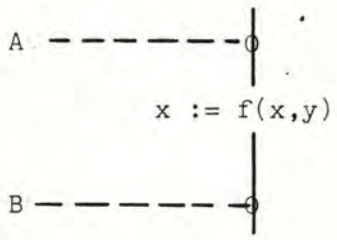
nous obtenons finalement $a = b_o$ et $b = a_o$

Ce qui respecte bien la spécification.

Cette méthode est d'un intérêt limité en elle-même; elle est assez lourde à utiliser et n'est applicable que pour des programmes sans boucles. Néanmoins, cette méthode peut servir d'auxiliaire aux autres méthodes de démonstration.

3.3. : Calcul des assertions

Outre l'exécution symbolique, on peut utiliser le calcul des assertions. Nous définirons une assertion comme étant l'expression d'une relation entre les valeurs courantes des variables [LER,78]. La différence entre les calculs d'assertions et l'exécution symbolique est que les premiers portent sur des ensembles d'états et non sur des états. Montrons cela sur une instruction d'affectation.



Soit x, une variable de programme
y, l'ensemble des autres variables

3.3.1. : évaluation progressive

Si on a une assertion au point A de la forme générale $\alpha(x,y)$, l'assertion la plus forte que doivent vérifier les variables au point de sortie B tenant compte de l'assertion au point d'entrée se notera $sp(x:= f(x,y),\alpha(x,y))$ (sp pour "strongest postcondition") et sera: $\exists x' : \{x = f(x',y) \text{ et } \alpha(x',y)\}$
Le quantificateur traduit le fait que l'instruction d'affectation a pour effet de détruire l'ancienne valeur de x. Celui-ci pourra être souvent supprimé lorsque l'ancienne valeur est fonction de la nouvelle.

Remarquons qu'il existe un ensemble de règles de calcul pour chaque type d'instruction [FLO,67].

3.3.2. : évaluation régressive

Si on a une assertion au point B de la forme générale $\beta(x,y)$, l'assertion à attacher au point A est déterminée de la manière suivante : la condition la moins forte, portant sur les valeurs des variables au point d'entrée pour que $\beta(x,y)$ soit vérifiée au point B

se notera wp(x:= f(x,y),ß(x,y)) (wp pour "weakest precondition") et sera ß(f(x,y),y).

Il existe également un ensemble de règles de calcul pour la transformation d'assertions pour chaque type d'instruction [HOA,69], [DIJ,75].

4. : ASSERTIONS INDUCTIVES

4.1. : Correction partielle et totale

Un programme est partiellement correct
Ssi soit son exécution se termine et produit des résultats conformes aux spécifications
soit son exécution ne se termine pas et ne produit donc pas de résultats.
Un programme est totalement correct
Ssi il est partiellement correct et termine toujours son exécution (sous les conditions initiales prévues par les spécifications).

4.2. : But de la méthode

La méthode des assertions inductives permet de démontrer la correction partielle d'un programme; la correction totale devant être démontrée séparément en prouvant la terminaison. Cette méthode possède également un aspect constructif; non pour fournir un moyen "automatique" de construction à partir de la spécification d'un programme, mais en offrant un moyen pour passer d'une idée intuitive de résolution à un programme résolvant le problème dans tous les cas.

4.3. : Démonstration a posteriori

4.3.1. : Principe de la méthode

Nous appellerons PRE, l'assertion dénotant la précondition; c'est à dire les conditions imposées par la spécification aux valeurs initiales des variables. Nous appellerons POST, l'assertion dénotant la post condition; c'est à dire les conditions imposées par la spécification aux valeurs finales des variables.

Démontrer la correction d'un programme S revient à établir :

{PRE} S {POST}

ce qui signifie : "Si S se termine, alors toute exécution de S à partir d'un état initial satisfaisant PRE produit un état final satisfaisant POST" [FLO,67] , [HOA,69].

Pour établir cela, le principe est de sélectionner un certain nombre de points de l'organigramme et d'attacher à chacun d'eux une assertion. On démontre ensuite que ces assertions sont vérifiées lors de tout passage de l'exécution par ces points.

4.3.2. : Détermination des assertions

4.3.2.1. : programme sans boucle

A partir de PRE, on va essayer de déterminer des assertions intermédiaires pour finalement obtenir POST à la fin du programme. Ceci sera fait d'amont en aval, par évaluation progressive des assertions.

Une seconde méthode consiste à partir de POST et de déterminer des assertions intermédiaires pour finalement obtenir PRE au début du programme. Ceci sera fait d'aval en amont, par évaluation régressive des assertions.

4.3.2.2. : programme avec boucle

Il faut sélectionner sur l'organigramme un ensemble de points suffisant pour "couper" toutes les boucles. A chacun de ces points, on attache une assertion appelée assertion inductive, que l'on se donne a priori.

Soit Ass_{i-1} et Ass_i, les assertions déterminées ou a déterminer avant et après la boucle. Il faut que l'assertion inductive I soit telle que: - $Ass_{i-1} \Rightarrow I$

- I et non C $\Rightarrow Ass_i$
- {I et C} S {I}

avec C, la condition de terminaison de la boucle

S, le corps de la boucle .

Pour montrer la 3ème condition, il faut donc réappliquer la méthode des assertions inductives pour le corps de la boucle. Cette condition exprime que l'assertion I doit être vérifiée à chaque passage.

Remarquons que l'on pourrait attacher une spécification à certains morceaux de programme; démontrer séparément la correction de ces morceaux et ensuite les considérer comme des primitives dans la démonstration du programme global.

Comment trouver I ? Il n'existe pas de méthode permettant de déterminer automatiquement une telle assertion; il faut la "deviner" et, de plus, ce choix n'est en général pas unique. L'assertion I traduit la stratégie de la résolution; elle caractérise un résultat partiel après un nombre quelconque d'itérations ou encore la situation générale qui est réalisée avant chaque tour de boucle quelque soit le numéro de celui-ci.

4.3.3. : Terminaison

La terminaison d'un programme se montre avec une technique inductive qui est, en fait, un cas particulier du raisonnement sur les données.

4.3.3.1. : relation bien fondée

Soit E , un ensemble
< , une relation binaire sur E
$x_1,\ldots,x_i,\ldots$ une suite d'éléments de E.

On dira que cette suite est décroissante
Ssi $x_1>x_2>\ldots>x_i>\ldots$
On dira que < est une relation bien fondée sur E, ou encore, que E est bien fondé par <
Ssi il n'existe aucune suite infinie décroissante d'éléments de E.

Soit X, un sous-ensemble de E
l, un élément de X

On dira que l est un élément minimal de X (pour la relation <)
Ssi $\forall x \in X, x \not< l$.
On pourrait montrer que < est bien fondée sur E
Ssi tout sous-ensemble non vide de E a un élément minimal.

4.3.2.2. : application à la terminaison

Pour démontrer la terminaison d'un programme partiellement correct, il suffit de :

- choisir un ensemble de points de coupure tel que chaque boucle soit coupée au moins une fois .
- associer à chaque point de coupure A une fonction f_A qui associe aux données et aux valeurs des variables (soit x cet ensemble) un élément d'un ensemble E, bien fondé par la relation < .
- vérifier que pour tout cycle d'exécution α de A à A on a $f_A(\alpha(x))<f_A(x)$ où $\alpha(x)$ représente l'ensemble des données et des valeurs des variables après exécution de α.

Dans la plupart des cas, on prendra E = N, ensemble des entiers positifs et il suffira de trouver $f_A(x)$ telle que :

- $f_A(x)\geq 0$ pour tout x
- toute exécution du corps de la boucle fait décroître la valeur prise par $f_A(x)$.

4.4. : Aspect constructif

Notre but n'est pas d'exposer ici les méthodes existantes de construction de programme s'inspirant de la méthode des assertions inductives, mais d'en donner quelques notions rudimentaires pour permettre de construire un programme de façon plus rigoureuse.

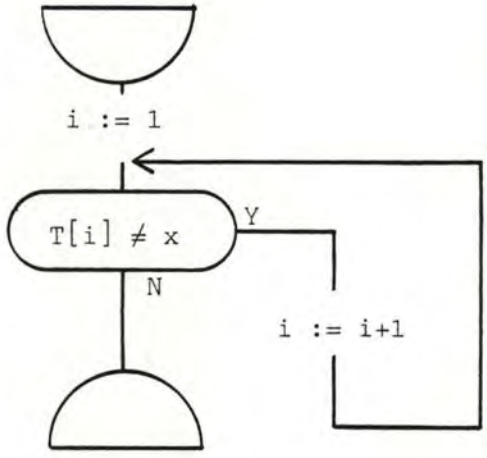
Une méthode constructive permet de dériver une séquence, une conditionnelle ou une boucle à partir des spécifications. Mais, il ne faut pas oublier que, écrire un programme revient à formaliser une idée intuitive de solution. Ainsi, lorsque la forme d'une assertion à obtenir suggère l'emploi d'une boucle pour réaliser l'objectif, un procédé de construction basé sur un raisonnement par induction pourrait être le suivant :

- Trouver une situation générale qui mette en évidence ce qu'il reste à faire pour atteindre le résultat. Cette assertion traduira donc le choix de stratégie de solution.
- Déterminer le test de fin et les initialisations à effectuer permettant de considérer la situation initiale et la situation finale comme cas particuliers de la situation générale.
- Dériver un corps de boucle tel que après exécution on "se rapproche" de la solution tout en restant dans la situation générale.

Une telle méthode devrait permettre d'obtenir un programme correct par construction. En pratique, l'utilisation de cette méthode permet d'obtenir des programmes facilement démontrables par la méthode des assertions inductives. Il est donc préférable de construire un programme comme cela plutôt que "à l'oeil" et essayer de le démontrer a posteriori; les résultats seraient plus compliqués à démontrer; les invariants difficiles à exprimer de même que la fonction de terminaison.

4.5. : Exemple de démonstration

Spécification du problème :
Soit T, un tableau de n éléments (n>0)
x, un élément de T



PRE : {x ∈ T[1:n]}
POST : {x ∉ T[1:i-1] et x = T[i]}

Déterminons un invariant I représentant la situation générale après un nombre quelconque d'itérations.

I : {x ∉ T[1:i-1] et x ∈ T[i:n]}

1) {PRE} i:=1 {I}

. évaluation progressive :

{x ∈ T[1:n] et i = 1}
-> {x ∈ T[i:n] et i = 1}
-> {x ∉ T[1:i-1] et x ∈ T[i:n]}

. évaluation régressive :

{x ∉ T[1:1-1] et x ∈ T[1:n]}
<- {x ∈ T[1:n]}

2) $\underline{I}$ et $(T[i]=x)$ $\rightarrow$ $\underline{POST}$

$\{x \notin T[1:i-1]$ et $x \in T[i:n]$ et $x = T[i]\}$

$\rightarrow \{x \notin T[1:i-1]$ et $x = T[i]\}$

3) $\{\underline{I}$ et $(T[i] \neq x)\}$ i:= i+1 $\{\underline{I}\}$

. évaluation progressive :

$\{\exists i_o : i = i_o+1$ et $x \notin T[1:i_o-1]$ et $x \in T[i_o:n]$ et $x \neq T[i_o]\}$

$\rightarrow \{x \notin T[1:i-2]$ et $x \in T[i-1:n]$ et $x \neq T[i-1]\}$

$\rightarrow \{x \notin T[1:i-1]$ et $x \in T[i:n]\}$

. évaluation régressive :

$\{x \notin T[1:i+1-1]$ et $x \in T[i+1:n]\}$

$\leftarrow \{x \notin T[1:i-1]$ et $x \neq T[i]$ et $x \in T[i:n]\}$

4) Terminaison

Soit f(T,x,i) = n-i; nous avons alors

. $\underline{I} \rightarrow x \in T[i:n]$

$\rightarrow n-i \geq 0$

. chaque exécution du corps de la boucle fait décroitre strictement la valeur prise par f.

remarque :

Que penser du choix entre évaluation progressive et évaluation régressive ? L'évaluation régressive ne fait pas intervenir de quantificateurs, mais grâce aux simplifications, ceux-ci disparaissent la plupart du temps. De plus, lorsqu'une instruction est un appel à un sous-programme, l'évaluation progressive semble plus aisée car plus en rapport avec le cheminement intellectuel humain; de plus, la substitution régressive telle que fomalisée par Hoare [HOA,71] pour les appels de procédures est assez complexe et difficile à utiliser en pratique. Remarquons encore que l'on pourrait faire de l'exécution symbolique, qui serait alors assez proche d'une évaluation progressive sans quantificateur mais avec des symboles représentant les valeurs des variables.

5. : RAISONNEMENT SUR LES DONNEES

5.1. : Induction complète

5.1.1. : Principe

Soit E, un ensemble bien fondé par la relation <, et p, un prédicat défini sur E. Supposons que l'on puisse déduire p(x) de l'hypothèse que p(y) est vrai pour tout y<x; et cela, quel que soit x ∈ E. Alors on peut conclure que p(x) est vrai pour tout x ∈ E.

Ceci s'exprime également de la manière suivante ;

de ∀ x (∀ y (y<x => p(y)) => p(x)

on peut déduire : ∀ x p(x)

en effet :

Soit X, l'ensemble des éléments de E ne vérifiant pas p. Si X n'est pas vide, il existe un élément x_o de X minimal vu que E est bien ordonné. Tous les éléments qui minorent strictement x_o sont en dehors de X et vérifient donc p. Il suit donc que x_o vérifie p, ce qui est contradictoire avec $x_o \in X$. X est donc vide.

5.1.2. : Application

En général, on appliquera ce principe de la façon suivante :

- $p(x_o)$ est vrai pour tout élément minimal x_o de E
- si x n'est pas minimal, il suffit pour que p(x) soit vrai, que p(y) soit vrai pour un certain nombre de y strictement inférieurs à x. On peut donc supposer par induction que tous ces y vérifient p et on en conclut que p(x) est vrai.

5.1.3. : Remarques

1. Le principe d'induction complète n'affirme pas qu'on doit utiliser chacune des hypothèses p(y) pour tout y<x dans la démonstration de p(x), mais seulement qu'on peut les utiliser.
2. L'ensemble des entiers positifs est bien fondé par la relation d'ordre habituel. Cet ensemble est fréquemment utilisé lors de démonstrations de programmes. Le choix de cet ensemble n'est pas toujours le meilleur pour faciliter les démonstrations.

3. Le principe de démonstration par récurrence sur les entiers est un cas particulier du principe de l'induction complète sur les entiers positifs. En effet, le principe de récurrence consiste à affirmer :

pour que p(n) soit vrai pour tout n entier positif,
il suffit que : -p(o) le soit ;
-pour tout n, p(n+1) soit vrai chaque fois que p(n) l'est.

5.2. : Démonstration a posteriori

5.2.1. : principe

Le but est de prouver la correction du programme, non pour quelques spécimens de données, mais pour toutes données initiales. Pour prouver cela, il faut trouver une relation bien fondée sur l'ensemble des données qui permettra de raisonner de manière inductive pour le problème posé. Il faut montrer que le programme est correct pour les données les plus élémentaires, et qu'il est correct pour une donnée de n'importe quel degré de complexité pourvu qu'il soit correct pour toutes données de complexité inférieure. Par induction, nous montrons donc qu'il est correct pour toutes données.

5.2.2. : application

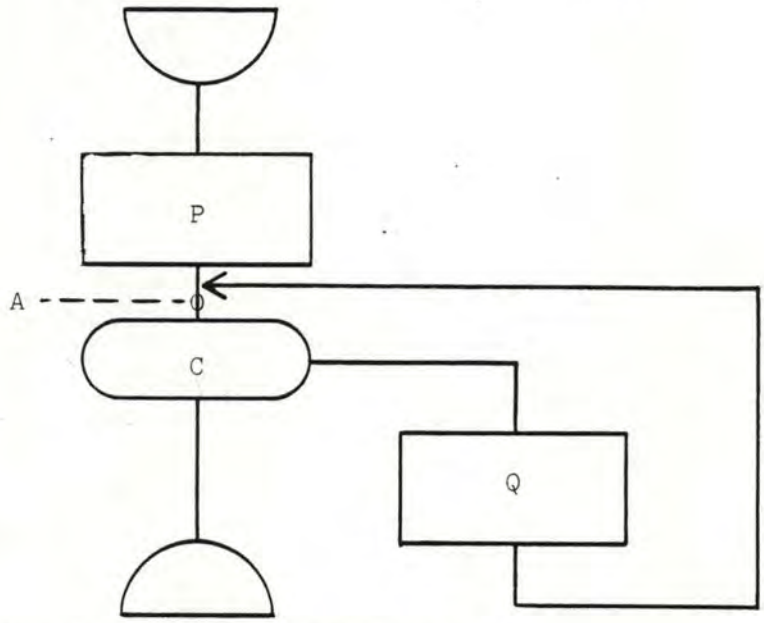
Associons à la structure des données un ensemble E, bien fondé par une relation <. Il faut démontrer que pour tout élément x de E, il existe un segment de l'exécution qui vérifie telle propriété dépendante de x; ou encore qu'il existe un passage à un point A de l'algorithme tel que cette propriété soit vrai.

Remarquons que cette technique n'exige pas de démonstration de terminaison car elle montre la correction totale du programme.

5.2.3. : raisonnement descendant et ascendant

La démonstration d'un programme par induction sur la structure des données peut se faire selon deux approches; la première est appelée raisonnement descendant et la seconde, raisonnement

ascendant. Ces approches se distinguent par le choix de la relation bien fondée < dans E et par la propriété à démontrer. La différence est le point de vue adopté. Nous allons essayer d'expliquer les différences sur le schéma de programme suivant :



5.2.3.1. : propriété à démontrer

Descendant : s'il se produit un passage au point A avec telle propriété, alors l'exécution du programme se termine avec telle autre propriété.
Ou encore : Tout suffixe de l'exécution, issu du point A, avec telle propriété, est fini et se termine avec telle autre propriété.

Ascendant : Il se produit un passage au point A avec telle propriété.
Ou encore : Il existe un préfixe de l'exécution, aboutissant au point A tel que, après son exécution on ait telle propriété.

5.2.3.2. : point de vue adopté

Descendant : La question que l'on se pose est : qu'est ce qu'il reste à faire ? Ainsi l'exécution d'un tour de boucle même d'un élément "compliqué" à un élément "simple" puisqu'il reste moins à faire. Comme on considère ce qu'il reste à faire, l'exécution d'un tour réduit ce qui reste à faire.

Ascendant : La question que l'on se pose est : qu'est ce qui a déjà été fait ? Ainsi, l'exécution d'un tour de boucle même d'un élément "simple" à un élément "compliqué" puisque l'on a "fait plus". Comme on considère l'exécution depuis le début, l'exécution d'un tour de boucle supplémentaire conduit à quelque chose de plus "compliqué"; plus proche du résultat final.

5.2.3.3. : induction

Dans les deux cas, on établit d'abord la propriété pour les éléments "les plus simples" ou éléments minimaux, pour la relation bien fondée considérée. Cette relation sera différente selon l'approche choisie : descendante ou ascendante. Ensuite, on montre que la propriété est vraie pour un élément "un peu plus compliqué" (successeurs), sachant qu'elle est vraie pour les éléments "un peu plus simples"(prédécesseurs).

5.2.3.4. : choix de la propriété

Descendant : la propriété est une formulation précise de ce qui reste à faire. Cela revient à sélectionner une partie du programme (généralement le programme sans les initialisations) auquel on associe une spécification qui est en quelque sorte une généralisation de celle du programme de départ. En effet, pour démontrer une proposition, il est souvent nécessaire d'en démontrer une beaucoup plus générale, mais cependant plus simple car plus facile à démontrer. Pour cette méthode, on considère donc les boucles comme des procédures récursives (d'un type très particulier).
La correction du programme initial est prouvée en montrant que, après l'exécution des initialisations on se trouve dans un cas particulier du cas général démontré .

Ascendant : la propriété est une formulation précise de ce qui a été fait. Dans ce cas, la partie du programme considérée pourrait être le programme sans le test de fin de boucle. Pour démontrer la propriété pour l'élément minimal, on utilisera le résultat de l'exécution de l'initialisation. Dans la méthode descendante, on utilisait le test de fin à cet effet. Pour démontrer la correction du programme, on utilise la propriété démontrée pour le cas particulier ou le test de fin est vrai.

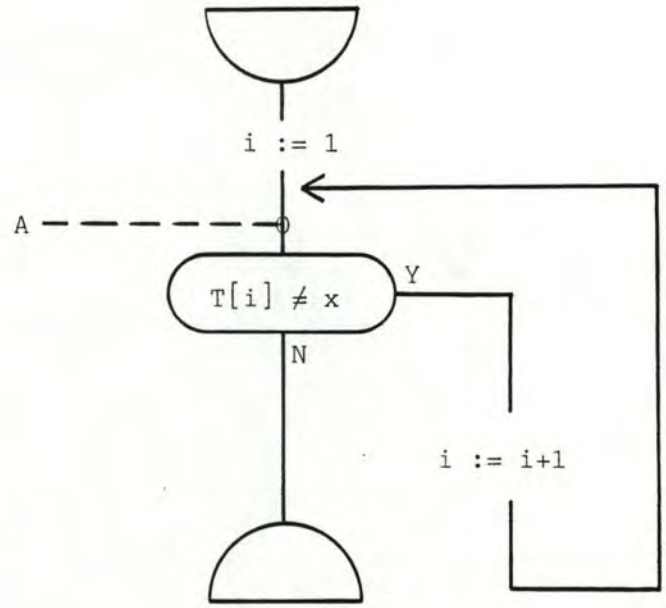
5.3. : Aspect constructif

Le raisonnement sur la structure des données se prête particulièrement à la construction d'un programme. En effet, on peut mettre en évidence la structure des objets et déterminer un ensemble bien ordonné relatif à cette structure de données. Comme pour construire un programme, il faut posséder une idée intuitive de solution, on cherchera à la formuler avec précision selon l'optique "ce qu'il reste à faire" ou "ce qui a déjà été fait". Dans la première optique, on spécifiera un programme auxiliaire capable de "faire" ce qu'il "reste à faire" dans tous les cas. Ensuite, on déterminera un ensemble de conditions suffisantes à imposer au programme auxiliaire pour qu'il respecte ces spécifications. Il suffira alors de construire ce programme auxiliaire de telle sorte qu'il respecte ces conditions.

Dans la seconde optique, on devra exprimer la suite des situations représentant ce qui a été fait et en déduire le terme générique. Ce terme exprimera de manière générale ce qui a déjà été fait.

5.4. : Exemple de démonstration

Spécification du problème :
Soit T, un tableau de n éléments (n>0)
x, un élément de T.
Déterminer i, le plus petit indice pour lequel T[i] = x.
Soit k, cet indice.



5.4.1. : Raisonnement descendant

Nous raisonnerons sur l'ensemble [1:k] bien fondé par la relation d'ordre habituelle >,sur les entiers. L'élément minimal est donc k.

La proposition à démontrer est :

$\forall\, i_o \in [1:k] : P(i_o)$

où $P(i_o)$ est : S'il se produit un passage au point A avec $i = i_o$ alors l'exécution de programme se termine avec i = k.

ou encore: Tout suffixe de l'exécution, issu du point A, avec $i = i_o$, est fini et se termine avec i = k.

1) vrai pour $i_o = k$

En effet, s'il se produit un passage au point A avec i = k, l'exécution se terminera avec i = k puisque T[k] = x.

2) si vrai pour i_o+1 alors vrai pour i_o $(1 \leqslant i_o < k)$

S'il se produit un passage en A avec $i = i_o$, l'exécution repassera en A avec $i = i_o+1$ car $i_o < k$ et donc $T[i_o] \neq x$. Par hypothèse d'induction, l'exécution du programme se terminera avec i = k.

corollaire : Le programme est correct.

Vu les initialisations, il se produira un passage au point A avec i = 1. Par conséquent, puisque $1 \in [1:k]$, l'exécution se terminera avec i = k d'après la proposition P(1).

5.4.2. : Raisonnement ascendant

Nous raisonnerons sur l'ensemble [1:k] bien fondé par la relation d'ordre habituelle <,sur les entiers. L'élément minimal est donc 1.

La proposition à démontrer est :

$\forall i_o \in [1:k] : P'(i_o)$

où $P'(i_o)$ est : Il se produira un passage au point A avec $i = i_o$

ou encore : Il existe un préfixe de l'exécution, aboutissant au point A tel que, après exécution de ce préfixe, $i = i_o$

1) vrai pour $i_o = 1$

En effet, après exécution de l'instruction i:=1, il se produira un passage au point A avec i = 1.

2) si vrai pour $i_o - 1$ alors vrai pour i_o $(1 < i_o \leq k)$

Par hypothèse d'induction, il se produira un passage au point A avec $i = i_o - 1$ et donc $T[i_o - 1]$ existe et est différent de x.

Par conséquent, l'exécution repassera au point A avec $i = i_o$.

corollaire : Le programme est correct.

D'après la proposition P(k), il se produira un passage au point A avec i = k. Alors, puisque T[k] = x, l'exécution se terminera avec i = k .

6. : COMPARAISON ASSERTIONS INDUCTIVES - RAISONNEMENT SUR LES DONNEES

6.1. : Assertions inductives et raisonnement ascendant

Nous allons essayer de montrer ici le lien qu'il peut exister entre un raisonnement ascendant et la méthode des assertions inductives.

Soit X, l'ensemble des données et des variables d'un programme. Une démonstration par raisonnement ascendant prendra la forme suivante :

Pour tout $x \in X$, il existe un préfixe p de l'exécution du programme tel que P(x).

Soit $\mathcal{P}$, l'ensemble des préfixes d'un programme. Cet ensemble est bien ordonné par la relation suivante :

$\forall q, q' \in \mathcal{P}$ $q<q'$ Ssi q est un préfixe propre de q'.

Une démonstration par assertion inductive est une démonstration par récurrence sur la longueur de l'exécution; récurrence sur le nombre de passages en un point A du programme. On définira par exemple l'ensemble $\mathcal{P}$ comme étant celui de tous les préfixes de l'exécution aboutissant à un point déterminé A du programme. Ensuite, parmi tous ces préfixes, on en choisit un, généralement l'élément maximal de $\mathcal{P}$ tel que P(q) entraine la correction du programme. Pour choisir ce préfixe, il faut pouvoir démontrer son existence et cela par ce que nous avons appelé une démonstration de terminaison.

Avec les assertions inductives, on montre que pour tout passage au point A, P(q) est vrai. Ainsi la propriété (appelée assertion) sera triviallement vrai si l'exécution n'atteint jamais le point auquel elle est attachée. Par contre, pour le raisonnement ascendant, on montre l'existence d'un passage au point A tel qu'une propriété P(x) soit vrai.

Un raisonnement ascendant peut donc être considéré comme un raisonnement par assertion inductive dans lequel la démonstration de terminaisaon et la démonstration de correction partielle sont faites simultanément.

Remarquons que la propriété P(i) d'un raisonnement ascendant peut toujours fournir un invariant qui sera de la forme $\exists$ i:P(i).

6.2. : Induction par sous-objectifs

Nous venons de montrer le parallélisme qui pouvait exister entre la méthode des assertions inductives et le raisonnement ascendant. Il devrait donc exister une méthode opposée aux assertions inductives qui serait le pendant du raisonnement descendant. Cette méthode est connue sous le nom d'induction par sous-objectifs (Subgoal Induction [MOR,77]).

Si les assertions inductives sont basées sur une induction sur le nombre d'itérations depuis le début de l'exécution l'induction par sous-objectifs se base, quant à elle, sur le nombre d'itérations pour que le programme s'arrête.

On raisonnera donc sur un ensemble S de suffixes de l'exécution en montrant que pour tout suffixe $s \in S$, l'exécution du programme ne peut se terminer qu'avec une propriété P(x)vérifiée. On démontre ensuite la correction partielle du programme en appliquant ce résultat au plus grand suffixe de S correspondant à l'exécution du programme tout entier. Cette fois la démonstration de terminaison porte sur l'existence d'un élément minimal de S.

Un raisonnement par induction par sous-objectifs peut être considéré comme un raisonnement descendant dans lequel au aurait dissocié la terminaison; c'est à dire que l'on ne fait pas intervenir la structure d'ordre associée aux différentes itérations.

7. : CHOIX D'UNE METHODE

7.1. : Limite des tests

La différence essentielle entre les tests et une démonstration de programme est que les premiers établissent la présence d'erreurs, tandis que la seconde établit l'absence d'erreurs. De plus, le processus de tests est intrinsèquement incomplet et fastidieux si l'on veut obtenir un degré raisonnable de confiance. Les tests sont avant tout un processus destructif. Ainsi, en cas d'erreur, un test fera généralement apparaître les symptômes et les conséquences d'une ou de plusieurs erreurs. Par contre, une démonstration a posteriori d'un programme aura plutôt tendance à déterminer les causes d'une erreur.

7.2. : Limite d'une démonstration

Les techniques de démonstration présentent de nombreuses limites. Tout d'abord, démontrer un programme un tant soit peu complexe est une tâche souvent difficile. La difficulté réside essentiellement dans le choix des invariants ou de l'ensemble bien fondé à considérer et de l'expression de la propriété à montrer. De plus, ce travail sera rendu plus complexe par les exécutions symboliques à effectuer si celles-ci sont longues et présentent peu de simplifications. Cet effort sera d'autant plus difficile si l'on s'encombre d'un formalisme astreignant.

Rappelons encore ici qu'une démonstration permet d'obtenir de meilleurs raisons de croire à la correction du programme; c'est-à-dire qu'il respecte ses spécifications. Mais rien n'est prouvé quant à la correction des spécifications. Il faudrait donc encore prouver que la formulation effectuée du problème représente effectivement le problème initial et pas un autre problème.

7.3. : Avantage d'une méthode constructive

La démonstration a posteriori d'un programme donné présente quelques limites supplémentaires. En effet, une démonstration peut ne pas aboutir pour de diverses raisons non aisément discernables. Le programme peut contenir une erreur; dans ce cas, comment la détecter et comment la corriger ? Ensuite, une démonstration peut elle-même contenir une erreur et enfin, l'invariant ou la propriété choisie peut être inadéquat. Tout ceci pourrait remettre en question la possibilité de démonstration face à un programme mal construit.

Nous préférerons donc les aspects constructifs des méthodes de démonstration. Ainsi ces techniques de démonstration feront partie intégrante du processus de construction. La construction du raisonnement se fera en même temps que le programme, et même avant. L'objectif unique est la simplicité des raisonnements et de l'algorithme. Nous considérons qu'un bon programme est celui dont la démonstration de correction est la plus simple, parce que

c'est celui qui a la plus de chance d'être correct et celui dont le fonctionnement est le plus facile à comprendre [LER,78].

7.4. : Problème de la construction

Le problème de la construction se pose au moment où l'on a non seulement l'énoncé du problème à résoudre, mais également une bonne idée de la succession des opérations à effectuer pour y parvenir. Une bonne idée est une idée intuitive assez précise pour qu'on puisse l'appliquer soi-même "à la main" dans n'importe quel cas.

La difficulté principale se situe avant la construction; c'est la nécessité de faire une "théorie" ou un modèle des objets dont le programme va manipuler des représentations.
Cette "théorie" doit fournir une connaissance de ces objets et donc mettre en lumière toutes les propriétés utiles pour la construction du programme. C'est de ces propriétés que naîtrons les idées intuitives de résolution.

La difficulté propre à la programmation sera de construire, à partir de ces idées, un algorithme assez rigoureux pour pouvoir être exécuté aveuglément. C'est ici que se situe l'effort d'abstraction qui permettra de construire le raisonnement et donc le programme à l'aide d'une des méthodes constructives.

Parce qu 'une démonstration, aussi rigoureuse ou formalisée soit-elle ne permet pas d'obtenir une certitude absolue, les raisonnements seront donc aidés et corrigés par des tests empiriques effectués sur le programme pour augmenter le caractère fiable du programme.

7.5. : Construction par raffinements successifs

Face à un problème complexe, on ne sait exprimer en une seule fois un algorithme résolvant ce problème. Pour vaincre cette complexité, on peut le décomposer en sous-problèmes plus simples. On écrit ainsi l'algorithme général à l'aide de ces primitives dont on se contente d'indiquer les spécifications et pour lesquels on construit ensuite les algorithmes de résolutions. Ces algorithmes pourront à leur tour faire appel à d'autres primitives et ainsi de suite. A chaque raffinement , on vérifie que l'algorithme est correct si l'on admet la correction des primitives restant à construire.

7.6. : Choix d'une méthode constructive

L'induction sur les données possède l'avantage intrinsèque de montrer la correction totale d'un programme, tandis que la récurrence sur la structure de l'exécution nécessite une démonstration supplémentaire de terminaison. La méthode des assertions inductives est souvent préférée car elle semble plus naturelle. En effet, elle systématise la "méthode" de vérification qui consiste à suivre le déroulement de l'exécution depuis le début; tandis que l'expression de "ce qu'il reste à faire" n'est pas un réflexe naturel mais qui s'acquiert facilement avec la pratique.

Le raisonnement sur les données se base sur le principe d'induction complète qui est extrêmement général. Il oblige seulement à considérer des ensembles bien fondés, ce qui est une condition beaucoup plus faible que celle que l'on s'impose généralement (ensemble totalement ordonné des nombres entiers). De plus, il n'impose aucune restriction quant à la nature de l'ensemble sur lequel on raisonne, contrairement à la méthode des assertions inductives. En général, pour un problème complexe, les types de données d'un langage de programmation seront insuffisants; on définit alors un ensemble d'objets sur lequel on raisonnera et dont les types de données du langage seront une représentation.

Nous préférerons donc les raisonnements sur la structure des données car il faut encore remarquer que lorsqu'on a un problème à résoudre et pas de programme, mieux vaut raisonner dans les termes du problème que sur la structure de l'exécution d'un programme qui n'existe pas encore. Cette approche évite de s'enfermer trop tôt dans des optiques de programmation que l'on risquerait de regretter. Mieux vaut donc raisonner sur les concepts du problème plutôt que sur l'exécution.

Parmi les raisonnements ascendants et descendants, nous choisirons plutôt les raisonnements descendants. Il est en général plus facile d'exprimer "ce qu'il reste à faire" que "ce qui a déjà été fait"; de plus, les raisonnements de construction seront souvent plus simples. Il faut néanmoins remarquer que pour certains problèmes, il est difficile d'exprimer "ce qu'il reste à faire" sans tenir compte de "ce qui a déjà été fait".

Pour d'autres, il sera plus aisé de trouver un invariant plutôt qu'une propriété exprimant "ce qui a été fait" vu qu'un invariant est une propriété plus générale. Ainsi, on préférera parfois une démonstration par assertions inductives à une démonstration par raisonnement ascendant si on peut trouver une propriété qui ne fait pas intervenir explicitement la structure d'ordre sur laquelle est basée l'itération et que, d'autre part, on peut démontrer facilement la terminaison.

En conclusion, on utilisera la méthode qui offre les raisonnements les plus simples, clairs et précis, sachant que, dans beaucoup de problèmes, les raisonnements sur les données sont les mieux adaptés pour la construction des programmes. Enfin un programme construit avec une méthode de construction sera plus facilement démontrable a posteriori et devra, de toute façon, être testé.

2EME PARTIE

APPLICATION

Chapitre 0 : S C H E M A C O D E

1. : Objectifs de Schémacode

L'école polytechnique de Montréal a développé un outil d'aide à la conception, la construction et la documentation d'un programme [ROB,81a et b], [ROB,82a et b]. Cet outil,appelé Schémacode, est un système interactif qui assiste le programmeur dans la construction d'une version abstraite de son programme en appliquant une démarche par raffinements successifs [VAN,82a]. Schémacode est conçu non seulement pour aider l'utilisateur à écrire un programme de façon structurée, mais aussi pour favoriser la documentation simultanée de ce programme. Il permet enfin, à partir d'une description schématique du programme obtenue à l'aide d'un éditeur de structure, de créer un programme Fortran contenant une documentation insérée aux points adéquats du programme et reflétant le raisonnement descendant de l'utilisateur.

2. : Principes du système

Le scénario de construction d'un programme peut être décrit comme suit : l'utilisateur formule dans le raffinement 0 les étapes de base de résolution correspondant aux sous-problèmes qu'il a identifiés. A chacun de ces sous-problèmes correspond un nouveau raffinement i qu'il faudra décrire et redécomposer en sous-problèmes plus fins. Tout sous-problème suffisamment simple sera écrit en instructions de base Fortran : déclarations, affectations, entrées-sorties, appels de procédure,... . Pour décrire l'enchainement des sous-problèmes au sein d'un raffinement, l'utilisateur dispose d'un pseudo-langage graphique appelé Pseudocode Schématique (SPC) dans lequel on retrouve les structures de contrôles classiques : séquentielle, conditionnelle et itérative. Chacune d'entre elles est représentée sous une forme graphique suggestive.

Le langage utilisé pour construire un programme est en fait la réunion de deux langages : le pseudo-langage SPC et le langage Fortran, appelé langage cible. Ces deux langages sont utilisés simultanément lors de la construction d'un programme.

Chaque structure séquentielle SPC est en fait une instruction de base du langage cible,sauf dans le cas de la description d'un sous-problème. De plus, le corps des structures de contrôle SPC conditionnelle et itérative est une suite de structures séquentielles SPC. Enfin, les conditions des structures conditionnelles SPC et les conditions de sorties des structures itératives SPC doivent être sous la forme d'expressions logiques du langage cible.

La construction du programme terminée, un codeur effectue automatiquement la traduction de celui-ci dans le langage cible. Schémacode intègre ainsi les différents raffinements; la description des sous-problèmes sert de commentaire reflétant ainsi le raisonnement descendant de l'utilisateur et les différentes structures SPC sont traduites dans le langage cible. Dans les dernières versions de Schémacode, l'utilisateur peut choisir son langage cible (FORTRAN 66, FORTRAN 77 et PASCAL), devra donc écrire finalement les instructions élémentaires dans ce langage cible et utilisera le codeur approprié à ce langage.

3. : Intérêt d'un analyseur Fortran interactif

Pour permettre la construction d'un programme, Schémacode contient un éditeur dit éditeur de structure par opposition aux éditeurs lignes car l'unité de base de cet éditeur est la structure SPC. Outre les fonctions classiques d'édition, celui-ci assure le formatage ainsi que certaines vérifications syntaxiques des structures SPC (itération sans condition de sortie, corps de conditionnelle vide,...). Malheureusement aucune vérification n'est effectuée sur les instructions écrites dans le langage cible. Ainsi toute erreur ne sera détectée qu'à la phase de compilation du programme, après traduction en langage cible par le codeur. L'utilisateur devra accéder au code produit et refaire la démarche inverse du codeur pour déterminer dans le programme SPC, la position de cette erreur.

Pour éviter cette perte de temps et pour aider le plus possible l'utilisateur, il nous a donc semblé utile d'écrire un analyseur interactif qui effectuerait certaines vérifications des instructions écrites dans le langage cible. Dans cette optique, nous avons développé un analyseur pour le langage cible FORTRAN 66 et nous avons limité l'analyse aux instructions de déclaration , d'affectation et aux expressions logiques associées aux

structures conditionnelles et itératives SPC. Après introduction par l'utilisateur de chaque instruction de ce genre, l'analyseur vérifie que celle-ci obéit bien aux règles syntaxiques du langage cible FORTRAN 66 et qu'elle est en accord avec l'ensemble des déclarations présentes dans le programme. Si une erreur est détectée, l'analyse de cette instruction s'arrête et l'utilisateur est averti de la nature de cette erreur.

Les contraintes imposées pour le développement de cet analyseur interactif Fortran ont été les suivantes : il a dû être écrit en moins de 3 mois, en Fortran 66 et à l'aide de l'outil Schémacode sur un IBM 4341.

Chapitre I : S P E C I F I C A T I O N

1. : SPECIFICATION NATURELLE

1.1. : Préliminaires

Dans ce qui suit, nous supposerons que le lecteur possède une connaissance suffisante du FORTRAN 66 (appelé aussi FORTRAN IV). De plus, nous faisons l'hypothèse que la norme Fortran 66 détermine de manière unique l'ensemble des programmes acceptables par les compilateurs Fortran 66.

1.2. : Programme compilable

Considérons un programme Fortran comme étant composé d'une suite d'instructions. Une instruction est une suite non vide de lignes de 80 caractères dont toutes les lignes, sauf la première, ont un caractère de continuation situé dans la colonne 6.

Nous dirons qu'un programme P est compilable SSi (abréviation de si et seulement si) il est accepté par un compilateur Fortran 66.

1.3. : Compilabilité

1.3.1. : Suite de déclarations compilable

Une déclaration est une instruction dont les premiers caractères de la première ligne sont soit INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, COMMON ou EXTERNAL.

Soit S, une suite de déclarations.
Nous dirons que S est une suite de déclarations compilable SSi il existe un programme P compilable commençant par S.

1.3.2. Instruction compilable

Soit S, une suite de déclarations compilable

< dec >, une déclaration

< aff >, une affectation

< cond >, une condition

Nous dirons que

- < dec > est une déclaration compilable par rapport à S
  SSi S < dec > est une suite de déclarations compilable .
- < aff > est une affectation compilable par rapport à S
  SSi il existe un programme compilable contenant < aff > et ayant S pour suite de déclarations.
- < cond > est une condition compilable par rapport à S
  SSi il existe un programme compilable contenant une instruction de la forme IF (< cond >) α, et ayant S pour suite de déclarations. (α étant une suite quelconque de caractères.)

1.4. : Ecarts par rapport à la norme

1.4.1. : Ecarts dus aux compilateurs existants

Il existe des différences significatives entre les différents compilateurs Fortran 66 existants; ceux-ci s'éloignent plus ou moins de la norme de 1966.

Nous allons spécifier les différences que nous prendrons en compte par rapport à la norme.

- Un identificateur pourra avoir jusqu'à 8 caractères (6 selon la norme) étant donné que beaucoup de compilateurs acceptent plus de 6 caractères.
- Le nombre d'indices d'un tableau sera limité à 7 (et non 3 selon la norme) pour rencontrer la plupart des compilateurs.
- Un indice pourra être une expression arithmétique quelconque de type entier ou réel vu que cette notion se généralise parmi les compilateurs.
- Le premier mot d'une déclaration pourra être suivi de *< entier > pour spécifier la longueur des variables déclarées (exemple : REAL*10). Ceci est nécessaire car l'analyseur est prévu pour analyser des instructions Fortran de type IBM.
- Une expression de la forme A**B**C est équivalente à A**(B**C) et non à (A**B)**C comme le spécifie la norme car tous les compilateurs font de même.

1.4.2. : Restrictions destinées à faciliter l'analyse

Nous allons également restreindre quelque peu certains aspects du langage Fortran 66 pour en faciliter l'analyse mais sans contraintes exagérées pour l'utilisateur.

- Nous exigerons qu'au moins un blanc soit introduit lorsque deux symboles qui se suivent sont chacun, soit un identificateur (ou mot réservé), soit une constante entière, réelle ou double précision. Cette mesure, quoique restrictive, favorise la lisibilité des programmes.
- Nous supprimerons la notion de déclaration implicite. Tous les tableaux, variables ou fonctions référencés devront être déclarés. Cette contrainte, exigée pour la plupart des langages, devrait conduire le programmeur à écrire des programmes plus clairs.
- Les mots réservés INTEGER, REAL, DOUBLE, PRECISION, COMPLEX, LOGICAL, EXTERNAL, COMMON ne pourront être utilisés comme nom de variable , tableau , fonction, common, external et label common.
- Le nom d'un label common ne peut apparaître qu'une fois. L'ensemble des commons appartenant au même label common devront être déclarés ensemble.

1.4.3. : Restrictions dues au caractère interactif de l'analyseur

Vu le caractère interactif de l'analyseur et vu que seuls certains types d'instructions seront analysées, certaines vérifications ne seront pas effectuées.

- Dans un common, le nombre d'indices d'un tableau ne sera pas comparé avec le nombre d'indices de sa déclaration.
- Nous accepterons les déclarations de tableaux avec des variables entières comme bornes, sans savoir si celles-ci font partie des paramètres formels de l'instruction SUBROUTINE car nous n'analyserons pas ce type d'instruction. Néanmoins, nous exigerons que ces variables entières soient déclarées avant l'instruction déclarant le tableau.

- Comme nous ne distinguons pas la notion de déclaration de variable de celle de déclaration de fonction, certaines instructions non compilables seront néanmoins considérées comme telles par l'analyseur. Ainsi, par exemple, après la déclaration de INTEGER F,I ,l'instruction I = F + F(2) sera acceptée.

1.5. : Spécification de l'analyseur Fortran

A tout instant, l'analyseur mémorise une suite de déclarations compilable . Soit S : $< dec_1 >< dec_2 > \ldots < dec_n >$ cette suite.

L'analyseur Fortran effectue 2 types de traitements :

- Vérifier qu'une déclaration, affectation ou condition est compilable par rapport à S.
- Supprimer une déclaration de la suite mémorisée S.

D'une manière plus précise :

• A partir d'une instruction accompagnée de son genre (déclaration, affectation ou condition), et d'un ordre de traitement,
l'analyseur Fortran détermine si celle-ci est compilable par rapport à S. De plus, si c'est une déclaration < dec > compilable par rapport à S, la suite de déclarations mémorisée sera S< dec >.

• A partir d'une instruction accompagnée de son genre (déclaration, affectation ou condition), et d'un ordre de destruction,
Si l'instruction est une déclaration $< dec_i > \in S$,
Alors la suite de déclarations mémorisée par l'analyseur deviendra $< dec_1 > \ldots < dec_{i-1} >< dec_{i+1} > \ldots < dec_n >$ pour autant que celle-ci soit compilable.
Sinon l'exécution ne modifiera pas l'état de l'analyseur.

Remarques :

- Le seul cas où la suite $< dec_1 > \ldots < dec_{i-1} >< dec_{i+1} > \ldots < dec_n >$ ne sera pas compilable est lorsqu'on détruit une déclaration contenant une variable simple entière apparaissant comme dimension dans une déclaration de tableau .
- Cette spécification ne constitue pas un mode d'emploi de l'analyseur au sein de Schémacode. Ce mode d'emploi faisant partie intégrante de celui de Schémacode, il ne saurait être exposé sans référence à l'utilisation de Schémacode.

2. : CONCEPTS

2.1. : Notion de construction

2.1.1. : Caractère

Nous utiliserons l'ensemble des caractères suivants :

< caractère > :: = 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/ /O/1/2/3/4/5/6/7/8/9/(/)/+/-/*/⁄/,/./=/'

2.1.2. Symbole de base

On appelera symbole de base toute suite de caractères définie par la grammaire ci-dessous.

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

< identificateur > :: = < lettre > / < identificateur >< lettre >/
< identificateur >< chiffre >

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

< chiffre > :: = O/1/2/3/4/5/6/7/8/9

< constante > :: = < entier >/< réel >/< double précision >/
< logique >

< entier > :: = < chiffre >/< entier >< chiffre >

< réel > :: = < nombre décimal >/< entier >E< exposant >/
< nombre décimal >E< exposant >

< double précision > :: = < entier >D< exposant >/
< nombre décimal >D< exposant >

< logique > :: = .TRUE./.FALSE.

< nombre décimal > :: = < entier >./.< entier >/
< entier >. <entier >

< exposant > :: = < entier 2 >/+< entier 2 >/- <entier 2 >

< entier 2 > :: = < chiffre >/< chiffre >< chiffre >

< symbole spécial > :: = +/-/*/⁄/**/=/(/)/,/'/.EQ./.NE./.LE./
.GT./.LT./.GE./.OR./.AND./.NOT./
< mot réservé >

< mot réservé > :: = INTEGER/REAL/COMPLEX/DOUBLE/PRECISION/
LOGICAL/EXTERNAL/COMMON

Restrictions :

- Un identificateur est long de 8 caractères au plus.
- Les mots réservés INTEGER, REAL, ..., COMMON ne sont pas des identificateurs.

2.1.3. : Construction

2.1.3.1. : Définition

Nous appellerons construction Fortran ou construction toute suite finie de symboles de base définie par la grammaire ci-dessous

< construction > :: = < affectation >/< condition >/< déclaration >

2.1.3.2. : Expression

< expression > :: = < expression simple >/< expression arithmétique >/
< expression logique >

expression simple :

< expression simple > :: = < exp designation >/< appel de fonction >/< constante >/(< complexe >)/(< expression >)

< exp designation > :: = < nom var simple >/< nom var indicée >

< nom var simple > :: = < identificateur >

< nom var indicée > :: = < nom tableau >(< liste indices >)

< nom tableau > :: = < identificateur >

< liste indices > :: = < indice >/< indice >, < liste indices >

< indice > :: = < expression arithmétique >

< appel de fonction > :: = < nom fonction >(< liste par.fonction >)

< nom fonction > :: = < identificateur >

< liste par.fonction > :: = < par.fonction >/ < par.fonction >,< liste par. fonction >

< par. fonction > :: = < expression >/< nom procédure >/ < nom fonction >/< nom tableau >

< nom procédure > :: = < identificateur >

< complexe > :: = < sign réel >,< sign réel >

< sign réel > :: = < réel >/+< réel >/-< réel >

restriction

Le nombre d'indices d'une variable indicée est limité à 7.

expression arithmétique

```
< facteur > :: = < expression simple >/< expression simple >**
                 < facteur >
< terme > :: = < facteur >/< terme >< op mult >< facteur >
    < op mult > :: = */
< expression arithmétique > :: = < terme >/< op add >< terme >/
                                 < expression arithmétique >< op add >
                                 < terme >
    < op add > :: = +/-
```

expression logique

```
< proposition atomique > :: = < expression arithmétique >/
                              < expression arithmétique >< op rel >
                              < expression arithmétique >
    < op rel > :: = .EQ./.NE./.LT./.GE./.LE./.GT.
< négation > :: = < proposition atomique >/.NOT.< proposition atomique>
< conjonction > :: = < négation >/< conjonction >.AND.< négation >
< expression logique > :: = < conjonction >/< expression logique >.OR.
                            < conjonction >
```

2.1.3.3. : Affectation

```
< affectation > :: = < exp designation >=< expression >
```

2.1.3.4. : Condition

```
< condition > :: = < expression logique >
```

2.1.3.5. : Déclaration

```
< déclaration > :: = < decl variable >/< decl external >/
                     < decl common >
    < decl variable > :: = < type >< longueur >< liste variables>>
        < type > :: = INTEGER/REAL/DOUBLE PRECISION/COMPLEX/
                      LOGICAL
        < longueur > :: = *< entier >/< vide >
            < vide > :: =
```

< liste variables > :: = < variable dec.>/< variable dec.>
< liste variable >

< variable dec.> :: = < nom var simple >/< nom fonction >/
< nom tableau >(< decl bornes >)

< decl bornes > :: = < borne >/< borne >,
< decl bornes >

< borne > :: = < entier >/< nom var simple >

Restriction :

Le nombre de bornes d'une déclaration de tableaux est limité à 7.

< decl external > :: = EXTERNAL < liste fct et proc >

< liste fct et proc > :: = < nom fct ou proc >/
< nom fct ou proc >,< liste fct et proc >

< nom fct ou proc > :: = < nom fonction >/< nom procédure >

< decl common > :: = COMMON < liste common >/
COMMON < liste common >< liste blocs common >/
COMMON < liste blocs common >

< liste blocs common > :: = < bloc common >/
< bloc common >< liste blocs common >

< bloc common > :: = |< nom bloc common >|< liste common >

< nom bloc common > :: = < vide >/< identificateur >

< liste common > :: = < elem common >/
< elem common >,< liste common >

< elem common > :: = < nom var simple >/< nom tableau >
< nom tableau >(< decl bornes com >)

< dec bornes com > :: = < borne com >/<borne com >,
< decl bornes com >

< borne com > :: = < entier >

restriction

Le nombre de bornes d'une déclaration d'un tableau dans un common est limité à 7

2.2. : Représentation externe d'une suite de symboles de base

2.2.1. : Définition

Soit $S = (s_1, \ldots, s_n)$ une suite de symboles de base ($n \geq 0$)
Soit α une chaîne de caractères.
α est une <u>représentation externe</u> de S
Ssi α est de la forme $\beta_0 s_1 \beta_1 s_2 \ldots \beta_{n-1} s_n \beta_n$
où $\forall i : 0 \leq i \leq n : \beta_i$ est une chaîne finie d'espaces
et $\forall i : 0 < i < n$: si s_i et s_{i+1} sont chacun soit un identificateur ou mot réservé, soit une constante entière réelle ou double précision.
alors β_i est non vide.

2.2.2. : Proposition

Toute chaîne de caractères α est la représentation externe d'au plus une suite S de symboles de base qu'on notera alors Ssymb(α).
Restriction : cette proposition n'est vraie que si on considère ** comme étant un seul caractère.

Ssymb est donc une fonction partielle de l'ensemble des chaînes de caractères dans celui des suites de symboles de base.

2.2.3. : Correction lexicale d'une chaîne de caractères

Soit α une chaîne de caractères.
Nous dirons que α est <u>lexicalement correcte</u>
Ssi il existe une suite de symboles de base S
dont α est une représentation externe (S = Ssymb(α)).

2.3. : Correction syntaxique d'une suite de symboles de base

2.3.1. : Catégorie syntaxique

La catégorie syntaxique associée à un symbole non terminal de la grammaire définissant les constructions est <u>par définition</u> l'ensemble des suites de symboles de base qui peuvent être déduites de ce symbole non terminal.

Soit < Snt > un symbole non terminal,
Nous noterons cat(< Snt >) cette catégorie syntaxique.

2.3.2. : Correction syntaxique

Soit S, une suite de symboles de base.
Soit < Snt > un symbole non terminal
Nous dirons que S est syntaxiquement correcte dans < Snt >
Ssi S $\in$ cat(< Snt >)

2.4. : Correction sémantique d'une construction

2.4.1. : Notion de type et compatibilité

L'ensemble des types est l'ensemble suivant : {INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL}. L'ensemble des types arithmétiques est {INTEGER, REAL, DOUBLE PRECISION, COMPLEX}.

L'ensemble des opérateurs monadiques est l'ensemble des symboles de base suivant : {.NOT., +, -}. L'ensemble des opérateurs diadiques est l'ensemble des symboles de base suivant :
{.EQ.,.NE.,.LE.,.GT.,.LT.,.GE.,.OR.,.AND.,+,-,*,/,**}

Définissons maintenant deux relations de compatibilité entre types et opérateurs, de même que deux opérations associant à des types et opérateurs compatibles un nouveau type appelé type résultat.

* Nous dirons que le type t est compatible avec l'opérateur monadique ω SSi :
  - si ω = .NOT. alors t est LOGICAL
    le type résultat sera LOGICAL
  - si ω = + ou - alors t est arithmétique
    le type résultat sera t

  Nous noterons tr ou f(t,ω) le type résultat qui ne sera défini que si t est compatible avec ω.
* Nous dirons que les types t1 , t2 sont compatibles avec l'opérateur diadique ω SSi :
  - si ω = .OR. ou .AND. alors t1 et t2 sont LOGICAL
    le type résultat sera LOGICAL
  - si $\omega \in$ {.EQ.,.NE.,.LE.,.GT.,.LT.,.GE.}
    alors t1 et t2 sont arithmétiques
    le type résultat sera LOGICAL

- si $\omega \in \{+,-,*,/\}$ alors t1 et t2 sont arithmétiques
  le type résultat sera celui tel que décrit dans le tableau ci-après
- si $\omega = **$ alors t1 et t2 sont arithmétiques et si t1 est COMPLEX, alors t2 n'est pas DOUBLE PRECISION.
  le type résultat sera celui tel que décrit dans le tableau ci-après.

t_2 \ t_1	INTEGER	REAL	DOUB. PREC.	COMPLEX
INTEGER	INTEGER	REAL	DOUB. PREC.	COMPLEX
REAL	REAL	REAL	DOUB. PREC.	COMPLEX
DOUB. PREC.	DOUB. PREC.	DOUB. PREC.	DOUB. PREC.	COMPLEX
COMPLEX	COMPLEX	COMPLEX	COMPLEX	COMPLEX

Tableau donnant le type résultat en fonction de t_1 et t_2.

Nous noterons tr ou f(t1,t2,ω) le type résultat qui ne sera défini que lorsque t1,t2 sont compatibles avec ω.

Nous dirons encore que 2 types sont compatibles
Ssi ils sont tous deux arithmétiques ou tous deux LOGICAL.

2.4.2. : Descripteur

On se donne un ensemble appelé ensemble des natures dont les 5 éléments seront notés sdec, tab, ext, labcom, com.
Un descripteur est constitué d'une nature : Nat(d)
et d'une description
une description est une suite d'au plus 3 éléments :

- le 1° : un identificateur
- le 2° : un type
- le 3° : un entier

Nous dirons que d est un descripteur de déclaration simple, de tableau, d'external, de label common, de common si Nat(d) vaut respectivement sdec, tab, ext, labcom, com.

La description d'un descripteur d'external, de label common et de common est composée de 1 élément.
La description d'un descripteur de déclaration simple est composée de 2 éléments.
La description d'un descripteur de tableau est composée de 3 éléments.

Soit d un descripteur, nous appellerons
Ident(d) le 1º élément de sa description
Type(d) le 2° élément de sa description, s'il existe
Nbind(d) le 3° élément de sa description, s'il existe
Nous dirons aussi que Ident(d) est le nom du descripteur d.

2.4.3. : Contexte

2.4.3.1. : Ensemble de descripteurs

Soit D un ensemble de descripteurs.

* Nous noterons Csdec(D), Ctab(D), Cext(D), Clabcom(D), Ccom(D) respectivement l'ensemble des descripteurs de déclaration simple, de tableau, d'external, de label common, de common de D .

ou encore : $Csdec(D) = \{d \in D : Nat(d) = sdec\}$

$\vdots$

$Ccom(D) = \{d \in D : Nat(d) = com\}$

nous obtenons :

$Csdec(D) \cup \ldots \cup Ccom(D) = D$

* Nous noterons Nsdec(D), Ntab(D), Next(D), Nlabcom(D), Ncom(D) respectivement l'ensemble des noms de descripteurs de déclaration simple, de tableau, d'external, de label common, de common de D . Nous noterons également Ntotal(D) la réunion de ces ensembles.

ou encore : $Nsdec(D) = \{x : \exists d \in Csdec(D) \text{ t.q } x = Ident(d)\}$

$\vdots$

$Ncom(D) = \{x : \exists d \in Ccom(D) \text{ t.q } x = Ident(d)\}$

$Ntotal(D) = Nsdec(D) \cup \ldots \cup Ncom(D)$

$= \{x : \exists d \in D \text{ t.q } x = Ident(d)\}$

2.4.3.2. : Définition

Soit C, un ensemble de descripteurs

C est un <u>contexte</u> Ssi (par définition)

1. Deux descripteurs de C de même nature ont des noms différents.
   $\forall$ d1,d2 $\in$ C : d1 $\neq$ d2 et Nat(d1) = Nat(d2) $\rightarrow$ Ident(d1) $\neq$ Ident(d2)
2. Si d est un descripteur de tableau de C, alors il n'existe aucun autre descripteur de même nom dans C sauf, éventuellement, un descripteur de common.
   $Ntab(C) \cap (Nsdec(C) \cup Next(C) \cup Nlabcom(C)) = \emptyset$
3. Si d est un descripteur de label common de C, alors il n'existe aucun autre descripteur de même nom dans C.
   $Nlabcom(C) \cap (Nsdec(C) \cup Ntab(C) \cup Next(C) \cup Ncom(C)) = \emptyset$
4. Si d est un descripteur de common de C, alors il n'existe aucun descripteur de même nom dans C sauf, éventuellement, un descripteur de déclaration simple ou de tableau.
   $Ncom(C) \cap (Next(C) \cup Nlabcom(C)) = \emptyset$
5. Si d est un descripteur d'external de C, alors il n'existe aucun descripteur de même nom dans C sauf, éventuellement, un descripteur de déclaration simple.
   $Next(C) \cap (Ntab(C) \cup Ncom(C) \cup Nlabcom(C)) = \emptyset$

<u>remarques</u>

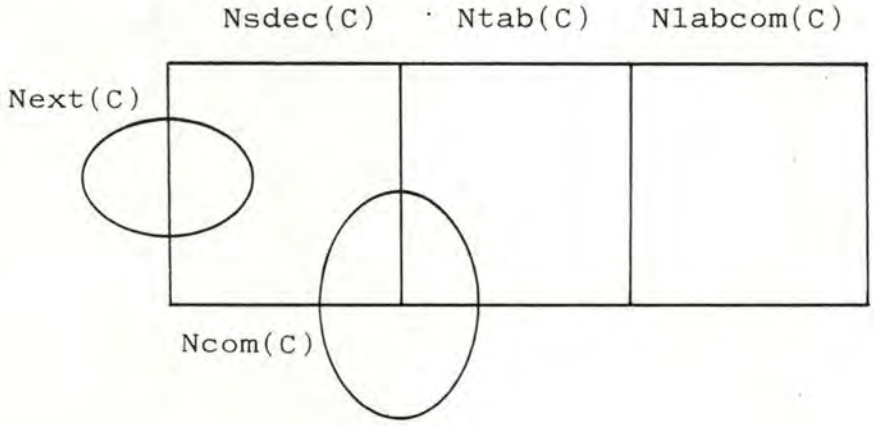
La condition 1. peut s'exprimer également comme suit:

$x \in Nsdec(C) \rightarrow \exists!\ d \in Csdec(C) : Ident(d) = x$

de même pour tab, ext, com, labcom.

Nous dirons que le <u>descripteur correspondant</u> à x dans Csdec(C), s'il existe, est unique.

Les conditions 2. à 5. contiennent certaines redondances. Nous pouvons les résumer en un schéma représentant les différents ensembles de noms.



2.4.3.3. : Opération sur les contextes

A. : *Addition d'un descripteur*

A1. admissibilité

Nous dirons que le descripteur d est admissible dans le contexte C ce que nous noterons (d ad C)

Ssi : 1. $d \notin C$

2. C U {d} est un contexte

A2. addition

Si (d ad C), nous noterons Add(C,d) le contexte C U {d}
sinon Add(C,d) n'est pas défini.
Add est donc une fonction partielle.

A3. critère d'admissibilité

Soit C, un contexte
d un descripteur avec x = Ident(d).

1° cas : Nat (d) = sdec

Pour que C U {d} soit un contexte, il faut et il suffit qu'il respecte les conditions 1. à 5. des contextes (2.4.3.2.) sachant que C les respecte.

1. Nat(d) = sdec;
   la condition devient : $\forall$ d1 $\in$ Csdec(C) : d1 $\neq$ d $\rightarrow$ Ident(d1) $\neq$ Ident(d)
   Si nous incluons ici qu' il faut $d \notin C$
   nous obtenons : $\forall$ d1 $\in$ Csdec(C) : Ident(d1) $\neq$ Ident(d)
   ou encore plus simplement; $x \notin$ Nsdec(C)

2. Ntab(C U {d}) = Ntab(C)
   Nsdec(C U {d}) = Nsdec(C) U {x}
   Next(C U {d}) = Next(C)
   Nlabcom(C U {d}) = Nlabcom(C)
   Ncom(C U {d}) = Ncom(C)
   La condition devient alors
   Ntab(C) $\cap$ (Ndec(C) U {x} U Next(C) U Nlabcom(C)) = $\emptyset$
   sachant que
   Ntab(C) $\cap$ (Ndec(C) U Next(C) U Nlabcom(C)) = $\emptyset$
   Nous obtenons alors comme condition
   Ntab(C) $\cap$ {x} $\neq \emptyset$
   ou encore $x \notin$ Ntab(C)

Avec un même raisonnement, nous obtenons encore :

3. $x \notin$ Nlabcom(b)

4. pas de condition

5. pas de condition

Donc (d ad C) Ssi $x \notin$ Nsdec(C) U Ntab(C) U Nlabcom(C).

2° cas : Nat(d) = tab

Un raisonnement de même type nous donnerait
(d ad C) Ssi $x \notin$ Nsdec(C) U Ntab(C) U Next(C) U Nlabcom(C).

3° cas : Nat(d) = ext

Un raisonnement de même type nous donnerait
(d ad C) Ssi $x \notin$ Ntab(C) U Next(C) U Nlabcom(C) U Ncom(C).

4° cas : Nat(d) = labcom

Un raisonnement de même type nous donnerait
(d ad C) Ssi $x \notin$ Ntotal(C).

5° cas : Nat(d) = com

Un raisonnement de même type nous donnerait
(d ad C) Ssi $x \notin$ Next(C) U Nlabcom(C) U Ncom(C).

B. : *Soustraction d'un descripteur*

Soit C, un contexte
d, un descripteur

Nous définissons l'opération Sub(C,d) = C - {d}
Sub est une fonction totale
propriété : Sub(C,d) est aussi un contexte.
en effet : Si $d \notin C$: Alors Sub(C,d) = C qui est un contexte.
Si $d \in C$: il faut vérifier que C - {d} respecte les conditions 1. à 5. des contextes.

1. évident car vrai aussi $\forall$ d1,d2 $\in$ C - {d}
2. à 5. Les contraintes d'intersection vide seront à fortiori respectées car les ensembles Nsdec(C - {d}), Ntab(C - {d}), Next(C - {d}), Nlabcom(C - {d}), Ncom(C - {d}) sont inclus respectivement à Nsdec(C), Ntab(C), Next(C), Nlabcom(C), Ncom(C) qui respectent ces propriétés.

C. : *Addition d'une liste de descripteurs*

C1. : admissibilité

Soit C, un contexte

l, une liste de descripteurs $(d_1, \ldots, d_n)$ $n \geq 0$

Nous dirons que la liste de descripteurs l est admissible dans le contexte C, ce que nous noterons (l L-ad C)

Ssi : 1. $C \cap \{d_1, d_2, \ldots, d_n\} = \emptyset$

2. $C \cup \{d_1, d_2, \ldots, d_n\}$ est un contexte

3. les d_i sont distincts deux à deux.

C2. : addition

Si (l L-ad C), nous noterons L-Add(C,l) le contexte $C \cup \{d_1, \ldots, d_n\}$ sinon L-Add(C,l) n'est pas défini.

Nous en déduisons les propriétés suivantes :

. (() L-ad C)

. Si (d_1 ad C)

Alors ($(d_1, \ldots, d_n)$ L-ad C) <-> ($(d_2, \ldots, d_n)$ L-ad Add(C,d_1))

. (non (d_1 ad C)) -> (non ($(d_1, \ldots, d_n)$ L-ad C))

Ces 3 propriétés nous permettent alors de donner une définition de (l L-ad C) plus algorithmique :

($(d_1, \ldots, d_n)$ L-ad C) <-> (Si n = 0 alors vrai
sinon (d_1 ad C) et
($(d_2, \ldots, d_n)$ L-ad Add(C,d_1))

avec la convention que (c1 et c2) vaut faux chaque fois que c1 est faux (c'est à dire que c2 peut alors être indéterminé).

D. : *Soustraction d'une liste de descripteurs*

Soit C, un contexte

l, une liste de descripteurs $(d_1, \ldots, d_n)$ $n \geq 0$

Nous définissions l'opération L-Sub(C,l) = $C - \{d_1, \ldots d_n\}$

propriété : L-Sub(C,l) est aussi un contexte.

en effet : si n = 0, L-Sub(C,l) = C qui est un contexte.

si n > 0 L-Sub(C,l) = $C - \{d_1, \ldots, d_n\}$
= Sub(C,d_1) - $\{d_2, \ldots, d_n\}$

avec Sub(C,d_1) qui est un contexte
et $(d_2, \ldots, d_n)$ suffixe propre de l.

Par induction sur n, nous obtenons donc que L-Sub(C,l) est un contexte.

E. : *Propriétés des opérations*

E1. : commutativité

Soit C, un contexte, d1 et d2, deux descripteurs
Si tous les termes ont un sens,

- Add (Add(C,d1),d2) = Add (Add(C,d2),d1)
  en effet : Add (C U {d1}, d2) = Add (C U {d2}, d1)
  car C U {d1} U {d2} = C U {d2} U {d1}

- Sub (Sub(C,d1),d2) = Sub (Sub(C,d2), d1)
  en effet : Sub (C - {d1}, d2) = Sub (C - {d2}, d1)
  car C - {d1} - {d2} = C - {d2} - {d1}

Soit aussi l, une liste de descripteurs $(d_1,\ldots,d_n)$ $n \geq 0$
Soit l', une liste des n descripteurs de l mais dans un ordre différent $(d'_1,\ldots,d'_n)$

- L-Add(C,l) = L-Add(C,l') car
  C U $\{d_1,\ldots,d_n\}$ = C U $\{d'_1,\ldots,d'_n\}$

- L-Sub(C,l) = L-Sub(C,l') car
  C - $\{d_1,\ldots,d_n\}$ = C - $\{d'_1,\ldots,d'_n\}$

E2. : opération inverse

Soit C, un contexte
d, un descripteur
Si tous les termes ont un sens

- Sub(Add(C,d),d) = C
  en effet : Sub(C U {d},d) = C
  car (C U {d}) - {d} = C

puisque Add(C,d) n'aura de sens que si, entre autre, $d \notin C$.

- Soit l, une liste de descripteurs $(d_1,\ldots,d_n)$ $n \geq 0$
  L-Sub(L-Add(C,l)) = C
  en effet : L-Sub(C U $\{d_1,\ldots,d_n\}$) = C
  car (C U $\{d_1,\ldots,d_n\}$) - $\{d_1,\ldots,d_n\}$ = C

puisque L-Add(C,l) n'aura de sens que si, entre autre,
$\{d_1,\ldots,d_n\} \cap C = \emptyset$.

2.4.4. : Correction sémantique

2.4.4.1. : Déclaration

A. : *liste de descripteurs associée à une déclaration*

Soit S, une déclaration : S $\in$ cat(<déclaration>).
On lui associe une liste de descripteurs selon les règles suivantes :

A1. : S $\in$ cat(<decl variable>)

S sera de la forme
Type long $v_1, v_2, \ldots, v_n$ avec Type $\in$ cat(<type>)
long $\in$ cat(<longueur>)
$v_i \in$ cat(<variable dec.>)
$n > 0$

A chaque v_i correspondra 1 descripteur d_i de la liste.
si $v_i \in$ cat(<identificateur>)
Alors Nat(d_i) = sdec
description de d_i : (v_i,Type)
si v_i est de la forme $t_i(b_1, \ldots, b_k)$
avec $t_i \in$ cat(<identificateur>)
$b_i \in$ cat(<borne>)
$k > 0$
Alors Nat(d_i) = tab
description de d_i : (t_i,Type,k)

A2. : S $\in$ cat(<decl external>)

S sera de la forme
EXTERNAL $id_1, id_2, \ldots, id_n$ avec $id_i \in$ cat(<identificateur>)
$n > 0$
A chaque id_i correspondra 1 descripteur d_i de la liste
avec Nat(d_i) = ext
description de d_i : (id_i)

A3. : S $\in$ cat(< decl common >)

S sera d'une des formes
.1. : COMMON $c_1, c_2, \ldots c_n$
.2. : COMMON $blc_1, blc_2, \ldots, blc_m$
.3. : COMMON $c_1, \ldots, c_n blc_1 \ldots blc_m$
avec $c_i \in$ cat(<elem common>)
$blc_j \in$ cat(<bloc common>)
$n, m > 0$

- Pour la forme .1., à chaque c_i correspondra 1 descripteur d_i de la liste associée à S.

  si $c_i \in$ cat(<identificateur>)
  Alors Nat(d_i) = com
  description de d_i : (c_i)
  si c_i est de la forme $t_i(b_1,\ldots,b_k)$
  avec $t_i \in$ cat(<identificateur>)
  $b_i \in$ cat(<borne com>)
  $k > 0$
  Alors Nat(d_i) = com
  description de d_i : (t_i).

- Pour la forme .2., à chaque blc_j correspondra 1 liste de descripteurs l_j. La liste associée à S sera la concaténation des listes $l_1,\ldots,l_m$.
  Chaque blc_j sera de la forme /lbnom/$c_1,\ldots,c_{n_j}$
  avec lbnom $\in$ cat(<nom bloc common>)
  c_i cat(<elem common>)
  $n_j > 0$
  si lbnom cat(<identificateur>)
  Alors l_j sera $(d_0,d_1,\ldots,d_{n_j})$
  avec Nat(d_0) = Labcom
  description de d_0 : (lbnom)
  Les descripteurs $d_1,\ldots,d_{n_j}$ seront comme ceux décrits à la forme .1.
  si lbnom $\in$ cat(<vide>)
  Alors l_j sera $(d_1,\ldots,d_{n_j})$
  avec les descripteurs comme ceux décrits à la forme .1.

- Pour la forme .3., la liste associée à S sera la concaténation des deux listes l_1 et l_2 avec
  l_1 liste associée à COMMON $c_1,\ldots,c_n$
  et l_2 liste associée à COMMON $blc_1,\ldots,blc_m$.

A4. : exemples

. REAL*4 A,TEST(8,N),TOTO
la liste associée sera (<sdec,<A, REAL> >,
<tab, < TEST,REAL,2> >,
<sdec,<TOTO,REAL> >).

. EXTERNAL X,TOTO,SEARCH
la liste associée sera (<ext,<X> >,
<ext,<TOTO> >,
<ext,<SEARCH> >) .

. COMMON A,TABLEAU(2,8),TOTO
la liste associée sera (<com, <A> >,
<com, <TABLEAU> >,
<com, <TOTO> >) .

. COMMON /LAB1/A,B//C/LAB2/F
la liste associée sera (<labcom, <LAB1> >,
<com , <A> >,
<com , <B> >,
<com , <C> >,
<labcom, <LAB2> >,
<com , <F> >) .

. COMMON A,TOTO/LBL/C
la liste associée sera (<com , <A> >,
<com , <TOTO> >,
<labcom, <LBL> >,
<com , <C> >) .

B. : *Correction sémantique d'une déclaration*

Soit C,un contexte
S,une déclaration

B1. : Si S $\in$ cat(<decl variable>)

Nous dirons que S est une déclaration partiellement sémantiquement correcte dans C
Ssi toutes les déclarations de bornes de tableaux qui sont des identificateurs (soit x un de ces identificateurs) sont telles que : . $x \in$ Nsdec(C)
. Type (d) = INTEGER avec d, le descripteur correspondant à x dans Csdec(C)

B2. : Correction sémantique

Nous dirons que S est une déclaration sémantiquement correcte dans C

Ssi . si S ∈ cat(decl variable), S est une déclaration partiellement sémantiquement correcte dans C.
. (l L-ad C) avec l, liste de descripteurs associée à S .

2.4.4.2. : Expression

Soit C, un contexte
S, une expression
S se décompose sous une et une seule des formes décrites ci-après. Nous donnerons, pour chacune de ces formes, tout d'abord les conditions nécessaires et suffisantes pour que S soit sémantiquement correct dans C, ensuite le type (tr) de S dans C qui ne sera défini que lorsque S est sémantiquement correcte dans C.

A) expr .OR. conj

avec expr ∈ cat(<expression>)
conj ∈ cat(<conjonction>)
- expr et conj sont des expressions sémantiquement correctes dans C
C de type t1,t2 compatibles avec l'opérateur diadique .OR.
- tr = f(t1,t2,.OR.)

B) conj .AND. neg

avec conj ∈ cat(<conjonction>)
neg ∈ cat(<négation>)
- conj et neg sont des expressions sémantiquement correctes dans C de type t1,t2 compatibles avec l'opérateur diadique .AND.
- tr = f(t1,t2,.AND.)

C) .NOT. patom

avec patom ∈ cat(<proposition atomique>)
- patom est une expression sémantiquement correcte dans C de type t1 compatible avec l'opérateur monadique .NOT.
- tr = f(t1,.NOT.)

D) aexpr1 ω aexpr2

avec aexpr1,aexpr2 ∈ cat(<expression arithmétique>)
ω ∈ cat(<op rel>)
- aexpr1 et aexpr2 sont des expressions sémantiquement correctes dans C, de type t1,t2 compatibles avec l'opérateur diadique ω .
- tr = f(t1,t2, ω)

E) ω term

avec term ∈ cat(<terme>)
ω ∈ cat(<op add>)

- term est une expression sémantiquement correcte dans C, de type t1 compatique avec l'opérateur monadique ω.
- tr = f(t1, ω)

F) aexpr ω term

avec aexpr ∈ cat(<expression arithmétique>)
term ∈ cat(<terme>)
ω ∈ cat(<op add>)

- aexpr et term sont des expressions sémantiquement correctes dans C de type t1,t2 compatibles avec l'opérateur diadique ω
- tr = f(t1,t2, ω)

G) term ω fac

avec term ∈ cat(<terme>)
fac ∈ cat(<facteur>)
ω ∈ cat(<op mult>)

- term et fac sont des expressions sémantiquement correctes dans C, de type t1,t2, compatibles avec l'opérateur diadique ω
- tr = f(t1,t2, ω)

H) sexpr ** fac

avec sexpr ∈ cat(<expression simple>)
fac ∈ cat(<facteur>)

- sexpr et fac sont des expressions sémantiquement correctes dans C, de type t1,t2 compatibles avec l'opérateur diadique ω
- tr = f(t1,t2, ω)

I) cste

avec cste ∈ cat(<constante>)

- néant
- tr sera INTEGER,REAL,DOUBLE PRECISION,COMPLEX ou LOGICAL selon que cste ∈ cat(<entier>),cat(<réel>),cat(<double précision>), cat(<complexe>) ou cat(<logique>)

J) (expr)

avec expr ∈ cat(<expression>)

- expr est une expression sémantiquement correcte dans C
- tr est le type de expr dans C

K) (cplxe)

avec cplxe ∈ cat(<complexe>)

- néant
- tr est COMPLEX

L) x

avec x ∈ cat(<identificateur>)

- x ∈ Nsdec(C)
- tr = type(d), avec d, le descripteur correspondant à x dans Csdec(C)

M) $x(expr_1, \ldots, expr_n)$

avec x ∈ cat(<identificateur>)

$expr_i$ ∈ cat(<expression>)

$n < 0$

Si x ∈ Nsdec(C)

- si $expr_i$ ∈ cat(<identificateur>)

  alors $expr_i$ ∈ Nsdec(C) U Ntab(C) U Next(C)

  sinon $expr_i$ est une expression sémantiquement correcte dans C
- tr = type(d), avec d, le descripteur correspondant à x dans Csdec(C).

Sinon

- x ∈ Ntab(C) et $expr_i$ est une expression sémantiquement correcte dans C de type INTEGER ou REAL, ∀ i : $1 \leqslant i \leqslant n$
- tr = type(d), avec d, le descripteur correspondant à x dans Ctab(C)

2.4.4.3. : Affectation

Soit C, un contexte

S, une affectation

S aura l'une des deux formes suivantes :

A) x = expr

avec x ∈ cat(<identificateur>)

expr ∈ cat(<expression>)

S sera une affectation sémantiquement correcte dans C

Ssi - x ∈ Nsdec(C). Soit d le descripteur correspondant à x dans Csdec(C)

- expr est une expression sémantiquement correcte dans C de type tr compatible avec type(d)

B) $\underline{x\ (expr_1, \ldots, expr_n) = expr}$

avec $x \in$ cat(<identificateur>)

$expr_i, expr \in$ cat(<expression>)

S sera une affectation sémantiquement correcte dans C

Ssi - $x \in$ Ntab(C). Soit d le descripteur correspondant à x dans Ctab(C)

- chaque $expr_i$ est une expression sémantiquement correcte dans C, de type INTEGER ou REAL
- n = nbind(d)
- expr est une expression sémantiquement correcte dans C et de type tr compatible avec type(d).

2.4.4.4. Condition

Soit C, un contexte

S, une affectation

S sera une condition sémantiquement correcte dans C

Ssi S est une expression sémantiquement correcte dans C de type LOGICAL.

3. : SPECIFICATION TECHNIQUE

3.1. : Environnement

L'environnement est un ensemble de variables et de fichiers accessibles. Cet environnement est constitué de plusieurs parties disjointes.

3.1.1. : chaîne de caractères courante

Une partie de l'environnement sert à représenter une chaîne de caractères. On fixera certaines règles de représentation de telle manière que l'on puisse dire qu'à un instant déterminé, la chaîne de caractères courante est α ; ce que l'on notera chc = α .

3.1.2. : contexte courant

Une partie de l'environnement sert à représenter un contexte. On fixera certaines règles de représentation de telle manière que l'on puisse dire qu'à un instant déterminé, le contexte courant est C; ce que l'on notera cntc = C.

3.1.3. : liste de descripteurs courante

Une partie de l'environnement sert à représenter une liste de descripteurs. On fixera certaines règles de représentation de telle manière que l'on puisse dire qu'à un instant déterminé, la liste de descripteurs courante est l; ce que l'on notera ldsc = l.

3.1.4. : partie constante

Une partie de l'environnement se compose d'un ensemble de variables ayant une valeur constante entière.

3.1.5. : message d'erreur

Une partie de l'environnement sert à représenter un message d'erreur dont les règles de représentation seront fixées ultérieurement.

3.2. : Analyseur Fortran

3.2.1. : fonction opération

Nous allons définir une fonction que nous appellerons fonction opération qui nous permettera de définir simplement la spécification technique de l'analyseur Fortran, de même que les primitives utiles pour ce dernier.

Soit $\mathcal{C}$, l'ensemble de tous les contextes

A, l'ensemble des suites finies de caractères

G = {Déclaration,Affectation,Condition}

O = {Destruction,Traitement}

OP : $\mathcal{C}$ x A x G x O $\longrightarrow$ $\mathcal{C}$ x {.TRUE.,.FALSE.} est définie comme suit :

Soit C, un contexte

α, une suite finie de caractères

- OP(C,α,Déclaration,Traitement)
  = (L-Add(C,l),.FALSE.)
  si α est une représentation externe d'une déclaration S sémantiquement correcte dans C.
  Avec l, liste de descripteurs associée à S.
  = (C,.TRUE.) sinon
- OP(C,α,Affectation,Traitement)
  = (C,.FALSE.)
  si α est une représentation externe d'une affectation S sémantiquement correcte dans C.
  = (C,.TRUE.) sinon
- OP(C,α,Condition,Traitement)
  = (C,.FALSE.)
  si α est une représentation externe d'une condition S sémantiquement correcte dans C.
  = (C,.TRUE.) sinon
- OP(C,α,Déclaration,Destruction)
  =(L-Sub(C,l),.FALSE.)
  si α est 1 représentation externe d'une déclaration S
  = (C,.TRUE.) sinon
- OP(C,α,Affectation,Destruction) = (C,.FALSE.)
- OP(C,α,Condition,Destruction) =(C,.FALSE.)

3.2.2. : Spécification de l'analyseur Fortran

Sous-routine ANAFOR

paramètres:

ANGENR : type INTEGER*2
signification : représente un genre de construction $g \in G$,
valeurs possibles (appartenant à la partie constante de l'environnement) :
GENDEC si g = Déclaration
GENAFF si g = Affectation
GENCON si g = Condition

ANOPER : type INTEGER*2
signification : représente une opération op $\in$ O
valeurs possibles (appartenant à la partie constante de l'environnement):
OPTRT si op = Traitement
OPDEST si op = Destruction

ANERR : type LOGICAL

effet :

Si chc = α et cntc = C
Alors si ANGENR ou ANOPER n'a pas une des valeurs possibles décrites ci-dessus
alors l'exécution de ANAFOR établit ANERR = .FALSE.
sinon l'exécution de ANAFOR établit
(cntc,ANERR) = OP (C,α,g,op)
et le message d'erreur appartenant à l'environnement sera rempli si ANERR = .TRUE.

3.2.3. : Analyse d'une déclaration, d'une affectation, d'une condition

- sous-routine TRTDEC : traitement déclaration
  paramètre : TRTERR : type LOGICAL
- sous-routine TRTAFF : traitement affectation
  paramètre : TRTERR : type LOGICAL
- sous-routine TRTCON : traitement condition
  paramètre : TRTERR :type LOGICAL

effet

Si chc = α et cntc = C

Alors l'exécution d'une de ces sous-routines établit

(cntc,TRTERR) = OP (C,α,g, traitement)

avec g = Déclaration pour TRTDEC

g = Affectation pour TRTAFF

g = Condition pour TRTCON

et le message d'erreur appartenant à l'environnement sera rempli si TRTERR = .TRUE.

3.2.4. : Destruction d'une déclaration

sous-routine DESDEC

paramètre :

DESERR : type LOGICAL

effet

Si chc = α et cntc = C

Alors l'exécution de DESDEC établit

(cntc,DESERR) = OP (C,α,Déclaration,Destruction)

et le message d'erreur appartenant à l'environnement sera rempli si DESERR = .TRUE.

3.3. : Analyse lexicale

Nous allons spécifier ici une primitive qui nous permettra de considérer la chaîne de caractères courante, selon certaines conditions, comme une suite de symboles de base que l'on pourra traiter.

3.3.1. : Représentation interne d'un symbole de base

Soit s, un symbole de base

Un triplet (i,v_1,v_2) avec $i \in$ {identi,const,spesym}

v_1 = chaîne de 8 caractères

v_2 = entier

sera une représentation interne de s

SSi

1) $s \in$ cat(<identificateur>)

i = identi

v_1 = chaîne de 8 caractères de la forme S^^^

2) s ∈ cat(<constante>)

i = const

v_2 = cstint si s ∈ cat(<entier>)
cstrea si s ∈ cat(<réel>)
cstlog si s ∈ cat(<logique>)
cstdpr si s ∈ cat(<double précision>)

3) s ∈ cat(<symbole spécial>)

i = spesym

v_2 = élément correspondant à s dans la table suivante

S	V_2	S	V_2	S	V_2
+	cdplus	.EQ.	cdleq	INTEGER	cdinte
-	cmoin	.NE.	cdlne	REAL	cdreal
*	cdfois	.GT.	cdlgt	COMMON	cdcomm
/	cddivi	.LT.	cdllt	DOUBLE	cddoub
ou également					
	cdrsla	.LE.	cdlle	PRECISION	cdprec
**	cdexpo	.GE.	cdlge	LOGICAL	cdlogi
=	cdegal	.NOT.	cdlnot	COMPLEX	cdcomp
(	cdpouv	.OR.	cdlor	CALL	cdcall
)	cdfer	.AND.	cdland	EXTERNAL	cdexte
,	cdvirg				
'	cdapos				

A chaque valeur distincte de i correspond une constante de même nom, appartenant à la partie constante de l'environnement; les valeurs de ces constantes sont distinctes. De même pour les valeurs de v_2.

Nous dirons que s est le symbole de base courant

SSi les variables IBSYMB de type INTEGER*2
VBSYMB de type REAL*8
CBSYMB de type INTEGER*2

appartenant à l'environnement sont égales à une représentation interne de s.

3.3.2. : Spécification de l'analyseur lexical

sous-routine LIRSYM

effet

Soit chc = α, plus grand préfixe de α

tel que - α' représente une suite de symboles de base $S = s_1, \ldots, s_n$

- Si s_n existe et se termine par un chiffre ou une lettre, s_n n'est pas suivi par une lettre ou un chiffre dans α.

Donc $\alpha = \alpha'\gamma$ où γ est une certaine chaîne de caractères.

α' se décompose d'une seule façon sous la forme

$\varepsilon\beta$ avec ε :suite (éventuellement vide) de blancs

β :suite (éventuellement vide) de caractères ne commençant pas par un blanc.

Si $\beta \neq \lambda$ (suite vide) Alors $\beta = s_1\beta'$

L'exécution de LIRSYM établit que s_1 est le symbole de base courant et que chc = $\beta'\gamma$.

Si $\beta = \lambda$ et $\gamma = \lambda$ Alors

L'exécution de LIRSYM affecte à IBSYMB la constante NOSYMB appartenant à la partie constante de l'environnement.

Si $\beta = \lambda$ et $\gamma \neq \lambda$ Alors

L'exécution de LIRSYM affecte à IBSYMB la constante ERRSYM appartenant à la partie constante de l'environnement.

On pourrait démontrer que le premier symbole s_1 de S est le plus grand préfixe de β qui soit un symbole de base,

sauf si $\beta = s_1 s_2 \beta''$

où s_1 est une constante entière

s_2 est une constante logique ou un symbole spécial de plusieurs caractères commençant par un point.

3.4. : Analyse d'une suite de symboles de base

Nous allons développer ici un ensemble de primitives effectuant des vérifications syntaxiques et sémantiques sur une suite de symboles de base.

3.4.1. : Suite de symboles de base à traiter

Soit chc = α, α' plus grand préfixe de α

tel que-α' représente une suite de symboles de base $S = s_1, \ldots, s_n$

- Si s_n existe et se termine par un chiffre ou une lettre, s_n n'est pas suivi par une lettre ou une chaîne dans α

  Donc $\alpha = \alpha'\gamma$ où γ est une certaine chaîne de caractères

- Si s_1 est le symbole de base courant

  Alors, par définition, la suite $s_1, s_2, \ldots, s_n$ est la suite des symboles de base à traiter (SSt en abrégé)

- Si IBSYMB = ERRSYM ou NOSYMB

  Alors, par définition, la suite vide est la suite des symboles de base à traiter.

3.4.2. : Traitement d'une suite de symboles

Soit SSt = $s_1, s_2, \ldots, s_n$ avec $n>0$

S = $s_1, \ldots, s_j$ préfixe non vide de SSt avec $0<j\leq n$

Nous dirons qu'une exécution d'une partie de l'analyseur Fortran aura pour effet, parmi d'autres, de traiter la suite S

Ssi 1. SSt = $s_{j+1}, \ldots, s_n$ après cette exécution

2. si $j<n$: chc = $\alpha'\beta$ avec α', représentation externe de S

   si $j=n$ et chc = λ, IBSYMB = NOSYMB

   si $j=n$ et chc $\neq \lambda$, IBSYMB = ERRSYM

remarque

Comme corollaire pour l'utilisation de LIRSYM, remarquons que j appels successifs à LIRSYM auront pour effet de traiter S.

3.4.3. : Préfixe cat-maximum

Soit S, une suite de symboles de base

<Snt>, une catégorie syntaxique

3.4.3.1. : continuateur

Soit une suite de symboles de base P $\in$ cat(<Snt>).

c est un continuateur de P dans <Snt>

Ssi il existe P' $\in$ cat(<Snt>) tel que Pc est un préfixe de P'.

Proposition

Il existe au plus 1 préfixe de S, soit P $\in$ cat(<Snt>) qui n'est pas suivi, dans S, par un continuateur de P dans <Snt>.

démonstration

Les préfixes de S ∈ cat(<Snt>) sont en nombre fini.
Soit $P_1,\ldots,P_n$ ces préfixes, rangés par ordre de longueur croissante.
Pour $1 \leqslant i < n-1$, P_i est suivi, dans S, par un continuateur de P_i dans <Snt> car P_i est un préfixe de P_{i+1}.
Si P_n n'est pas suivi, dans S, par un continuateur de P_n dans <Snt>, P_n sera le préfixe P unique
Sinon P n'existera pas.

3.4.3.2.. : Définition
.....

La propriété précédente justifie la définition suivante.
P est cat-max(<Snt>) dans S
Ssi - P est un préfixe de S
- P ∈ cat(<Snt>)
- P n'est pas suivi, dans S, par un continuateur de P dans <Snt>.

3.4.4. Analyse des déclarations de variables
---

sous-routine ANDECV
paramètres :
SYNERR : type LOGICAL
PSEMER : type LOGICAL
effet
Si SSt = S et cntc = C
Alors, si S ∈ cat(<decl variable>)
alors l'exécution de ANDECV établira
-SYNERR = .FALSE.
-ldsc = liste de descripteurs associée à S
-PSEMER = .FALSE. si S est une déclaration de variables partiellement sémantiquement correcte dans C
PSEMER = .TRUE. sinon
sinon l'exécution de ANDECV
établira SYNERR = .TRUE.
et le message d'erreur appartenant à l'environnement sera rempli.

3.4.5. : Analyse des déclarations d'externals

Sous-programme ANDECE

paramètre :

SYNERR : type LOGICAL

effet

Si SSt = S et cntc = C

Alors si s ∈ cat(<decl external>)

alors l'exécution de ANDECE établira

- SYNERR = .FALSE.
- ldsc = liste de descripteurs associée à S.

sinon l'exécution de ANDECE établira SYNERR = .TRUE. et le message d'erreur appartenant à l'environnement sera rempli.

3.4.6. : Analyse des déclarations de commons

Sous-programme ANDECC

paramètre :

SYNERR : type LOGICAL

effet

Si SSt = S et cntc = C

Alors si s ∈ cat(<decl common>)

alors l'exécution de ANDECC établira

- SYNERR = .FALSE.
- ldsc = liste de descripteurs associée à S.

sinon l'exécution de ANDECC établira SYNERR = .TRUE. et un message d'erreur appartenant à l'environnement sera rempli.

3.4.7. : Analyse des expressions

Sous-programme ANEXPR

paramètres :

ERREXP : type LOGICAL

EXPTYP : type INTEGER*2.

effet

Si SSt = S et cntc = C

Alors s'il existe un préfixe P cat-max(<expression>) dans S,
et si P est une expression sémantiquement correcte dans C

alors l'exécution de ANEXPR
- traitera ce préfixe
- établira ERREXP = .FALSE.
EXPTYP = type de l'expression P dans C

sinon l'exécution de ANEXPR
- établira ERREXP = .TRUE.
- remplira un message d'erreur appartenant à l'environnement.

3.5. : Manipulation du contexte courant

Nous allons spécifier ici des primitives qui manipulent le contexte courant comme par exemple l'implémentation des opérations définies précédemment sur les contextes

3.5.1. : Représentation interne d'un descripteur

3.5.1.1. : définitions

- 2 descripteurs sont compatibles Ssi la description de l'un est un préfixe de celle de l'autre.
- Un quadruplet (dnat, dide, dtyp, dnb)
avec dnat, dtyp, dnb , entiers
et dide , chaîne de 8 caractères.

sera une représentation interne d'un descripteur d
Ssi -dnat représente un ensemble de natures dont fait partie Nat(d)
-dide représente Nom(d)
-dtyp représente Type(d) si celui-ci est défini
-dnb représente Nbind(d) si celui-ci est défini.

Nous ne donnerons pas ici les règles de représentation d'un ensemble de natures et d'un type par un entier.

3.5.1.2. : proposition

Soit x, un identificateur
C, un contexte

Si x ∈ Ntotal(C)

Alors il existe au plus deux descripteurs de nom x et, dans ce cas, ils sont compatibles.

démonstration

Il y a 8 possibilités

1. x ∈ Nsdec(C) exclusivement
2. x ∈ Ntab(C) exclusivement
3. x ∈ Nlabcom(C) exclusivement
4. x ∈ Next(C) exclusivement
5. x ∈ Ncom(C) exclusivement
6. x ∈ Nsdec(C) Next(C) exclusivement
7. x ∈ Nsdec(C) Ncom(C) exclusivement
8. x ∈ Ntab(C) Ncom(C) exclusivement

Dans les cas 1. à 5., il existera un et un seul descripteur de C, soit d, tel que ident(d) = x

Dans les cas 6. à 8., il existera 2 descripteurs de C de même nom. Mais la description de l'un sera incluse dans celle de l'autre et ils seront donc compatibles.

en effet:

cas 6. Soit d_1 ∈ Csdec(C) et d_2 ∈ Cext(C) ces descripteurs
nous avons <Ident(d_1), Type(d_1)>, la description de d_1
et <Ident(d_2)>, la description de d_2
Or, Ident(d_1) = Ident(d_2).
Donc, la description de d_2 est incluse dans celle de d_1.

cas 7. Soit d_1 ∈ Csdec(C) et d_2 ∈ Ccom(C) ces descripteurs
nous avons <Ident(d_1), Type(d_1)>, la description de d_1
<Ident(d_2)>, la description de d_2.
Or, Ident(d_1) = Ident(d_2).
Donc, la description de d_2 est incluse dans celle de d_1.

cas 8. Soit d_1 ∈ Ctab(C) et d_2 ∈ Ccom(C) ces descripteurs
nous avons <Ident(d_1), Type(d_1), Nbind(d_1)>, la description de d_1 <Ident(d_2)>, la description de d_2.
Or, Ident(d_1) = Ident(d_2).
Donc, la description de d_2 est incluse dans celle de d_1.

C.Q.F.D.

3.5.2. : Accès à un descripteur

fonction FINDES
paramètre :
NOMDES : type REAL*8
effet
Si cntc = C
Alors si le contexte C contient un ou deux descripteurs de nom
NOMDES
alors l'exécution de FINDES établira
- FINDES = .TRUE.
- (DESNAT, DESIDE, DESTYP, DESDIM) contiendra une représentation interne de ou de ces descripteurs
sinon l'exécution de FINDES établira
FINDES = .FALSE.
DESNAT, DESTYP, DESDIM sont de type INTEGER*2 .
DESIDE est de type REAL*8 .
Ces variables font partie de l'environnement.

3.5.3. : Détermination de la nature d'un descripteur

fonction NATURD
paramètres :
DNAT : type : INTEGER*2
signification : représentation d'un ensemble de natures
NAT : type: INTEGER*2
signification : représentation d'une nature

valeurs possibles (appartenant à la partie constante de l'environnement) :
NATSDE pour sdec
NATTAB pour tab
NATEXT pour ext
NATLBC pour labcom
NATCOM pour com

effet:
Si NAT $\in$ DNAT
Alors l'exécution de NATURD établit NATURD = .TRUE.
Sinon l'exécution de NATURD établit NATURD = .FALSE.

3.5.4. : Addition d'une liste de descripteurs

sous-routine LADD
paramètre :
LADERR : type LOGICAL
effet
Si cntc = C et ldsc = l
Alors si (l L-ad C)
alors l'exécution de LADD établit
-cntc = L-Add(C,l)
-LADERR = .FALSE.
sinon l'exécution de LADD établit
-cntc = C
-LADERR = .TRUE.
et un message d'erreur appartenant à l'environnement sera rempli.

3.5.5. : Soustraction d'une liste de descripteurs

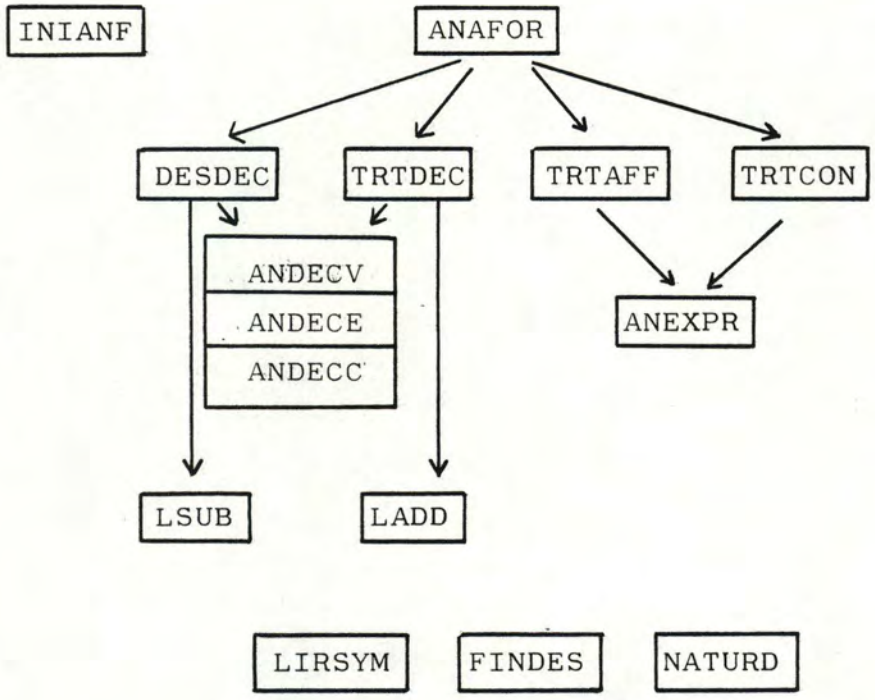
Sous-routine LSUB
paramètre : /
effet
Si cntc = C et ldsc = l
Alors l'exécution de LSUB établit cntc = L-Sub(C,l)
remarque
La propriété d'opération inverse sur les contextes nous permet d'écrire la propriété du programme suivant :

{cntc = C et ldsc = l}
LADD (LADERR)
IF NOT LADERR THEN LSUB
{cntc = C}

3.6. : Initialisation de l'analyseur

spécification sous-routine INIANF
paramètre : /
effet
l'exécution de INIANF établit cntc = {}

3.7. : Architecture



Si un arc relie A ———> B, cela signifie que B est une primitive de A

remarques :- ceci n'est évidemment qu'une ébauche d'architecture; bien d'autres primitives devront s'y ajouter.

-l'exécution d'une des sous-routine spécifiées ci-dessus n'a pas d'autres effets sur l'environnement que ceux qui sont explicitement mentionnés par sa spécification.

4. : LIEN ENTRE SPECIFICATION NATURELLE ET TECHNIQUE

4.1. : Introduction

Nous avons introduit une série de concepts non explicités tels quels dans la norme Fortran. Ces concepts nous ont permis d'obtenir une nouvelle spécification de l'analyseur Fortran dite spécification technique. Ces concepts sont tels qu'ils facilitent les raisonnements de construction des programmes et sont assez proches du Fortran pour être justifiables.

Avant de construire les programmes, nous devrons donc montrer que la spécification technique est équivalente à la spécification initiale du problème posé. Ceci est essentiel; sans quoi nous ne pourrions être raisonnablement convaincus que le programme respecte ses spécifications initiales.

Pour ce faire, nous allons montrer que si un programme respecte la spécification technique, alors il respectera la spécification naturelle du problème.

4.2. : Contexte et suite de déclarations compilable

Soit S une suite de déclarations $<dec_1>...<dec_n>$
Nous dirons que D est l'ensemble de descripteurs associé à S
Ssi D est composé des éléments des l_i, où l_i est la liste de descripteurs associée à $<dec_i>$.

proposition

L'ensemble de descripteurs associé à une suite de déclarations compilables est un contexte.

démonstration

Soit C, cet ensemble de descripteurs. Nous devons vérifier que C respecte les conditions nécessaires pour être un contexte.

1. 2 descripteurs de même nature ont des noms différents

Si tel n'était pas le cas, il existerait 2 descripteurs de même nature et de même nom . Ceci est impossible sinon S ne serait pas une suite compilable. En effet, un programme Fortran ne peut avoir 2 variables simples, tableaux, externals, commons ou labels common de même nom.

2. A un nom de descripteur de tableau ne correspond aucun descripteur de même nom, sauf éventuellement un descripteur de common.

   En effet, la norme Fortran stipule qu' un nom de tableau déclaré ne peut apparaître que dans une déclaration de common.

3. A un nom de descripteur de label common ne correspond aucun descripteur de même nom.

   En effet, une des restructions imposée au langage Fortran p2.6 stipule qu'un nom de label common ne peut apparaître qu'une fois dans un programme.

4. A un nom de descripteur de common ne correspond aucun descripteur de même nom sauf éventuellement un descripteur de déclaration simple ou de tableau.

   En effet, la norme Fortran et les restrictions effectuées spécifient qu'un common ne peut être éventuellement qu'une variable simple ou un tableau.

5. A un nom de descripteur d'external ne correspond aucun descripteur de même nom, sauf éventuellement un descripteur de variables simples.

   En effet, d'après la norme Fortran et les restrictions effectuées, un external ne peut-être qu'un nom de procédure ou un nom de fonction. Dans ce dernier cas, celle-ci devra figurer dans une déclaration de variable simple.

4.3. : Correction sémantique et instruction compilable

4.3.1. : Déclaration

Soit <dec>,une déclaration
S ,une suite de déclarations compilable
C ,le contexte associé à S

proposition :

<dec>est une déclaration sémantiquement correcte dans C
Ssi <dec> est une déclaration compilable par rapport à S.

démonstration :

- <dec> est une déclaration compilable par rapport à S
  -> <dec> est une déclaration sémantiquement correcte dans C.
  S<dec> est donc une suite de déclarations compilable
  . soit l, la liste de descripteurs associée à <dec>.
  Nous avons - l ∩ C = Ø car S<dec> est compilable;
  sinon, une même déclaration serait deux fois présente, ce qui est impossible.
  - les descripteurs de l sont distincts deux à deux; sinon, une même déclaration serait deux fois présente au sein d'une même instruction.
  . soit C', le contexte associé à S<dec> .
  Nous avons alors C' = C U l
  Nous obtenons alors que (l L-ad C) et donc que <dec> est une déclaration sémantiquement correcte dans C.

- <dec> est une déclaration sémantiquement correcte dans C
  -> <dec> est une déclaration compilable par rapport à S.
  Soit l, la liste de descripteurs associée à <dec>.
  Nous avons donc que (l L-ad C); c'est à dire que
  1. C ∩ l = Ø
  2. C U l est un contexte
  3. les descripteurs de l sont distincts deux à deux.
  Nous devons montrer que dans ces conditions, toutes les règles Fortran sont respectées pour que S<dec> soit une suite de déclarations compilable; ou encore que <dec> soit une déclaration compilable par rapport à S.
  Les conditions 1. et 3. nous montrent que rien ne peut-être déclaré deux fois.
  La condition 2. montre que les déclarations de <dec> ne sont pas en désaccord avec S ni avec d'autres déclarations de dec .
  Il faudrait prouver que si C U l est un contexte avec 1. et 3., alors toutes les règles Fortran sont respectées pour que <dec> soit une déclaration compilable par rapport à S.
  Pour montrer cela, le plus facile est de montrer que si une des règles Fortran n'est pas respectée, alors une des propositions 1. 2. ou 3. ne serait pas respectée. Nous n'énumérerons pas ici toutes les erreurs possibles, mais nous donnerons quelques exemples :

- si <dec> contient une déclaration de variable simple dont le nom est déjà utilisé comme variable simple d'un autre type.

  alors C U 1 ne sera pas un contexte
- de même si <dec> contient une déclaration de variable simple dont le nom est déjà utilisé comme tableau
- etc...

4.3.2. : Affectation

Soit <aff>, une affectation
S , une suite de déclarations compilable
C , le contexte associé à S

proposition :

<aff> est une affectation sémantiquement correcte dans C
Ssi <aff> est une affectation compilable par rapport à S

Pour montrer cela, il faut montrer que les conditions de correction sémantique sont équivalentes aux règles Fortran déterminant si une affectation est compilable par rapport à S.

Pour ce faire, nous pourrions montrer que si une des conditions de correction sémantique n'est pas respectée, alors <aff> ne serait pas compilable par rapport à S. De même, si une des règles Fortran n'était pas respectée, <aff> ne serait pas sémantiquement correcte dans C.

Nous ne ferons pas ici cette énumération qui serait trop longue.

4.3.3. : Condition

Soit <cond>, une condition
S , une suite de déclarations compilable
C , le contexte associé à S

proposition :

< cond> est une condition sémantiquement correcte dans C
Ssi <cond> est une condition compilable par rapport à S.

Pour montrer cela, la même démarche qu'au point 4.3.2. peut-être appliquée. De plus, il existe beaucoup de similitudes entre affectation et condition car toutes deux font appel à la notion de correction sémantique d'une expression.

Notons que nous pourrions déterminer α faisant partie de l'instruction du programme compilable comme suit :

IF (<cond>) GOTO 10
10 CONTINUE

4.4. : Correction lexicale et syntaxique

En Fortran, une affectation, une déclaration ou une condition est une suite de caractères; or nous les considérons comme des suites de symboles de base. Remarquons que cette suite de caractères est une représentation externe de cette suite de symboles de base. Le concept de représentation externe d'une affectation, déclaration et condition correspond donc à la notion d'affectation, déclaration et condition Fortran moyennant les restrictions énoncées au point 1.4.2., page 2.6.

Pour qu'une affectation, déclaration ou condition soit compilable par rapport à une suite de déclarations, il faut qu'elle soit une affectation, déclaration ou condition Fortran; c'est à dire respectant un ensemble de règles souvent appelées syntaxiques.

Ainsi si une chaîne de caractères n'est pas lexicalement correcte ou si la suite de symboles de base dont elle est la représentation externe n'appartient pas à la catégorie syntaxique adéquate, alors cette chaîne de caractères ne sera pas une déclaration, affectation ou condition Fortran et vice versa.

Pour montrer cela, il faudrait montrer que la correction lexicale et les règles de productions BNF introduites sont équivalentes aux règles syntaxiques Fortran assorties des quelques restrictions énoncées.

4.5. : Soustraction de descripteurs

Soit S, une suite de déclarations compilable $<dec_1>,\ldots$
$\ldots<dec_i>,\ldots<dec_n>$
C, le contexte associé à S
l_i, la liste de descripteurs associée à $<dec_i>$

proposition :
Le contexte associé à la suite de déclarations compilables $<dec_1>,\ldots<dec_{i-1}>,<dec_{i+1}>,\ldots<dec_n>$ est L-Sub(C,l_i)

démonstration :

Si S est une suite de déclarations compilable, alors S' = S - <dec_i> est toujours compilable (moyennant la remarque du point 1.5.). Le contexte associé à S' sera $l_1 U \ldots U l_{i-1} U l_{i+1} \ldots U l_n$; soit C' ce contexte. Nous obtenons C' = C - l_i = L-Sub(C,l_i)

remarques :

- la spécification technique de l'analyseur est plus générale pour la destruction que la spécification naturelle. En effet, elle permet de détruire n'importe quelle déclaration; donc en particulier une déclaration existante.
- la remarque du point 1.5. n'est plus significative avec la spécification technique car, après une destruction, l'ensemble de descripteurs résultats sera toujours un contexte.

4.6. : Justification de la spécification technique

Soit S,une suite de déclarations mémorisée par l'analyseur
<dec_1>...<dec_i>...<dec_n>
C,le contexte associé à S
l_i, la liste de descripteurs associée à <dec_i>
<dec_{n+1}>, une déclaration

proposition :

Si l'analyseur respecte la spécification technique
Alors il respecte la spécification naturelle.

démonstration :

1. cas d'un traitement
   * affectation et condition
     Les propriétés établies aux points 4.3.2., 4.3.3., et 4.4. établissent la proposition.
   * déclaration <dec_{n+1}>
     Les propriétés établies aux points 4.3.1. et 4.4. avec le fait que le contexte associé à la suite de déclarations mémorisée S<dec_{n+1}> est L-Add(C,l_{n+1}).
2. cas d'une destruction
   * affectation et condition
     La suite dedéclarations mémorisée sera inchangée, de même que le contexte courant qui sera encore le contexte associé à S.

* déclaration $<dec_i>$

La destruction de $<dec_i> \in S$, d'après la spécification technique, établira que le contexte courant sera $L\text{-}Sub(C,l_i)$.

D'après la propriété établie au point 4.5., la suite de déclarations mémorisée à laquelle est associée ce contexte sera $<dec_1>...<dec_{i-1}><dec_{i+1}>...<dec_n>$ comme le décrit la spécification naturelle.

Chapitre II :C O N S T R U C T I O N E T V A L I D A T I O N

1. : INTRODUCTION

Notre but sera de construire le programme ANEXPR spécifié ch.I (point 3.4.7.). La construction de ce programme n'étant pas évidente, nous le construirons étape par étape en essayant de raisonner le plus rigoureusement possible. De plus, certaines parties seront démontrées de différentes façons pour illustrer quelques méthodes de démonstration.

Sachant que nous devons obtenir un programme Fortran, donc non récursif, nous procéderons de la façon suivante :

- Supposons que nous disposons de la récursivité.
  1. construire un ensemble de programmes qui ne fait que l'analyse syntaxique :
     a) s'il n'y a pas d'erreur à détecter
     b) s'il peut y avoir des erreurs à détecter.
  2. construire un ensemble de programmes qui fait l'analyse syntaxique et sémantique :
     a) s'il n'y a pas d'erreur sémantique à détecter
     b) s'il peut y avoir des erreurs sémantiques à détecter.
  3. construire le programme ANEXPR.
- Elimination de la récursivité.
  4. intégration des différents programmes.
  5. élimination des appels récursifs.

2. : ANALYSE SYNTAXIQUE

2.1. : Sous-problèmes

Nous spécifierons 8 sous-problèmes :

SXLEXP : analyse syntaxique d'une expression logique
SXCONJ : d'une conjonction
SXNEGA : d'une négation
SXPATO : d'une proposition atomique
SXAEXP : d'une expression arithmétique
SXTERM : d'un terme
SXFACT : d'un facteur
SXSEXP : d'une expression simple

Soit cat ∈ {LEXP, CONJ, NEGA, PATO, AEXP, TERM, FACT, SEXP}
Spécification de SXcat :

Soit SSt = S
S'il existe un préfixe P cat-max(<cat>) dans S
Alors l'exécution de SXcat traitera ce préfixe
Sinon l'exécution de SXcat
- établira ERREXP = .TRUE.
- remplira le message d'erreur appartenant à l'environnement
- s'arrêtera immédiatement ainsi que toutes les procédures SXcat initialisées jusque là.

2.2. : SXLEXP : analyse syntaxique d'une expression logique

2.2.1. : Supposons que S possède un préfixe cat-max(<expression logique>)

Soit P ce préfixe
Nous avons alors SSt = PS' où P = C{.OR.P'}
avec C ∈ cat(<conjonction>)
P' ∈ cat(<expression logique>)
{} signifie éventuellement

1. exécutons SXCONJ
Avant cette exécution, C est le préfixe cat-max(<conjonction>) de P, donc également de S car
* .OR. n'est pas un continuateur de C dans <conjonction> et le premier symbole de S', s'il existe, n'est pas un continuateur de P dans <conjonction> car il n'est pas un continuateur d'expression logique.
* C est préfixe de P et de S
* C ∈ cat(<conjonction>)
Après cette exécution nous aurons SSt = {.OR.P'} S'
2. Si SSt est vide, on a fini car P = C et SSt = S' = λ (λ signifie vide)

Sinon, soit s, le premier symbole de SSt
si s = .OR. alors SSt = .OR.P'S'
- on supprime s; SSt devient P'S'
- on réexécute SXLEXP
sinon SSt = S' et on a fini

2.2.2. : Supposons que S ne possède pas de préfixe cat-max(<expression logique>)

Soit Q, le plus grand préfixe de S qui est préfixe d'une expression logique P.

P = C {.OR.P'} avec C ∈ cat(<conjonction>)

P' ∈ cat(<expression logique>)

Le dernier symbole de Q est soit dans C

soit .OR.

soit dans P'

soit inexistant si Q = λ

1° cas : Q = λ

Dans ce cas, aucun préfixe de S n'est une conjonction car, si un tel préfixe existait, celui-ci serait préfixe de Q, ce qui est impossible puisque Q = λ .

Donc, il n'existe pas de préfixe cat-max(<conjonction>) de S; l'exécution de SXCONJ va détecter une erreur.

2° cas : le dernier symbole de Q est dans C

a) Q = C

Dans ce cas, on ne peut avoir S = Q.OR.S' sinon Q.OR. serait un préfixe de S, plus grand que Q, d'une expression logique.

Donc Q n'est pas suivi dans S d'un continuateur de Q dans <expression logique> et est une conjonction, donc une expression logique. Q serait alors préfixe cat-max(<expression logique>) dans S; ce qui est impossible vu que l'on suppose qu'un tel préfixe n'existe pas.

b) Q est un préfixe propre de C

Montrons qu'il n'existe pas de préfixe cat-max(<conjonction>) de S. En effet, supposons qu'un tel préfixe existe; soit C' ce préfixe et s, le symbole de base suivant ce préfixe dans S.

(i) s ne peut être un continuateur de conjonction
sinon C' ne serait pas cat-max(<conjonction>) de S

(ii) s doit être un continuateur d'expression logique sinon C' serait également préfixe cat-max(<expression logique>) de S, ce qui est contraire à nos suppositions.

(i) et (ii) ⇒ s doit être le symbole .OR., ce qui est impossible car C's serait un préfixe de Q, et Q ne pourrait être un préfixe de C car C est une conjonction et ne peut donc contenir de symbole .OR.

Donc, il n'existe pas de préfixe cat-max(<conjonction>) de S; l'exécution de SXCONJ va détecter une erreur.

3° cas : le dernier symbole de Q est .OR.

Dans ce cas, la nouvelle exécution de SXLEXP détectera l'erreur (cas où Q = λ).

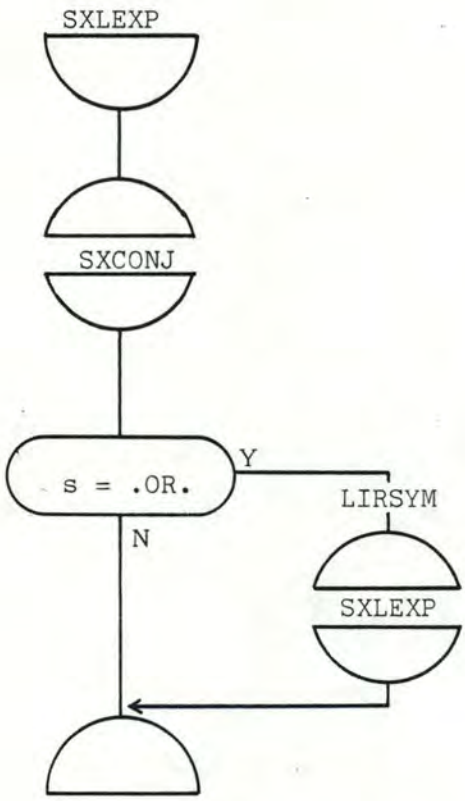
4° cas : le dernier symbole de Q est dans P'

Dans ce cas également, la nouvelle exécution de SXLEXP détectera l'erreur.

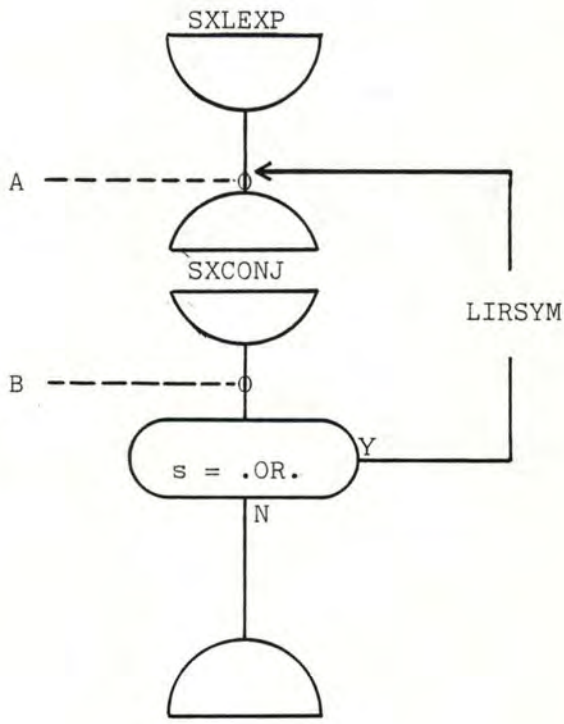
2.2.3. : Algorithme

Dans les algorithmes qui suivront, s représente le premier symbole de base de SSt, appelé aussi symbole de base courant, avec la convention que si SSt est vide, alors s aura pour valeur la représentation d'un symbole de base fictif, différent de tous les symboles existants.

Algorithme A1



Algorithme A2



L'algorithme A2 est équivalent à A1; on a remplacé l'appel récursif par une boucle car il n'y a plus rien à faire quand l'exécution de cet appel récursif sera terminée.

2.2.4. :Démonstrations de correction

Nous allons essayer de montrer que le programme SXLEXP respecte ses spécifications. Nous supposerons que SXCONJ est correct et que le préfixe P cat-max(<expression logique>) dans S,existe. Si celui-ci n'existait pas, c'est SXCONJ qui détecterait l'erreur et arrêterait immédiatement l'analyse.

2.2.4.1. : Assertions inductives

Soit SSt = S = PS' avec P' cat-max(<expression logique>) dans S
Le programme est correct par rapport aux pré et post conditions suivantes : PRE : {SSt = PS'}
POST : {SSt = S'}
Les assertions A et B seront vérifiées pour tout passage en A et B :
A : {il existe P'cat-max(<expression logique>) dans P'S'
et SSt = P'S'}
B : {SSt = S' ou il existe P'cat-max(<expression logique>) dans P'S' et SSt = .OR.P'S'}

. PRE -> A
évident : il suffit de prendre P' = P

. (B et s ≠ .OR) -> POST
La suite des symboles à traiter ne commence pas par .OR.
On a donc SSt = S'

. {A} SXCONJ {B}
P' aura l'une des deux formes suivantes :
P' = C avec C cat-max(<conjonction>) dans P'S'
P' = C.OR.P" avec C cat-max(<conjonction>) dans P'S'
P" cat-max(<expression logique>) dans P"S'
Après exécution de SXCONJ, nous obtenons alors
SSt = S' ou
SSt = .OR.P"S'
et l'assertion B est donc vérifiée

. {B et s = .OR.} LIRSYM {A}
Comme SSt commence par .OR., on ne peut avoir SSt = S', sans quoi S' commencerait par .OR. et P ne serait donc pas cat-max(<expression logique>). On est donc dans le cas où SSt = .OR.P'S'.
L'exécution de LIRSYM établit alors SSt = P'S' et l'assertion A est donc vérifiée.

. Terminaison :

Sst est une suite finie de symboles de base.
Soit n, le nombre de symboles de base de SSt : $n \geq 0$
A chaque nouveau passage au point A, on est assuré que n diminue vu l'appel à LIRSYM.

2.2.4.2. : Raisonnement descendant sur la structure des données

Nous raisonnerons par induction sur l'ensemble des expressions logiques, bien fondé par la relation suivante :
Soit B et B' deux expressions logiques
B'<B Ssi B est de la forme C.OR.B' où C est un conjonction.
Les éléments minimaux de cet ensemble sont toutes les conjonctions
a) le programme est correct pour les éléments minimaux
En effet, dans ce cas, SSt = CS' et l'exécution de SXCONJ va établir SSt = S' car C est préfixe cat-max(<conjonction>) dans S.
De plus, s ≠ .OR. et le programme SXLEXP se termine avec SSt = S'
b) si le programme est correct pour P', il est correct pour P = C.OR.P'
SSt = C.OR.P'S' avec C ∈ cat(<conjonction>)
P' cat-max(<expression logique>) dans P'S'
Après exécution de SXCONJ, SSt = .OR.P'S' et s = .OR.
Après exécution de LIRSYM, SSt = P'S'
Comme P'<P, par hypothèse d'induction, le programme se terminera avec SSt = S'
Le programme est donc correct pour toute expression logique P cat-max(<expression logique>) dans PS'.

2.2.4.3. : Raisonnement ascendant sur la structure des données

Montrons que s'il existe un passage en A tel que SSt = PS' avec P préfixe cat-max(<expression logique>) et composé de n conjonctions (n>1), alors pour tout i : $1 \leq i \leq n$, il existe un passage en B tel que SSt = KS' où K est composé des i-1 dernières conjonctions de P, chacune précédée du symbole .OR.
a) vrai pour i = n
Après la première exécution de SXCONJ, nous arrivons au point B avec SSt = KS' où K est composé des n-1 dernières conjonctions de P, chacune précédée du symbole .OR.

b) si vrai pour i_o+1, alors vrai pour i_o $(1 \leq i_o < n)$

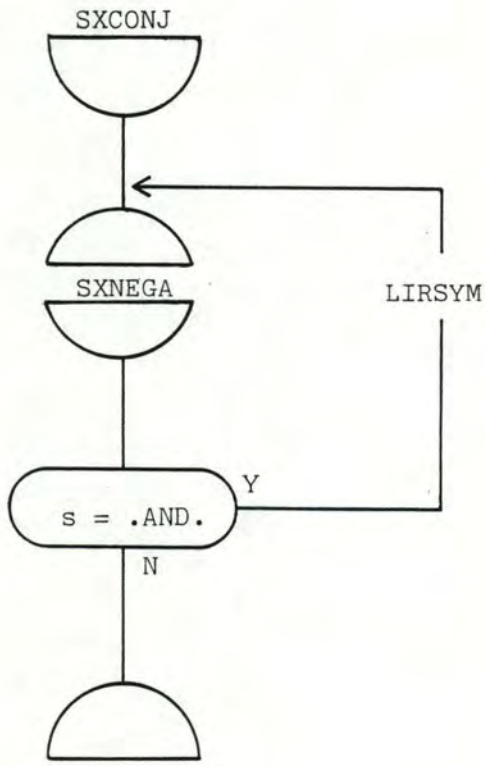
En effet, si c'est vrai pour i_o+1, il existe un passage au point B avec SSt = KS' où K est composé des i_o dernières conjonctions de P, chacune précédée du symbole .OR.

Comme $i_o \geq 1$, s = .OR. et après exécution de LIRSYM et de SXCONJ, on arrivera au point B avec SSt = K'S' où K' est composé des i_o-1 dernières conjonctions de P, chacune précédée du symbole .OR.

La correction du programme s'établit en prenant i = 1. On arrive donc au point B avec SSt = S' et donc s ≠ .OR. et le programme se termine.

2.3. : SXCONJ : analyse syntaxique d'une conjonction

Nous ne referons pas les raisonnements. Ceux-ci sont semblables à ceux effectués pour SXLEXP. Nous obtenons ainsi l'algorithme suivant :



2.4. : SXNEGA : analyse syntaxique d'une négation.

2.4.1. : Supposons que S possède un préfixe cat-max(<négation>)

Soit P ce préfixe

Nous avons alors SSt = PS' où P = {.NOT.} T

avec T ∈ cat(<proposition atomique>)

si P = T, alors s ≠ .NOT. et il suffit d'exécuter SXPATO

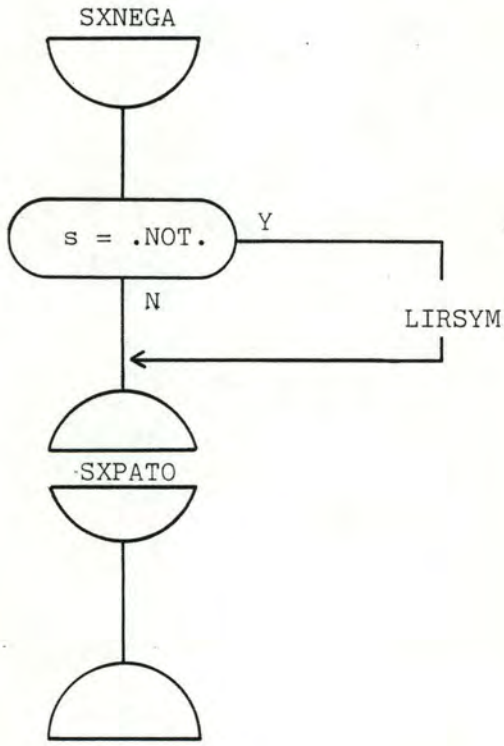
si P = .NOT. T, alors il suffit d'exécuter LIRSYM et ensuite d'exécuter SXPATO.

2.4.2. : Supposons que S ne possède pas de préfixe cat-max(<négation>

si s ≠ .NOT., alors il n' existe pas de préfixe T cat-max (<proposition atomique>) dans S. L'exécution de SXPATO va détecter une erreur.

si s = .NOT., alors, après avoir exécuté LIRSYM, il n'existera pas de préfixe T cat-max(<proposition atomique>) dans S amputé de son 1° symbole. L'exécution de SXPATO va détecter une erreur.

2.4.3. Algorithme



2.5. : SXPATO : analyse syntaxique d'une proposition atomique

2.5.1. : Supposons que S possède un préfixe cat-max(<proposition atomique>)

Soit P, ce préfixe

Nous avons alors SSt = PS' où P = $A_1 \{\omega\ A_2\}$

avec $A_1, A_2 \in$ cat(<expression arithmétique>)

$\omega \in$ cat(<op rel>)

1) exécutons SXAEXP .

Avant cette exécution, A_1 est le préfixe cat-max(<expression arithmétique>) de P, donc de S .

Après cette exécution, nous aurons SSt = $\{\omega\ A_2\}$ S'.

2) Si SSt est vide ou si $s \notin$ cat(<op rel>)

on a fini car SSt = S'

Si $s \in$ cat(<op rel>),

on exécute LIRSYM, SSt devient alors A_2S'

on exécute SXAEXP et comme A_2 est préfixe cat-max(<expression arithmétique>) de A_2S',

on obtient SSt = S' et on a donc fini.

2.5.2. : supposons que S ne possède pas de préfixe cat-max(<proposition atomique>)

Le même raisonnemnet que celui pour SXLEXP peut être effectué avec cependant quelques petites différences :

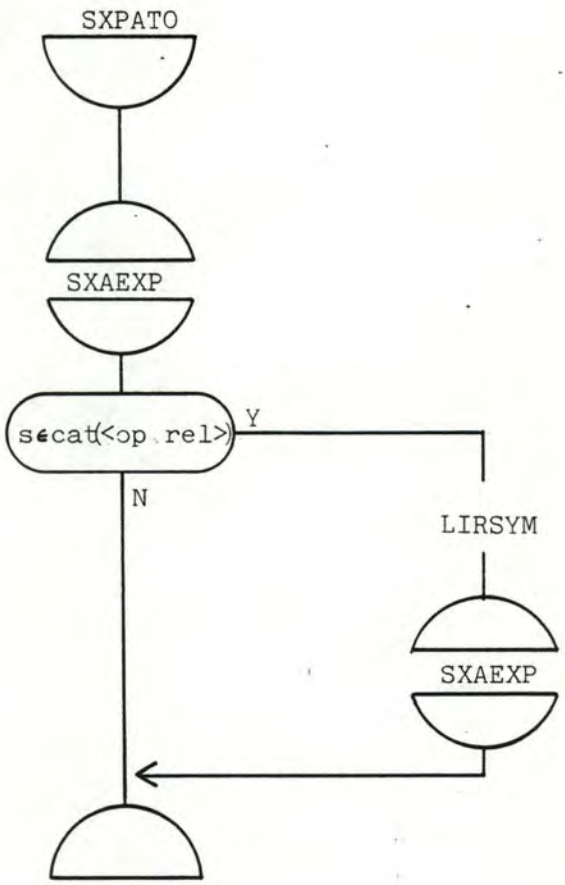
3° cas : le dernier symbole de Q est ω

Dans ce cas, on va réexécuter SXAEXP et l'on est dans la même situation que celle où Q = λ.

4° cas : le dernier symbole de Q est dans A_2

Dans ce cas on va réexécuter SXAEXP et l'on est dans la même situation que celle où le dernier symbole de Q est dans A_1 .

2.5.3. : Algorithme



2.6. : SXAEXP : analyse syntaxique d'une expression arithmétique

Nous pourrons effectuer la même démarche que pour SXLEXP avec les différences suivantes :

P sera de la forme $\{\omega_1\}$ T $\{\omega_2 P'\}$

avec $\omega_1, \omega_2 \in$ cat(<op add>)

T ∈ cat(<terme>)

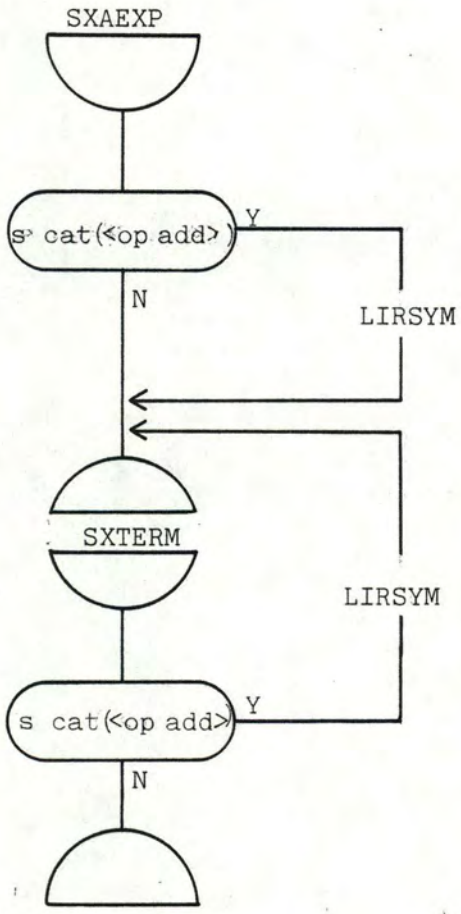
$\omega_2 P' \in$ cat(<expression arithmétique>)

Si P existe, il faut tout d'abord voir si s cat(<op add>); si tel est le cas, il faudra exécuter LIRSYM.

Si P n'existe pas, il faut envisager le cas où Q = ω_1.

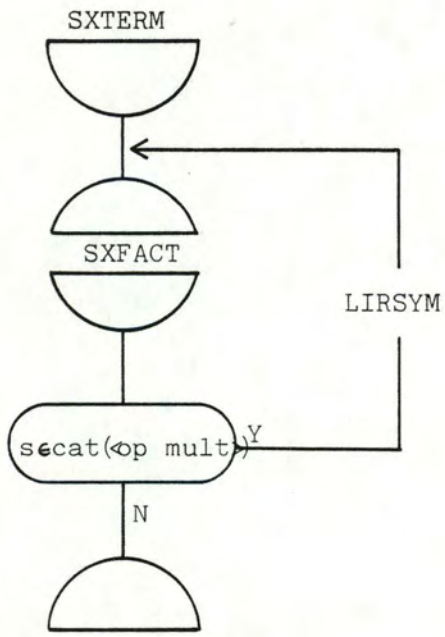
Dans ce cas, après exécution de LIRSYM, on se retrouve dans le cas où Q = λ.

On obtient l'algorithme suivant:



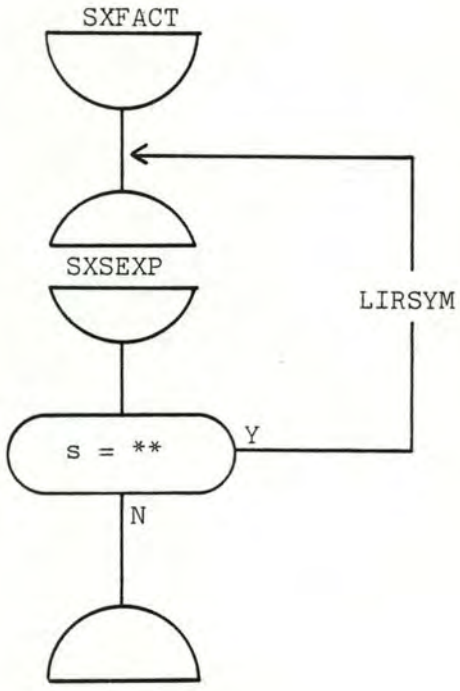
2.7. : SXTERM : analyse syntaxique d'un terme

Les raisonnements sont semblables à ceux effectués pour SXLEXP. Nous obtenons l'algorithme suivant :



2.8. : SXFACT : analyse syntaxique d'un facteur

Les raisonnements sont semblables à ceux effectués pour SXLEXP. Nous obtenons l'algorithme suivant :



2.9. : SXSEXP : analyse syntaxique d'une expression simple

C'est dans cette partie que seront détectées les erreurs syntaxiques. Si tel est le cas, ERREXP sera mis à .TRUE., le message d'erreur appartenant à l'environnement sera rempli et toute analyse de l'expression sera arrêtée.

2.9.1. : problème auxiliaire

fonction ISCMPX type :LOGICAL

effet : Soit SSt = S

S'il existe un préfixe cat-max(<complexe>) dans S,

Alors l'exécution de ISCMPX établira -ISCMPX = .TRUE.

-le préfixe C sera traité.

Sinon soit -ERREXP sera mis à .TRUE.

-le message d'erreur sera rempli

-l'analyse sera arrêtée

(dans ce cas, il n'existe pas de préfixe cat-max (<expression logique>) dans S)

soit -ISCMPX sera mis à .FALSE.

-SSt ne sera pas modifié.

Cette fonction ne sera pas construite ici. Elle fait appel à la notion de représentation interne de la chaîne de caractères courante, représentation que nous n'avons pas exposée ici. Remarquons seulement que si lors de la recherche d'une constante complexe, on remarque qu'il y a une erreur syntaxique, on la traitera immédiatement au lieu de la laisser découvrir par la suite de l'analyse.

2.9.2. : Supposons que S possède un préfixe cat-max(<expression simple>)

Soit P ce préfixe

Nous avons alors SSt = PS' où P est d'une des 5 formes ci-dessous.

1. P = id
2. P = id(expr$_1$,...,expr$_n$)
3. P = cst
4. P = (expr)
5. P = (cmpx)

avec id $\in$ cat(<identificateur>)

expr, expr$_i$ $\in$ cat(<expression>)

cst $\in$ cat(<constante>)

cmpx $\in$ cat(<complexe>)

* Si s = (

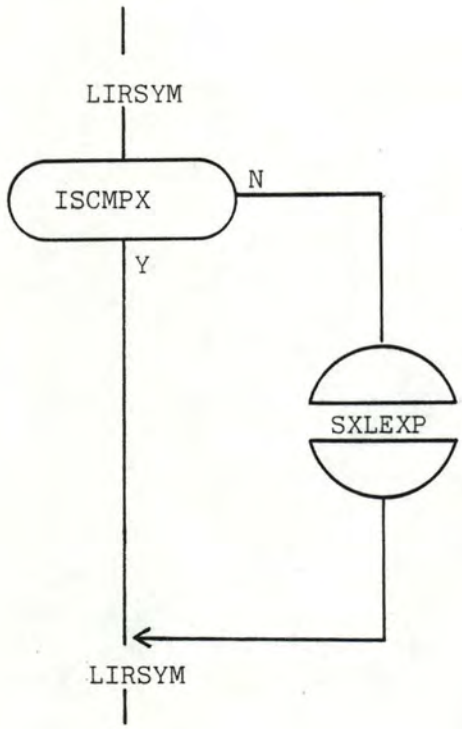
P sera de la forme 4 ou 5

Il suffit -d'exécuter LIRSYM .

-Si ISCMPX = .TRUE. Alors, nous obtenons SSt =)S'

Sinon, P est de la forme 4 et l'exécution de SXLEXP traitera le préfixe expr qui est cat-max(< expression >) dans expr)S' nous obtenons encore SSt =)S'.

-d'exécuter LIRSYM



* Si s = cst

P sera de la forme 3

Alors il suffit d'exécuter LIRSYM et on a fini

* Si s = id

P sera de la forme 1 ou 2.

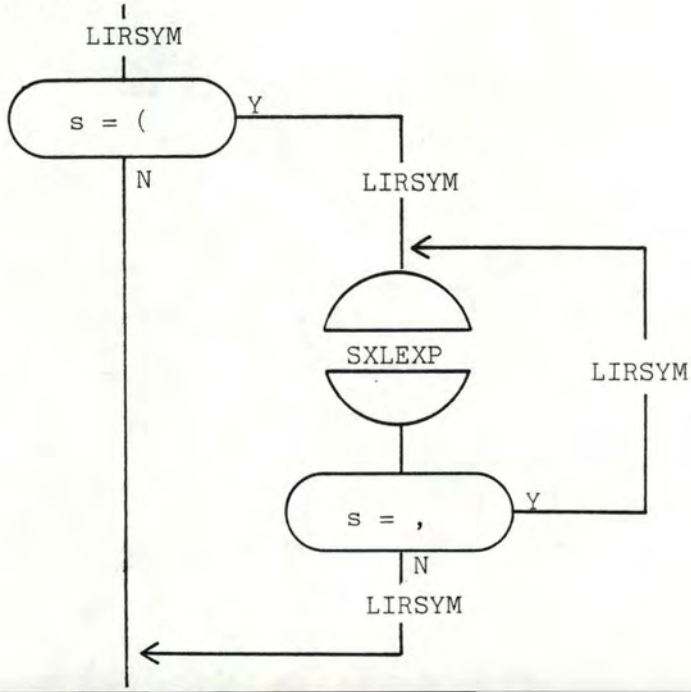
- exécutons LIRSYM .
- Si s ≠ (, P est de la forme 1 et on a fini

  sinon - exécutons LIRSYM

  SSt = $exp_1, \ldots, exp_n)S'$ où n>0

  - exécutons SXLEXP
  - si s ≠ , on exécute LIRSYM et on a fini

    sinon on exécute LIRSYM et on se retrouve dans la situation précédente, mais avec moins d'expressions à traiter.



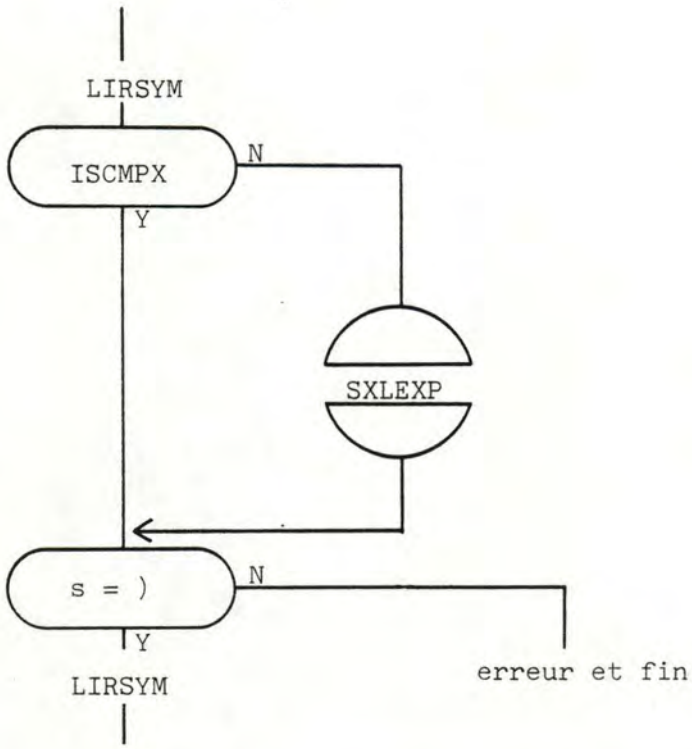
Nous pouvons montrer que, après exécution de la boucle, on a SSt =)S'
Soit n, le nombre d'expressions entre parenthèses
Pour n = 1 : après la première exécution de SXLEXP, on a SSt =)S'
et donc s ≠ ,
Si vrai pour n_o, alors vrai pour n_o+1.
Après la première exécution de SXLEXP, on a
SSt = ,exp_2,...,exp_{n_o+1})S' et s = ,
après exécution de LIRSYM, on se retrouve dans la situation initiale avec n_o expressions.
Par hypothèse, cette boucle se termine et on aura SSt =)S'.

2.9.3. : Supposons que S ne possède pas de préfixe cat-max(<expression simple>)

* si s ≠ (et n'est ni une constante ni un identificateur
Alors il y a une erreur; il faudra -mettre ERREXP à .TRUE.
-remplir le message d'erreur
-arrêter l'analyse

* si s = cst
impossible sinon s serait préfixe cat-max(<expression simple>)

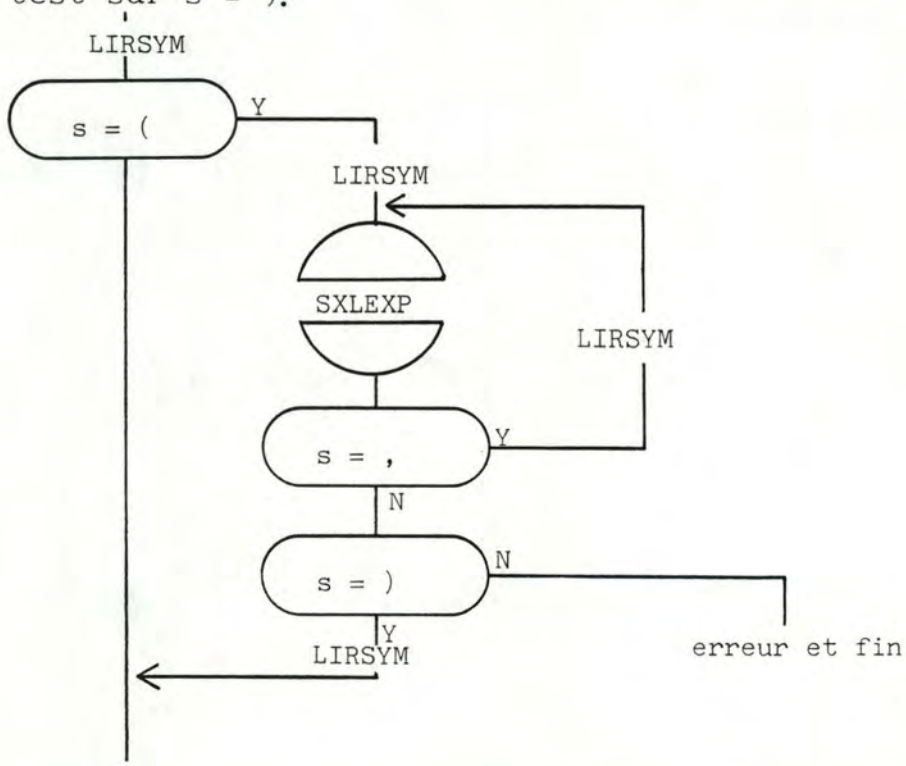
* si s = (
soit ISCMPX détectera une erreur et arrêtera l'analyse
soit ISCMPX sera .TRUE.. Dans ce cas, après exécution de ISCMPX, on ne pourra avoir s =) sinon un préfixe cat-max(<expression simple>) existerait. L'erreur sera donc détectée par un test sur s =)
soit ISCMPX sera .FALSE. Dans ce cas,
soit l'exécution de SXLEXP détectera une erreur et arrêtera l'analyse
soit exp existe et après exécution de SXLEXP on ne pourra avoir s =) sinon un préfixe cat-max(expression simple) existerait. L'erreur sera donc détectée par un test sur s =)



remarque : "erreur et fin" signifie -mettre ERREXP à .TRUE.
-remplir le message d'erreur de l'environnement
-arrêter toute analyse.

* si s= id

Dans ce cas, après exécution de LIRSYM, on doit avoir s = (, sinon id serait préfixe cat-max(<expression simple>) dans S. L'erreur sera détectée soit lors d'une exécution de SXLEXP, soit à la fin de la boucle où on aura s ≠) sinon un préfixe cat-max(<expression simple>) existerait. L'erreur sera donc détectée par un test sur s =).



3. : ANALYSE SEMANTIQUE

3.1. : Sous-problèmes

Nous spécifierons 8 sous-problèmes :

ANLEXP : analyse d'une expression logique
ANCONJ : d'une conjonction
ANNEGA : d'une négation
ANPATO : d'une proposition atomique
ANAEXP : d'une expression arithmétique
ANTERM : d'un terme
ANFACT : d'un facteur
ANSEXP : d'une expression simple

Soit cat $\in$ {LEXP, CONJ, NEGA, PATO, AEXP, TERM, FACT, SEXP}

Spécification de ANcat :

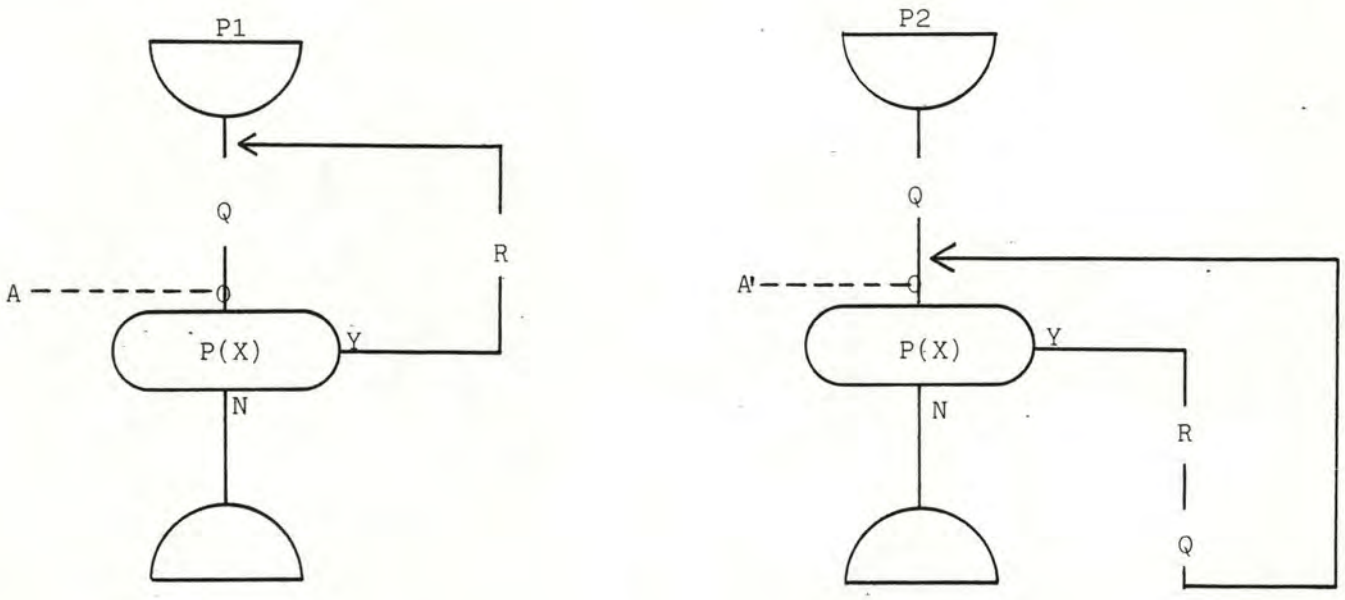
Soit SSt = S
cntc = C
S'il existe un préfixe P, cat-max(<cat>) dans S, et si P est une expression sémantiquement correcte dans C
Alors l'exécution de ANcat
-traitera ce préfixe
-augmentera la pile PILTYP d'un élément qui sera le type résultat de P dans C.
Sinon l'exécution de ANcat
-établira ERREXP = .TRUE.
-remplira le message d'erreur appartenant à l'environnement
-s'arrêtera immédiatement ainsi que toutes les procédures ANcat initialisées jusque là.

3.2. : ANLEXP : analyse d'une expression logique

3.2.1. : Supposons que P existe et est sémantiquement correcte dans C

3.2.1.1. : Schéma de programme

Montrons tout d'abord l'équivalence des deux algorithmes suivants où Q et R sont des séquences d'instructions, P(x) un test sur l'ensemble x des variables.



Considérons simultanément une exécution de P1 et de P2, pour les mêmes valeurs initiales des variables.

A tout passage au point A avec $x = x_o$, correspondra un passage au point A' avec également $x = x_o$.

En effet, raisonnons par récurrence sur le nombre n de passages au point A (n est supposé fini).

- vrai pour le premier passage : évident
- si vrai pour le $i^{ème}$ passage, alors vrai pour le $i+1^{ème}$ $(i < n)$
  comme $i < n$, nous avons $p(x_o)$ vrai et le programme P1 va exécuter R, puis Q et reviendra au point A avec $x = x'_o$. De même pour P2 qui va exécuter R, puis Q et reviendra donc au point A' avec $x = x'_o$.

A tout passage au point A' avec $x = x_o$, correspondra un passage au point A avec également $x = x_o$

L'équivalence de P1 et P2 s'établit en prenant i = n. Les deux programmes se terminent donc avec la même valeur de x.

3.2.1.2. construction du programme

* En prenant Q = SXCONJ, R = LIRSYM et p(x) = (s=.OR.), nous obtenons l'algorithme de SXLEXP.

Nous avons montré dans le raisonnement ascendant que

si SSt = P S' avec $P = c_1.OR.c_2 \ldots .OR.c_n$

avec $C_i \in$ cat(<conjonction>)

alors pour tout i : $1 \leqslant i \leqslant n$

il existe un passage au point A avec

SSt = $.OR.C_{i+1}.OR. \ldots .OR.C_n$ S'

* Modifions le programme P2 tel que cette propriété soit encore vraie, mais également que :

PILTYP = | type (C1.OR. ... .OR.C_i) |
|---|
| P_o |

où P_o est la valeur initiale de PILTYP

* Pour que ces propriétés soient vérifiées pour i = 1, remplaçons SXCONJ par ANCONJ.
* Pour que ces propriétés soient vérifiées pour i > 1, remplaçons la seconde occurence de SXCONJ par ANCONJ et ajoutons-y les instructions suivantes :

t2 ⇐ PILTYP

t1 ⇐ PILTYP

t := f(t1,T2,.OR.)

où l'opération t ⇐ P signifie retirer le sommet de la pile P s'il existe, et le metter dans t.

P ⇐ t signifie metter la valeur de t sur la pile P.

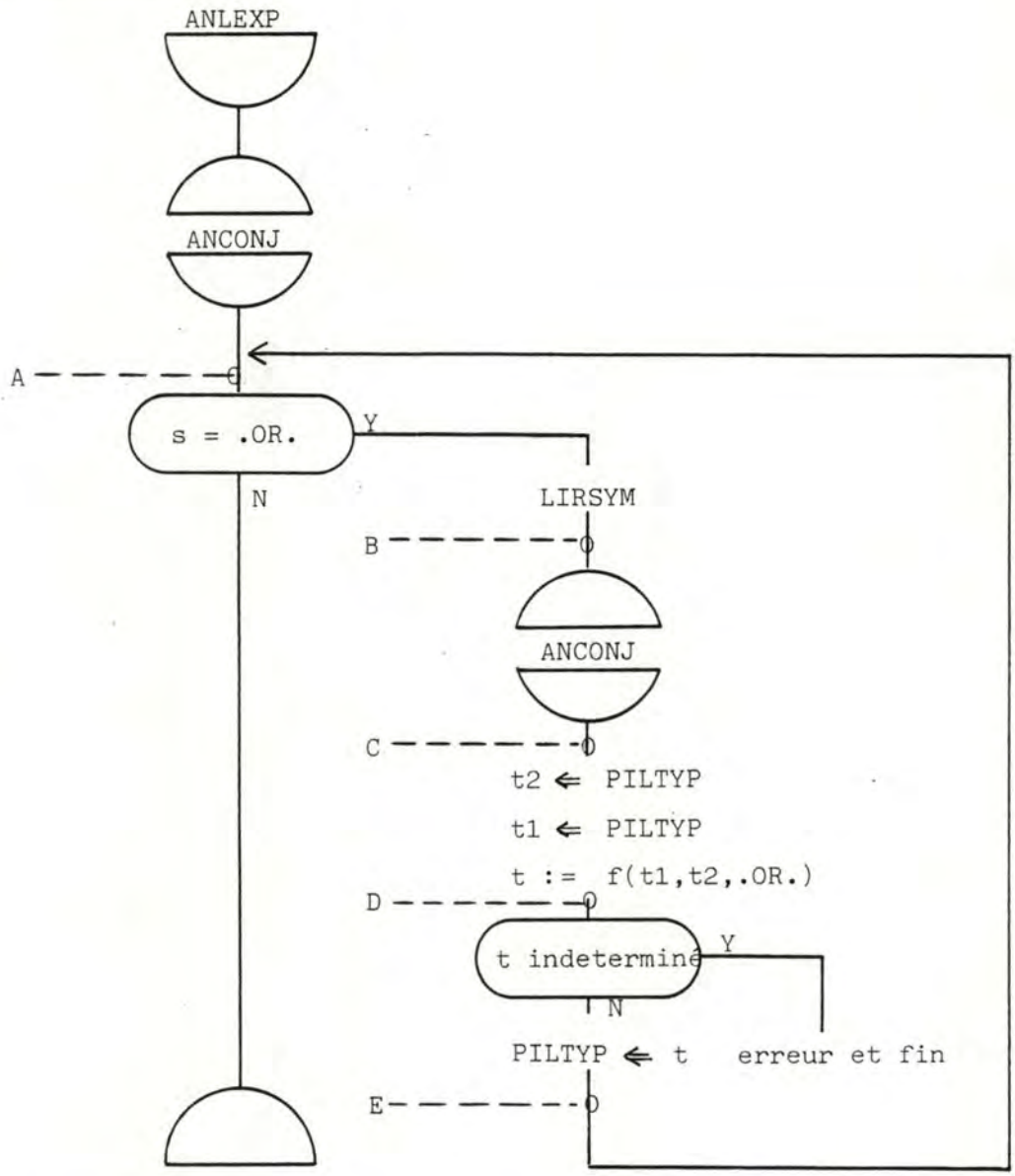
f est la fonction définie au ch I 2.4.1.

3.2.2. : retrait des suppositions

Par construction de ANLEXP qui respecte les propriétés de SXLEXP, nous ne devons pas nous préoccuper ici des erreurs syntaxiques.

Si P n'est pas sémantiquement correct dans C, cette erreur sera détectée lors d'un des appels à ANCONJ sauf, si à un des passages, on a t1 et t2 non compatibles avec l'opérateur .OR.. Supposons que dans un tel cas, f(t1,t2,.OR.) aura une valeur distincte de tous les types possibles, ce qui nous permettra de tester si f(t1,t2,.OR.) est non défini. Nous appellerons "indéterminée" cette valeur.

3.2.3. : Algorithme



3.2.4. : démonstrations de correction

Nous allons essayer de montrer que le programme ANLEXP respecte ses spécifications. Nous supposerons que ANCONJ est correct, que le préfixe P cat-max(<expression logique>) dans S existe et que celui-ci est sémantiquement correct dans C. Si P n'existait pas, ANCONJ détecterait l'erreur et arrêterait l'analyse; de même si une erreur sémantique est détectée par ANCONJ. Si une erreur sémantique était détectée par ANLEXP, ce serait parce que f(t1,t2,.OR.) est indéterminé et l'analyse s'arrêterait.

3.2.4.1. : assertions inductives

Soit SSt = S = PS' avec P, cat-max(<expression logique>) dans S sémantiquement correct dans C et composé de n conjonctions ($n \geq 1$)

Le programme est correct par rapport aux pré et post conditions suivantes :

PRE : {SSt = PS' et PILTYP = | PILE |}

POST : {SSt = S' et PILTYP = | type (P) / PILE |}

où la notation type (P) représente le type résultat de P dans C (ch I 2.4.2.2.).

L'assertion A sera vérifiée pour tout passage en A.

A : {il existe $i < n$ tel que SSt = $.OR.C_{i+1}$ ... $.OR.C_n S'$

et PILTYP = | type (C_1.OR. ... .OR.C_i) / PILE |}

* {PRE} ANCONJ {A}

après exécution de ANCONJ, nous obtenons

SSt = $.OR.C_2$ ... $.OR.C_n$

et PILTYP = | type (C_1) / PILE |

L'assertion A est donc vérifiée avec $i = 1 \leq n$.

* (A et s ≠ .OR.) ⇒ POST

(A et s ≠ .OR.) ⇒ i = n (le premier symbole de S' ne pouvant être .OR.)

Nous avons donc SSt = S'

et PILTYP = | type (C_1.OR. ... .OR.C_n) / PILE |

⇒ POST

* {A et s = .OR.} Q {A} où Q représente le corps de la boucle.

(A et s = .OR.) ⇒ $i < n$

après exécution de LIRSYM, nous obtenons l'assertion suivante :

B : {il existe $i < n$ tel que SSt = C_{i+1}.OR. ... .OR.$C_n S'$

et PILTYP = | type (C_1.OR. ... .OR.C_i) / PILE |}

Après exécution de ANCONJ, nous obtenons l'assertion suivante :

C : {il existe i < n tel que SSt = .OR.C_{i+2} OR.C_nS'

et PILTYP = | type (C_{i+1}) |
| --- |
| type (C_1.OR. ... .OR. C_i) |
| PILE |

}

Au point D, nous obtenons l'assertion suivante :

D : {il existe i < n tel que SSt = .OR.C_{i+2}... .OR.C_nS'

et t1 = type (C_1.OR. ... .OR.C_i)

et t2 = type (C_{i+1})

et t = type ((C_1.OR. ... C_i).OR.C_{i+1})

et PILTYPE = |PILE|

}

Nous obtenons l'assertion suivante au point E :

E : {il existe i < n tel que SSt = .OR.C_{i+2} ... .OR.C_nS'

et PILTYP = | type (C_1.OR. ... .OR.C_{i+1}) |
| --- |
| PILE |

}

Enfin E ⇒ A

* Terminaison

SSt étant une suite finie de symboles de base, on est assuré de la terminaison vu l'appel à LIRSYM dans le corps de la boucle. Il est clair que l'exécution de toutes les instructions élémentaires se termine.

3.2.4.2. : Raisonnement descendant sur la structure des données

Considérons le programme ANLEXP amputé du premier appel à ANCONJ.

Soit $C_1,\ldots,C_n \in$ cat(<conjonction>) et sémantiquement correctes dans C

S', une suite de symboles de base ne commençant pas par un continuateur de C_n dans <expression logique> (ch I 3.4.3.1.)

E ∈ cat(<expression>) et sémantiquement correcte dans C.

Si SSt = .OR.C_1 ... .OR.C_nS' ($n \geq 0$)

et PILTYP = | type (E) |
| --- |
| PILE |

Alors ce programme se termine et établit

SSt = S' et

PILTYP = | type (E.OR.C_1 ... .OR.C_n) |
| --- |
| PILE |

a) vrai pour n =0

En effet, si n = 0, alors s ≠ .OR. et le programme se termine avec SSt = S' et

PILTYP = | type (E) |
| PILE |

b) si vrai pour n_o, alors vrai pour n_o+ 1 ($n_o \geq 0$)

Si SSt = $.OR.C_1 \ldots .OR.C_n S'$ avec $n = n_o+1 > 0$

et PILTYP = | type (E) |
| PILE |

Après exécution de la boucle, nous obtenons :

SSt = $.OR.C_2 \ldots .OR.C_n S'$ avec $n = n_o+1 > 0$ et

PILTYP = | type (E.OR.C_1) |
| PILE |

Renumérotons les indices de telle sorte que $n' = n-1 = n_o > 0$

Par hypothèse d'induction, le programme se termine et établit :

SSt = S' et

PILTYP = | type (E.OR.$C_1 \ldots .OR.C_{n_o+1}$) |
| PILE |

La correction du programme ANLEXP en entier s'établit en remarquant que initialement SSt = $C_o.OR. \ldots .OR.C_n S'$ avec $n \geq 0$ et

PILTYP = | PILE |

Après le premier appel de ANCONJ, nous obtenons :

SSt = $.OR.C_1 \ldots .OR.C_n S'$ avec $n \geq 0$ et

PILTYP = | type (C_o) |
| PILE |

Le programme se terminera donc et établira :

SSt = S' et

PILTYP = | type (C_o.OR. ... .OR.C_n) |
| PILE |

3.2.4.3. : Raisonnement ascendant sur la structure des données

Soit P,cat-max(<expression logique>) dans S, sémantiquement correct dans C et composé de n conjonctions ($n \geq 1$).

Initialement nous avons :

SSt = $C_1.OR. \ldots .OR.C_n S'$ avec $n \geq 1$ et

PILTYP = | PILE |

Montrons que :

Pour tout $1 \leqslant i \leqslant n$; il existe un passage au point A avec

SSt = .OR.C_{i+1}... .OR.C_nS' et

PILTYP = | type (C_1.OR. ... .OR.C_i) |
|---|
| PILE |

a) vrai pour i = 1

Au premier passage au point A, nous avons

SSt = .OR.C_2... .OR.C_nS' et $n \geqslant 1$ et

PILTYP = | type (C_1) |
|---|
| PILE |

Ceci vérifie l'affirmation pour i = 1

b) si vrai pour i_o, alors vrai pour $i_o + 1$ $(1 \leqslant i_o < n)$

Il existe donc un passage au point A avec

SSt = .OR.C_{i_o+1} ... .OR.C_nS' et

PILTYP= | type (C_1.OR. ... .OR.C_{i_o}) |
|---|
| PILE |

Comme $1 \leqslant i_o < n$, on aura $1 \leqslant i_o + 1 \leqslant n$ et donc s = .OR.

Après exécution de la boucle, on revient au point A avec

SSt = .OR.C_{i_o+2}... .OR.C_nS' et

PILTYP = | type (C_1.OR. ... .OR.C_{i_o+1}) |
|---|
| PILE |

ce que nous devions démontrer.

La correction de ce programme s'établit en prenant i = n. On arrivera donc au point A avec

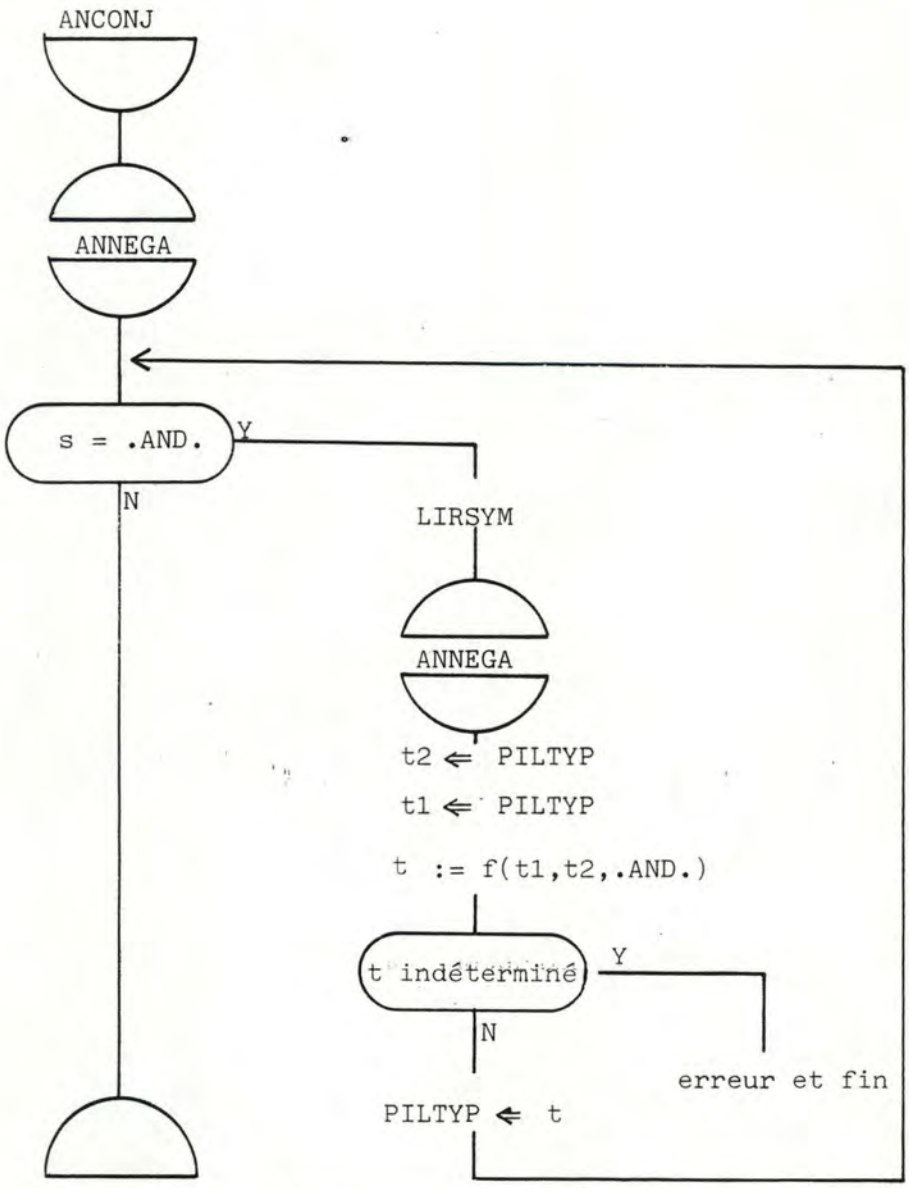
SSt = S' et

PILTYP = | type (C_1.OR. ... .OR.C_n) |
|---|
| PILE |

Comme s ≠ .OR., le programme se terminera.

3.3. : ANCONJ : analyse d'une conjonction

Les raisonnements sont semblables à ceux effectués pour ANLEXP. Nous obtenons l'algorithme suivant :



3.4. : ANNEGA : analyse d'une négation

3.4.1. : Supposons que P existe et est sémantiquement correcte dans C

si P = T avec T ∈ cat(<proposition atomique>), P sera sémantiquement correcte dans C car T le sera.

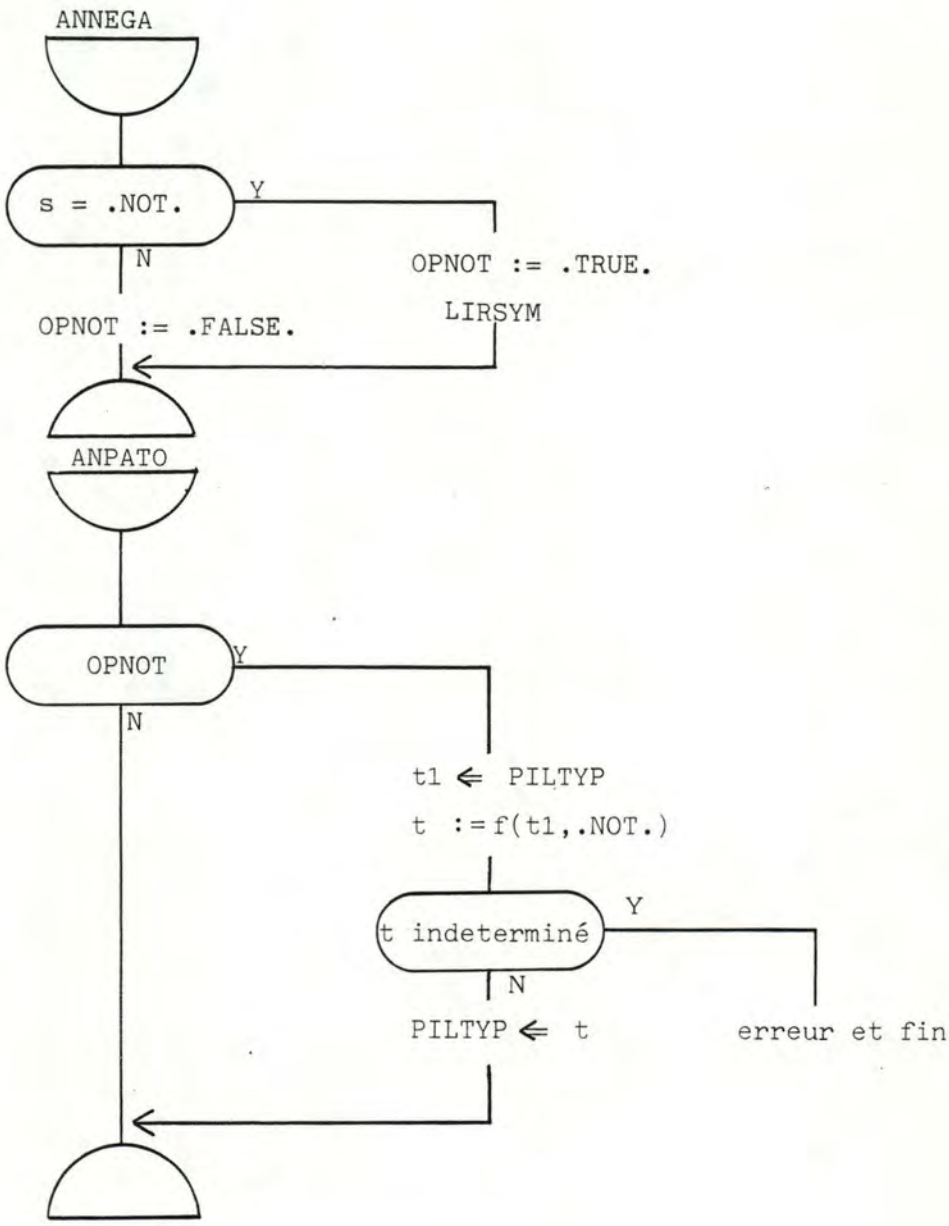
si P = .NOT.T, P sera sémantiquement correcte dans C car T le sera et le type résultat de T sera compatible avec l'opérateur monadique .NOT.

3.4.2. : Retrait des suppositions

Comme ANNEGA sera construit à partir de SXNEGA, une erreur syntaxique sera détectée par l'exécution de ANPATO. Pour une erreur sémantique, si celle-ci n'est pas détectée par ANPATO, c'est que le type résultat de T n'est pas compatible avec l'opérateur .NOT. rencontré.

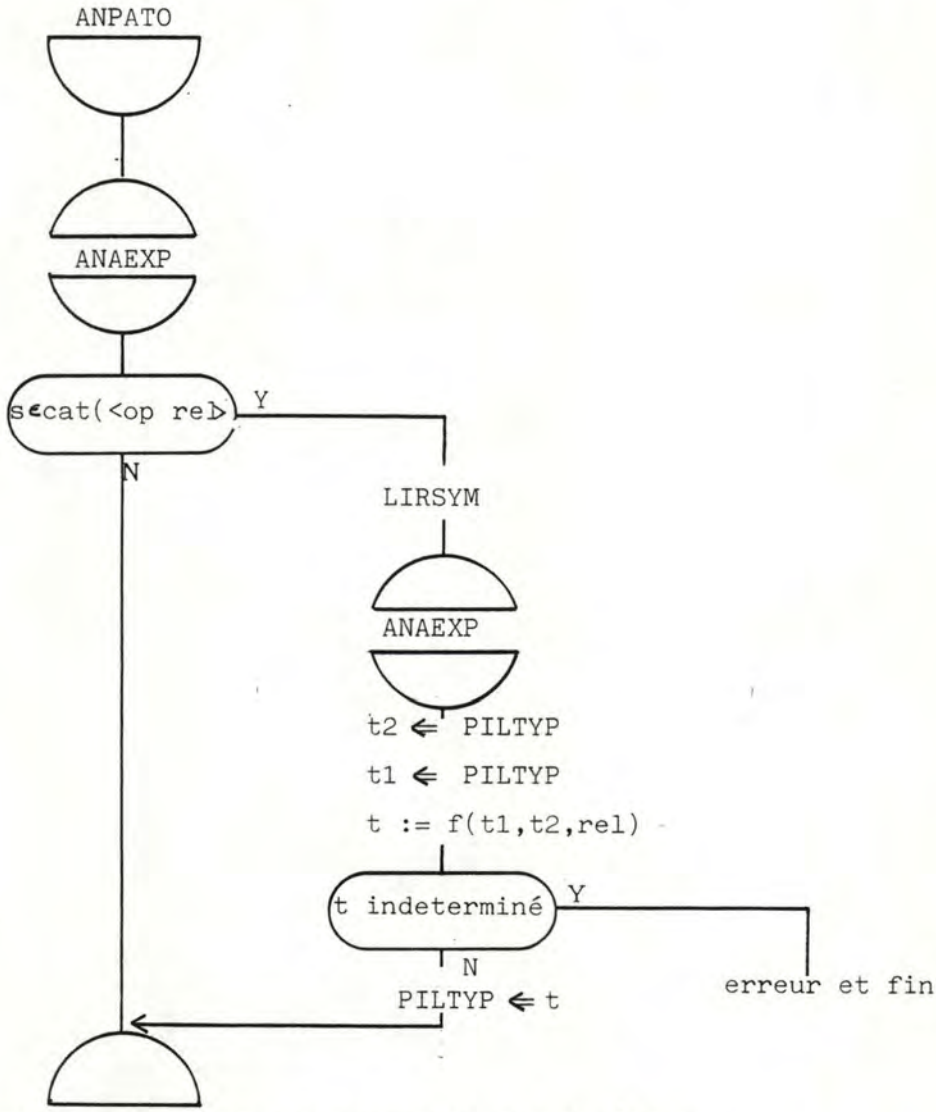
3.4.3. : Algorithme

Il faut donc mémoriser si l'on rencontre l'opérateur .NOT., et si tel est le cas, vérifier la compatibilité du type résultat de T avec cet opérateur. La variable OPNOT permet de mémoriser si l'on a rencontré - l'opérateur .NOT. et est locale à ANNEGA.



3.5. : ANPATO : analyse d'une proposition atomique

Modifions l'algorithme SXPATO pour qu'il vérifie, après le second appel à ANAEXP si les deux types au sommet de PILTYP sont compatibles et qu'il détermine le type résultat de P.



3.6. : ANAEXP : analyse d'une expression arithmétique

Nous pouvons effectuer les mêmes raisonnements que pour ANLEXP avec la différence suivante :

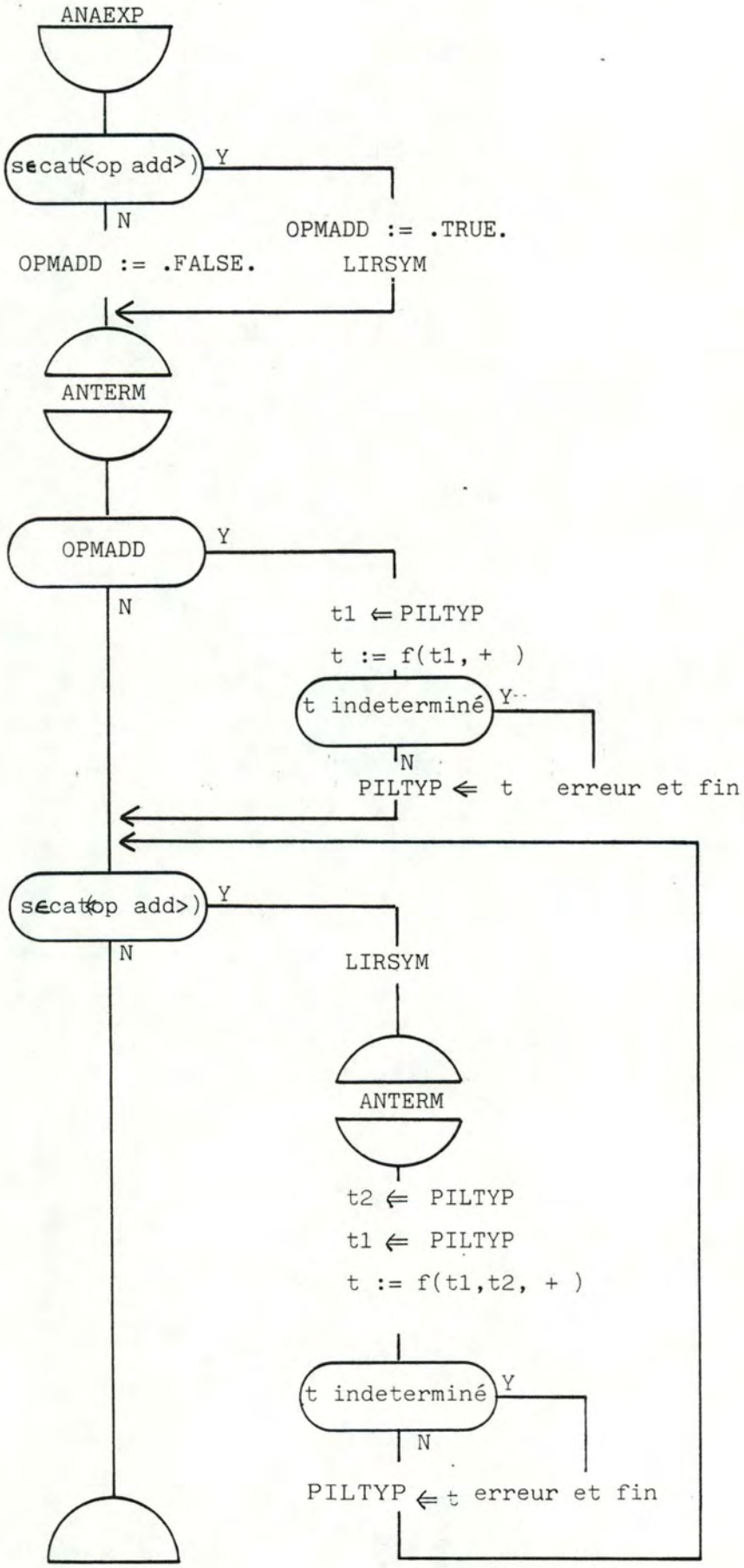
P est de la forme $\{\omega_1\}$ T $\{\omega_2 P'\}$ avec $\omega_1, \omega_2 \in$ cat(<op add>)

T ∈ cat(<terme>)

$\omega_2 P' \in$ cat(<expression arithmétique>)

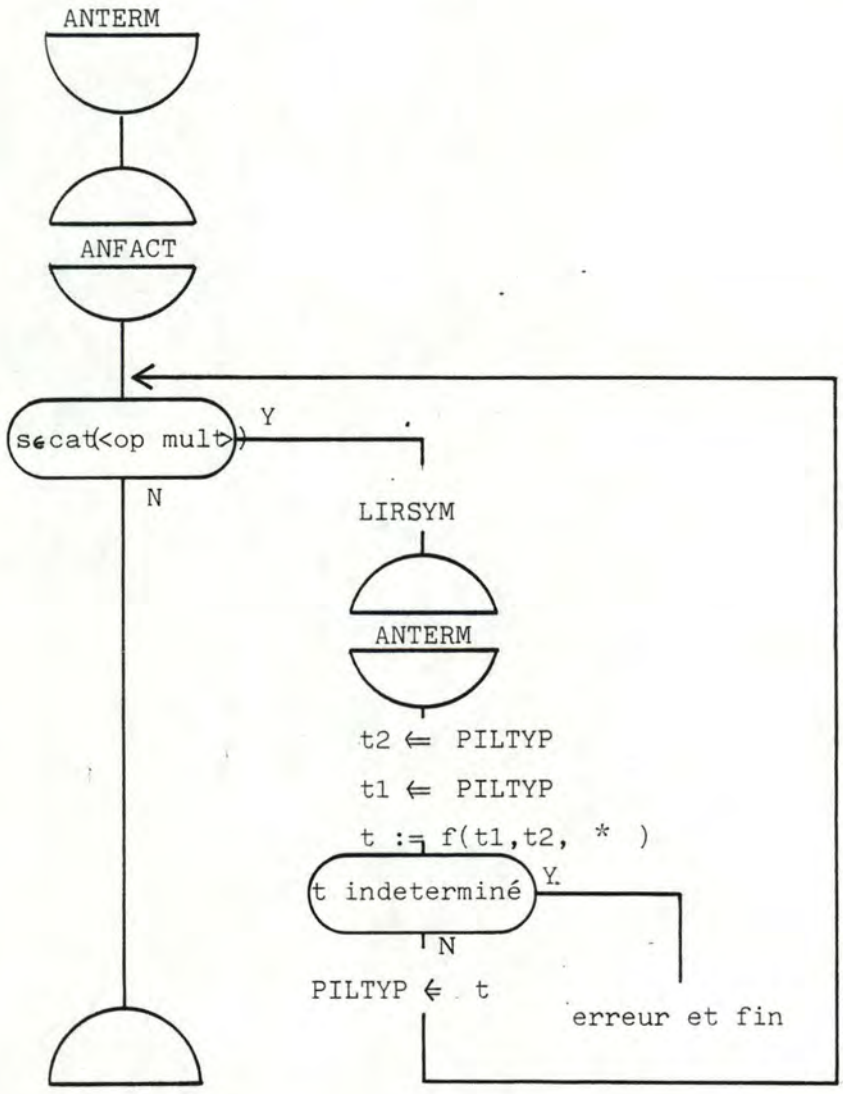
Si ω_1 est présent, il faut vérifier si le type résultat de T est compatible avec ω_1 et calculer le type résultat de $\omega_1 T$.

La variable locale OPMADD permet de mémoriser si ω_1 est présent.



3.7. : ANTERM : analyse d'un terme

Les raisonnements sont semblables à ceux effectués pour ANLEXP. Nous obtenons l'algorithme suivant :



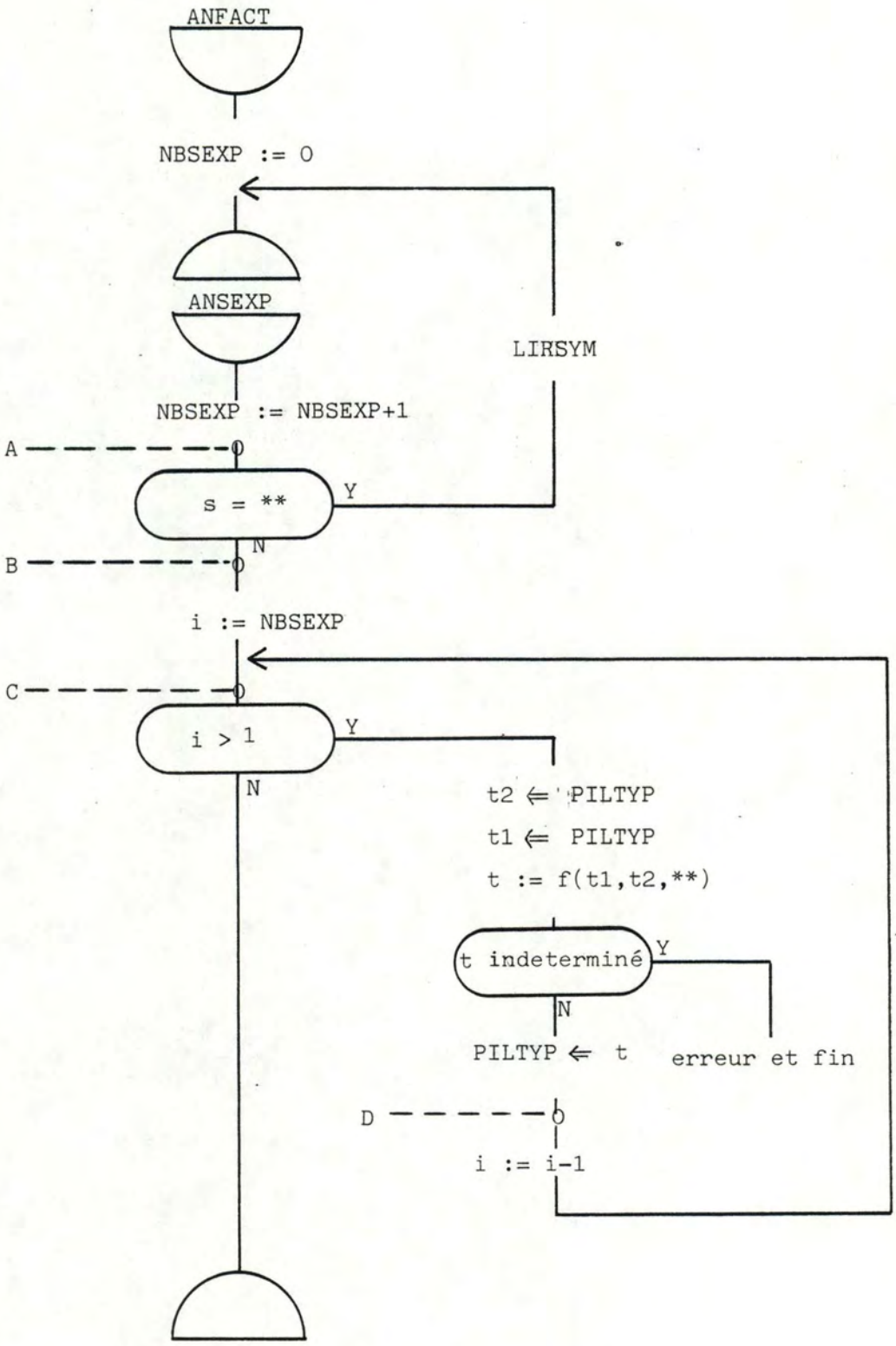
3.8. : ANFACT : analyse d'un facteur

3.8.1. : Algorithme

P sera de la forme $K_1 ** K_2 \ldots ** K_n$ avec $n \geq 1$

$K_i \in$ cat(<expression simple>)

Dans le cas d'un facteur, l'évaluation doit se faire de droite à gauche. Pour ce faire, nous utiliserons l'algorithme SXFACT où l'appel à SXSEXP est remplacé par un appel à ANSEXP pour empiler sur PILTYP les types résultats des K_i. Nous comptons le nombre d'expressions simples formant le facteur (variable locale NBSEXP). Dans une seconde partie, nous calculerons le type résultat de P tout en effectuant les vérifications sémantiques nécessaires.


ANFACT
NBSEXP := 0
ANSEXP
LIRSYM
NBSEXP := NBSEXP+1
A
s = **
Y
N
B
i := NBSEXP
C
i > 1
Y
N
t2 ⇐ PILTYP
t1 ⇐ PILTYP
t := f(t1,t2,**)
t indeterminé
Y
N
PILTYP ⇐ t
erreur et fin
D
i := i-1

3.8.2. : Démonstration

Nous supposons que P cat-max(< facteur >) dans S existe, est sémantiquement correct et est composé de n expressions simples sémantiquement correctes dans C.

1° partie

Montrons que pour tout $n \geq 1$, si initialement nous avons

SSt = K_1**K_2 ... **K_nS' et

PILTYP = | PILE |

Alors l'exécution arrivera au point B avec

SSt = S' et NBSEXP = n et

PILTYP =

type (K_n)
:
type (K_1)
PILE

Pour montrer cela, considérons le programme amputé de sa première instruction et prouvons que si initialement nous avons en plus NBSEXP = nb_o, alors l'exécution arrivera au point B comme ci-dessus avec en plus NBSEXP = $nb_o + n$

* vrai pour n = 1

Au point A, nous obtenons

SSt = S' et NBSEXP = $nb_o + 1$ et

PILTYP =

type (K_1)
PILE

Comme s ≠ **, l'exécution arrivera au point B

* si vrai pour n_o, alors vrai pour $n_o + 1$ ($n_o \geq 1$)

Au point A, nous obtenons

SSt = **K_2** ... K_nS' avec $n = n_o + 1 > 1$ et

NBSEXP = $nb_o + 1$ et

PILTYP =

type (K_1)
PILE

Comme s = **, après exécution de LIRSYM, nous nous retrouvons dans les conditions initiales du programme avec n_o expressions simples et NBSEXP = $nb_o + 1$.

Par hypothèse d'induction, le programme arrivera au point B avec

SSt = S' et NBSEXP = $(nb_o + 1) + n_o$ et

PILTYP =

type (K_{n_o+1})
⋮
type (K_2)
type (K_1)
PILE

La correction de cette première partie s'établit en considérant la première instruction qui établit NBSEXP = 0

2° partie

Considérons le programme à partir du point C. Montrons que pour tout $i_o \geq 1$,

Si $i = i_o$ et

PILTYP =

type (K_{i_o})
⋮
type (K_1)
PILE

Alors le programme se termine et établit

PILTYP =

type (K_1** ... **K_{i_o})
PILE

* vrai pour $i_o = 1$

En effet, dans ce cas, le programme se termine immédiatement et établit

PILTYP =

type (K_1)
PILE

* si vrai pour i_o, alors vrai pour i_o+1 ($i_o \geq 1$)

Comme $i = i_o + 1 > 1$, le programme arrivera au point D avec

$i = i_o + 1$ et

PILTYP =

type (K_{i_o}**K_{i_o+1})
type (K_{i_o-1})
⋮
type (K_1)
PILE

Après l'exécution de l'instruction i:=i-1, nous pouvons appliquer l'hypothèse d'induction. Le programme se terminera donc avec

PILTYP =

type (K_1** ... **(K_{i_1}**K_{i_o+1}))
PILE

La correction du programme entier s'établit en remarquant qu'au premier passage au point C, nous avons i = n et que la seconde partie du programme laisse SSt intacte. Signalons encore que type (K_1** ... **K_{i_o} **K_{i_o+1}) = type (K_1** ... **(K_{i_o} **K_{i_o+1})) le prgramme se terminera donc avec

SSt = S' et

PILTYP = | type (K_1** ... **K_n) |
| PILE |

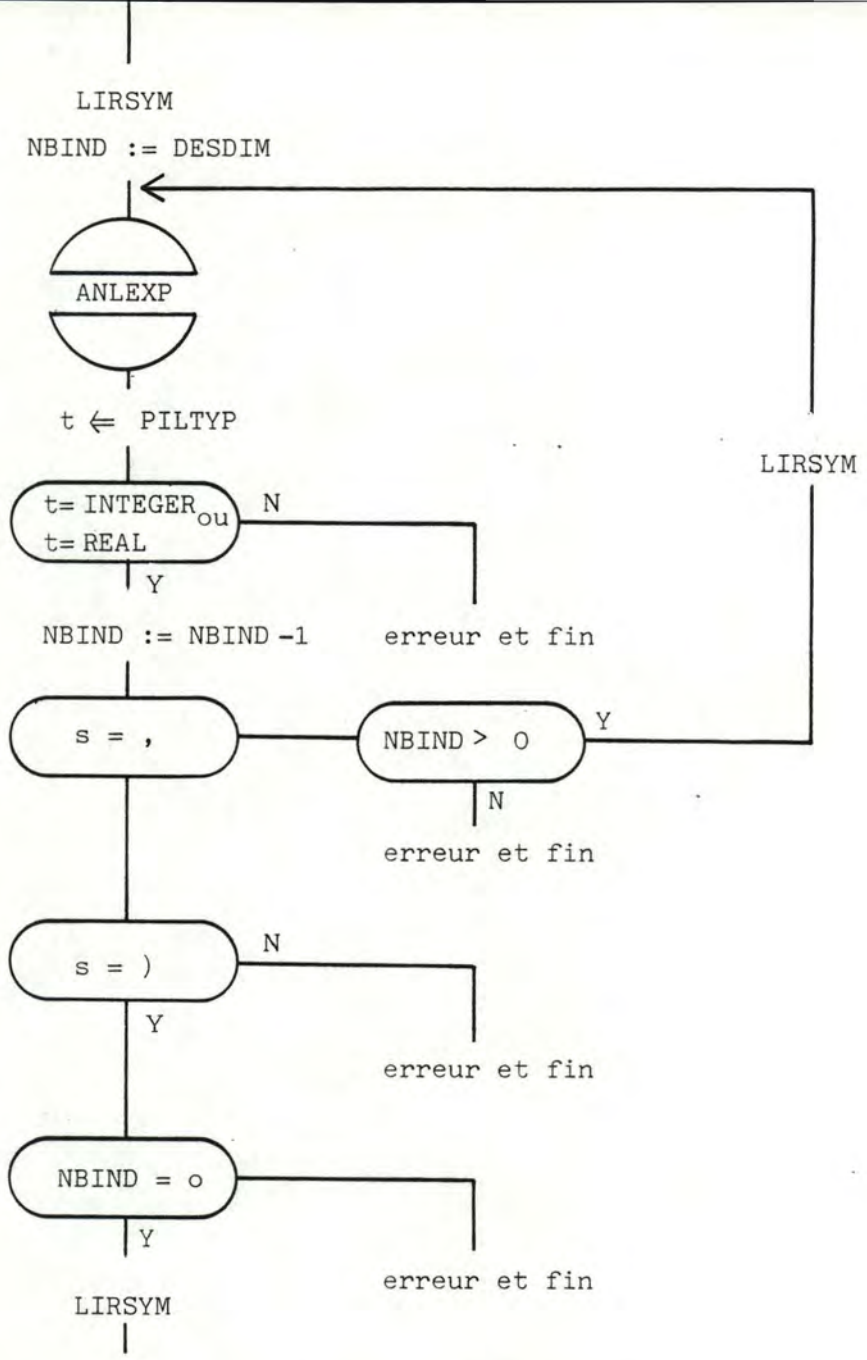
3.9. : ANSEXP : Analyse d'une expression simple

3.9.1. : Problèmes auxiliaires

3.9.1.1. : Vérification des indices d'un tableau

Soit SSt = (S
cntc = C
S'il existe un préfixe P cat-max(<liste d'indices>) dans S, suivi du symbole) et composé de n expressions sémantiquement correctes dans C de type résultat INTEGER ou REAL
Si (DESBAT,DESIDE,DESTYP,DESDIM) contient une représentation interne d'un descripteur de tableau dans C avec DESDIM = n
Alors l'exécution de cette partie de programme
- traitera le préfixe (P)

Sinon l'exécution de cette partie de programme
- établira ERREXP = .TRUE.
- remplira le message d'erreur appartenant à l'environnement
- s'arrêtera immédiatement.



La variable NBIND est locale à cette partie de programme. Le test NBIND > 0 dans la boucle permet d'écourter l'analyse dans le cas où le nombre d'indices est supérieur à DESDIM.

3.9.1.2. : Vérification des paramètres d'une fonction

Soit SSt = (S
cntc = C
S'il existe un préfixe P,cat-max (<liste par. fonction>) dans S, suivi du symbole) et composé de n expressions.
Si chacune de ces expressions est sémantiquement correcte dans C; ou si elle se réduit à un identificateur appartenant à l'un des ensembles Nsdec(C), Ntab(C) ou Next(C).

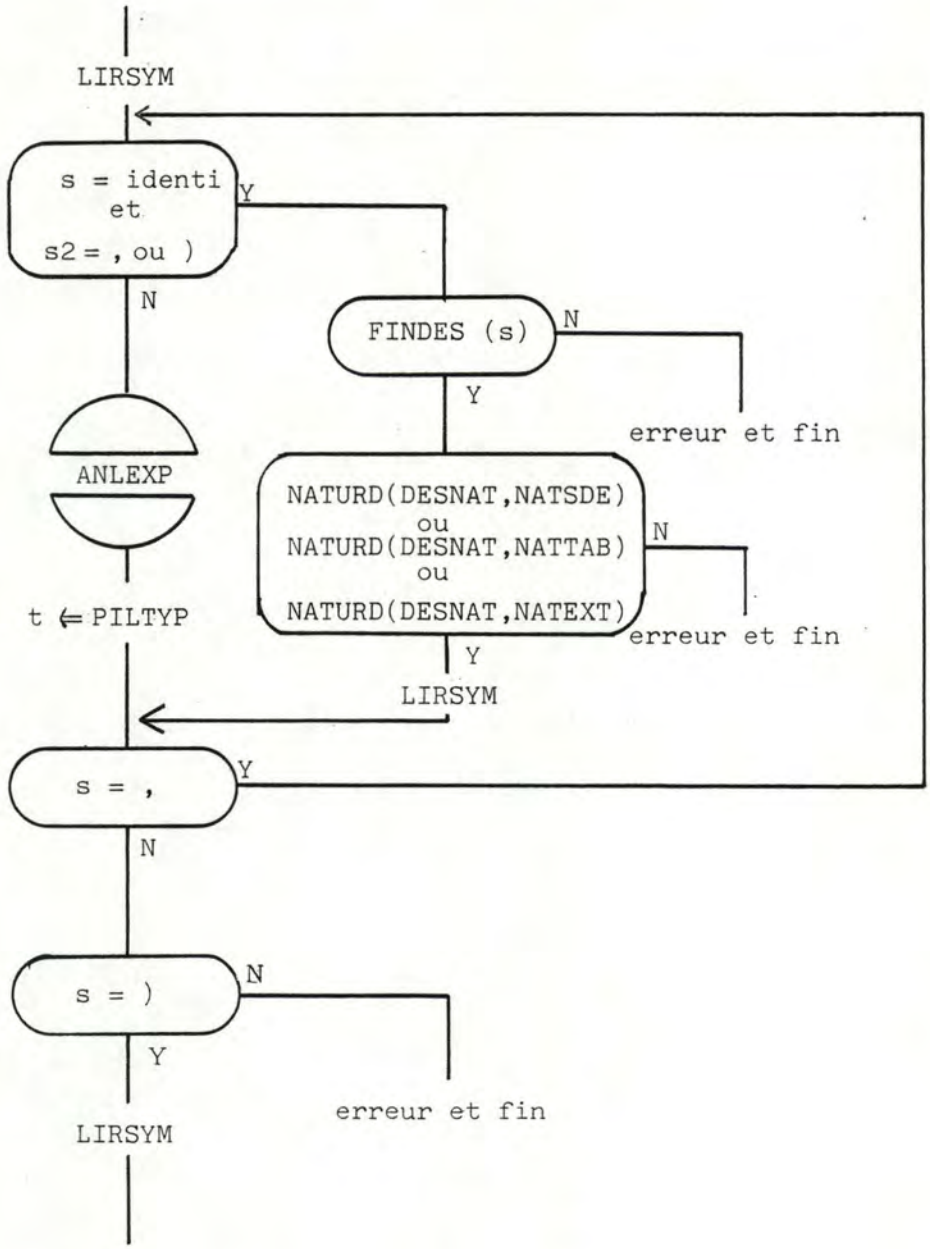
Alors l'exécution de cette partie de programme

- traitera le préfixe (P)

Sinon l'exécution de cette partie de programme

- établira ERREXP = .TRUE.
- remplira le message d'erreur appartenant à l'environnement
- s'arrêtera immédiatement.

Comme précédement, s représente le premier symbole de base de SSt. De plus, ici, s2 représentera le deuxième symbole de base de SSt. (si SSt est vide, s2 aura la même valeur que s). Les fonctions FINDES et DESNAT ont été spécifiées ch I points 3.5.2. et 3.5.3.



3.9.2. : Algorithme

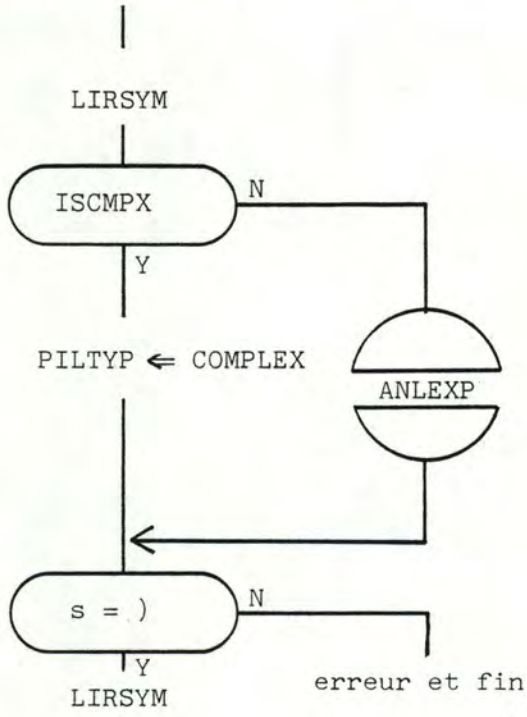
Nous allons enrichir l'algorithme SXSEXP pour qu'il effectue en plus les vérifications sémantiques.

* si s est une constante

Dans ce cas, il suffira de faire PILTYP <= type(s)
LIRSYM

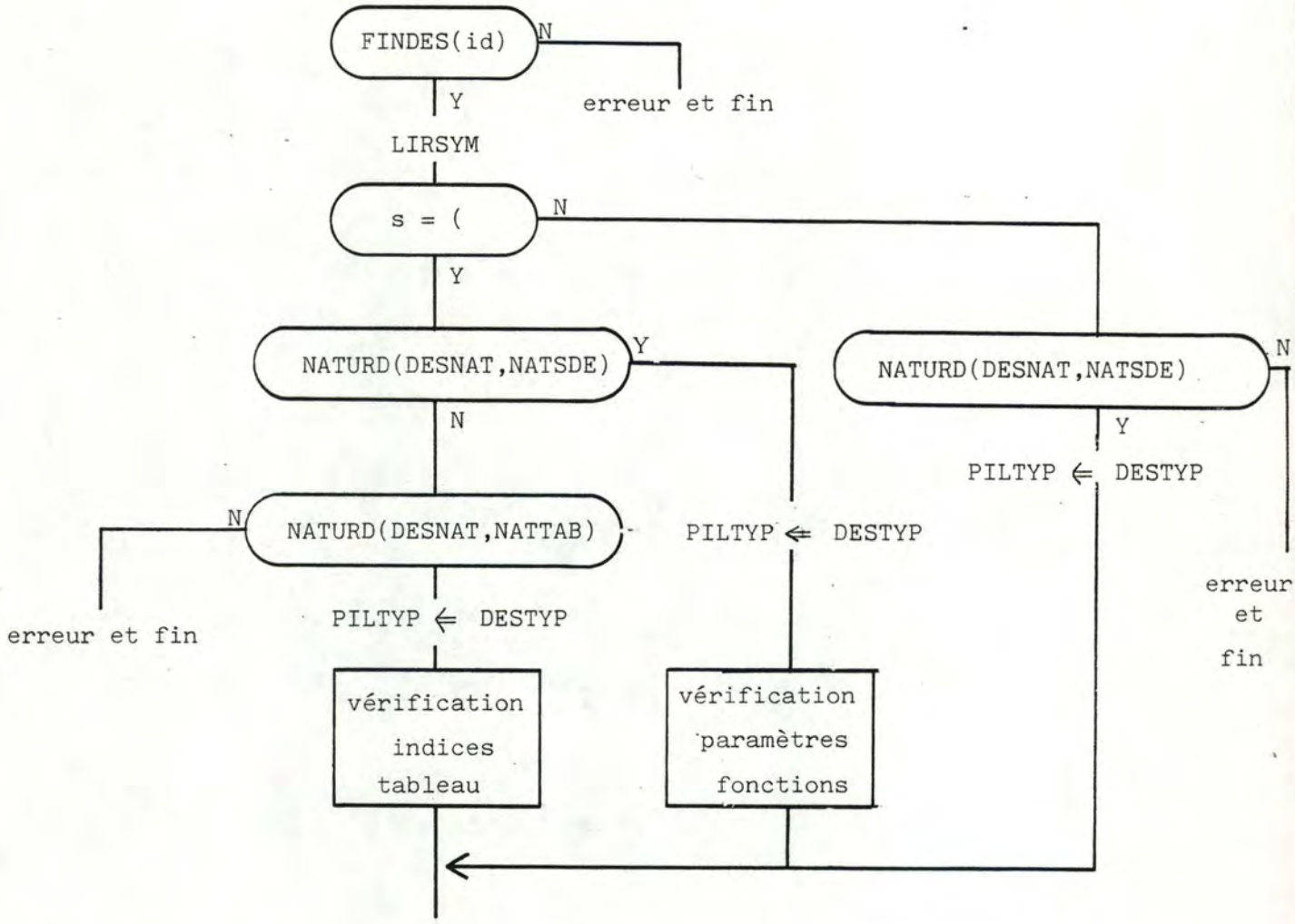
* si s = (

Dans ce cas, il faudra mettre le type COMPLEX sur la pile PILTYP si P est une constante complexe.



* si s = id avec id ∈ cat(<identificateur>)

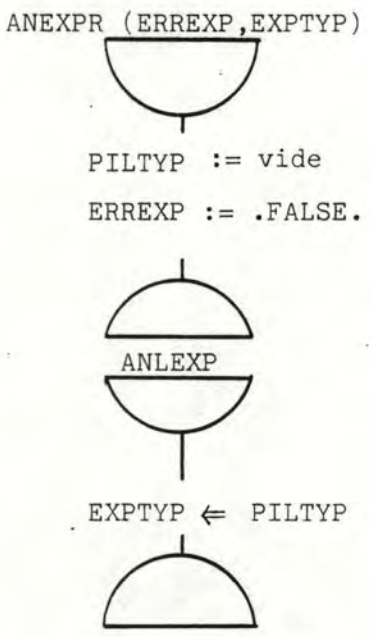
- si le symbole suivant n'est pas (, il faut qu'il existe un descripteur d de nom id et de nature sdec dans C.
  Le type résultat sera alors type(d).
- si le symbole suivant est (, il faut qu'il existe un descripteur d de nom id et de nature sdec ou tab dans C.
  Si, selon le cas, les indices du tableau ou les paramètres de la fonction sont syntaxiquement et sémantiquement corrects dans C, alors le type résultat sera type (d)



4. : PROGRAMME ANEXPR : ANALYSE D'UNE EXPRESSION

4.1. : Algorithme

Une expression peut-être considérée comme une expression logique car tout préfixe P,cat-max(<expression>) dans S,sera aussi préfixe cat-max(<expression logique>) dans S et réciproquement. D'après la spécification de ANLEXP, nous obtenons l'algorithme suivant de ANEXPR :



4.2. : Correction du programme

D'après les raisonnements effectués jusqu'ici, nous pouvons dire que ANEXPR est correct si ANLEXP l'est

ANLEXP est correct si ANCONJ l'est

:

ANFACT est correct si ANSEXP l'est

ANSEXP est correct si ANLEXP l'est.

Nous en déduirons que

ANLEXP est correct si ANSEXP l'est

et que ANSEXP est correct si ANLEXP l'est.

Voici donc un cercle vicieux qui ne nous permet pas d'affirmer que ANEXPR est correct.

Mais remarquons que ANSEXP est correct pour un préfixe P cat-max (<expression simple>) dans S si ANLEXP est correct pour un préfixe E cat-max(<expression logique>) inclus strictement dans P. De plus, si P se réduit à 1 symbole de base, ANSEXP sera correct car il n'y a pas d'appel à ANLEXP.

Pour tous préfixes P cat-max(<expression logique>) dans S, on arrivera tôt ou tard à une exécution de ANSEXP qui ne fera pas appel à ANLEXP et qui sera correcte. En remontant la chaîne des appels récursifs, on établit alors que ANSEXP est correct.

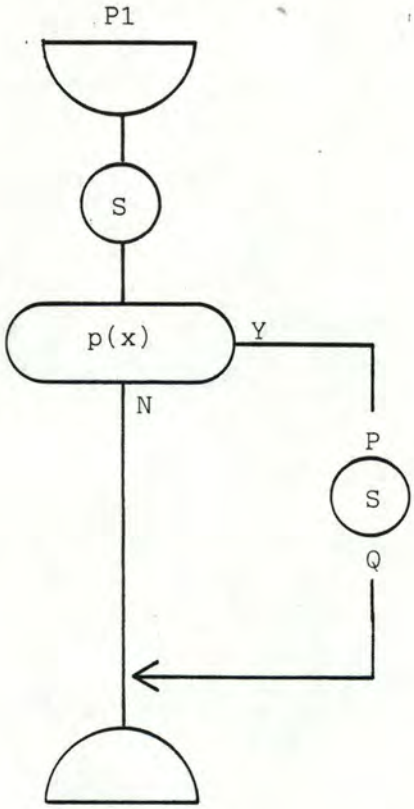
5. : ELIMINATION DE LA RECURSIVITE

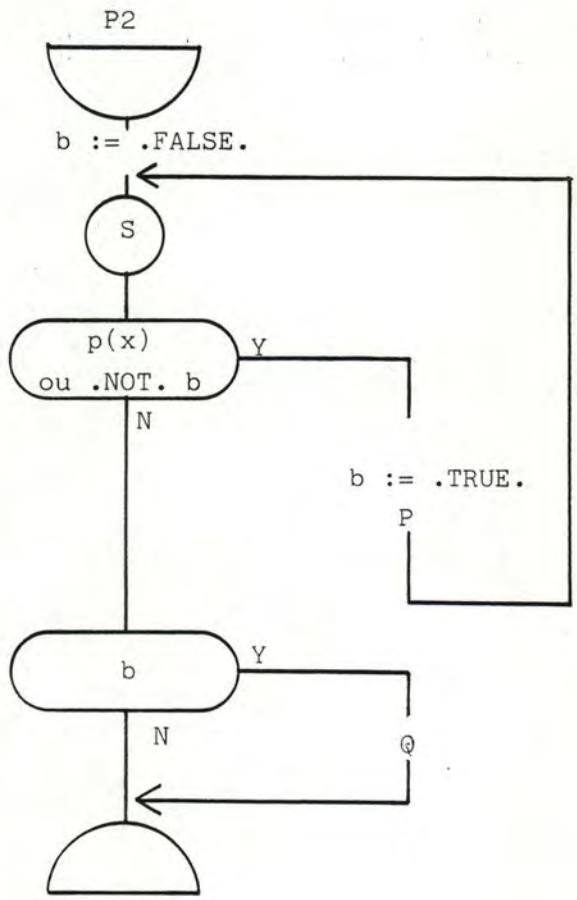
5.1. : Intégration des différents programmes

Nous allons intégrer les programmes ANLEXP, ANCONJ, ANNEGA, ANPATO, ANAEXP, ANTERM, ANFACT et ANSEXP en un seul programme récursif équivalent à ANLEXP. Nous allons substituer, à chaque appel de programme, le texte de ce programme. Comme il est non acceptable de copier plusieurs fois le même texte, nous allons transformer ces programmes pour qu'ils ne fassent qu'un appel au programme de niveau inférieur. Il faudra également que toutes les variables locales aient des noms distincts.

5.1.1. : Transformation de ANPATO

Prouvons tout d'abord l'équivalence des deux algorithmes suivants où P, Q, S, sont des séquences d'instructions et p(x) un test sur l'ensemble x des variables et b une variable logique n'appartenant pas à x.

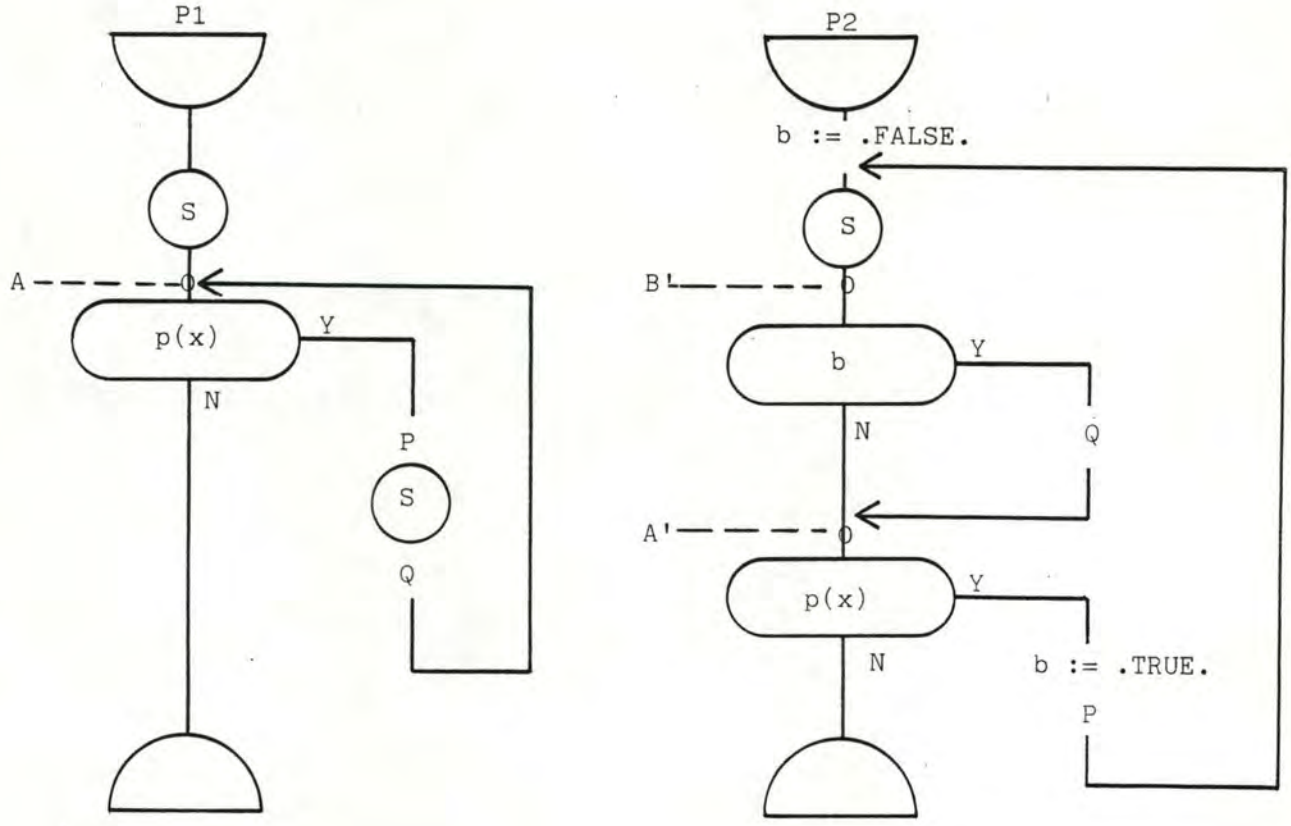




L'équivalence de ces deux algorithmes peut se montrer par une exécution symbolique de ceux-ci. Les deux programmes exécuteront une première fois S et p(x) aura la même valeur dans le premier programme que dans le second. Ainsi si p(x) est faux, les deux programmes se termineront dans le même état final. Si p(x) est vrai, le premier programme va exécuter P, S puis Q; le second exécutera également P, S et comme b sera .TRUE., il exécutera Q avant de se terminer dans le même état que le premier.
Nous utiliserons cette équivalence pour transformer l'algorithme ANPATO où S représente l'appel à la procédure de niveau inférieur. Nous appellerons OPREL la variable logique b qui sera locale.

5.1.2. : Transformation de ANLEXP, ANCONJ, ANTERM

Prouvons tout d'abord l'équivalence des deux algorithmes suivants :



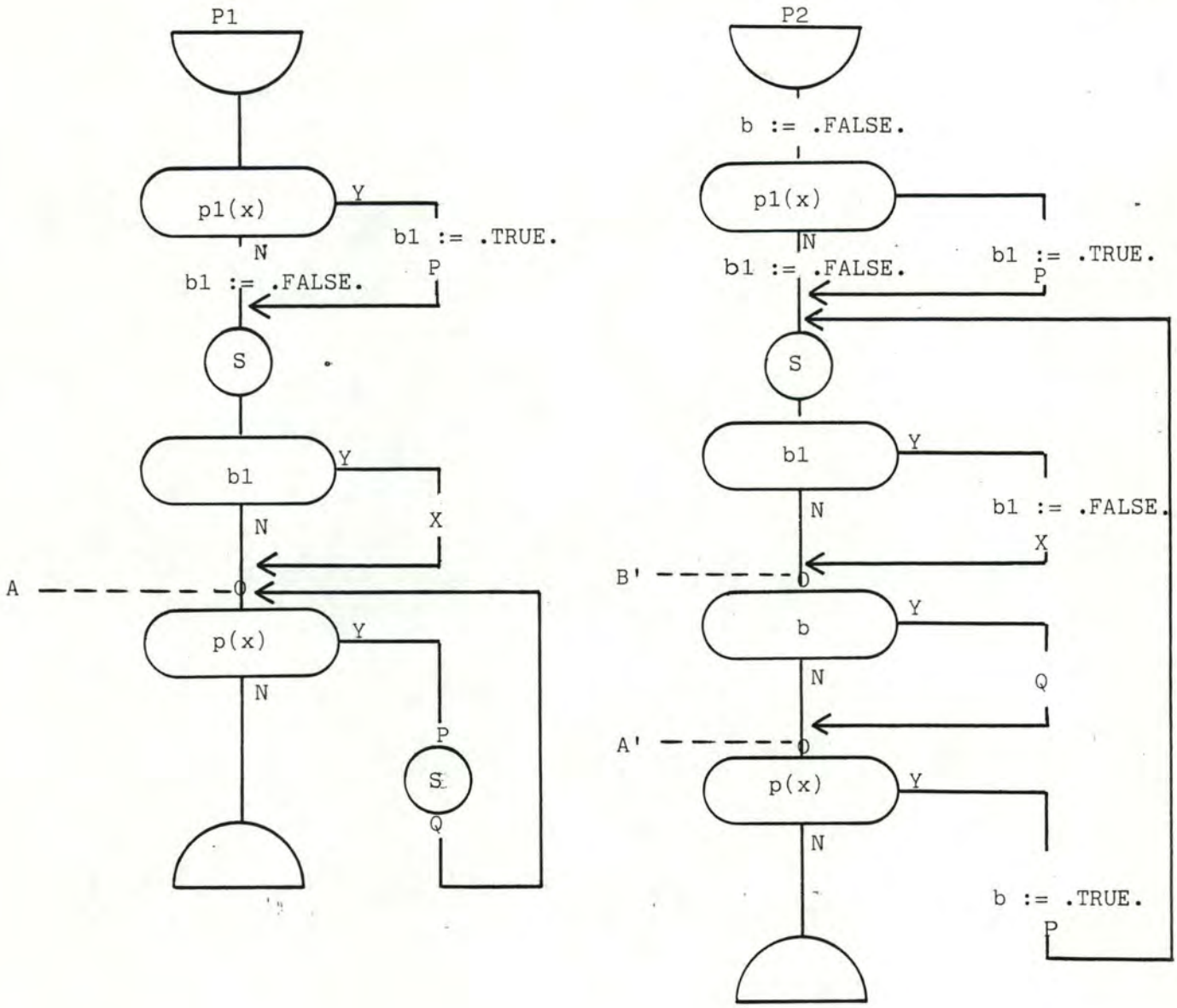
Montrons que à tout passage au point A avec $x = x_o$, correspond un passage au point A' avec également $x = x_o$.
En effet, raisonnons par récurrence sur le nombre n de passages au point A.(n est supposé fini)

- vrai pour le premier passage.
  si le premier programme arrive au point A avec $x = x_o$, le second arrivera au point B' avec $x = x_o$ et b = .FALSE.; et donc arrivera au point A' avec encore $x = x_o$.
- si vrai pour le $i^{ème}$ passage, alors vrai pour le $i+1^{ème}$ $(i<n)$
  Au $i^{ème}$ passage au point A du premier programme, nous avons $x = x_o$ et $p(x_o)$ vrai car $i<n$. Au $i^{ème}$ passage du point A' du second programme, nous avons également $x = x_o$ et $p(x_o)$ vrai. Le premier programme va revenir au point A après avoir exécuté P, S et Q. Nous aurons alors $x = x'_o$. Le second programme va d'abord revenir au point B' après avoir exécuté P et S; nous aurons notamment b = .TRUE. et on exécutera donc Q avant de revenir au point A avec $x = x'_o$.

Avec un même raisonnement que celui-ci, nous pourrions établir que à tout passage au point A' avec $x = x_o$, correspond un passage au point A avec $x = x_o$.
L'équivalence de P_1 et P_2 s'établit alors en prenant i = n. Les deux programmes se terminent donc avec la même valeur de x.
Cette équivalence nous permet de transformer les algorithmes ANLEXP, ANCONJ et ANTERM où S représente l'appel à la procédure de niveau inférieur. Nous appellerons OPOR, OPAND et OPMULT les 3 variables logiques supplémentaires de ces 3 algorithmes; ces variables seront locales.

5.1.3. : Transformation de ANAEXP

Montrons tout d'abord l'équivalence des deux algorithmes suivants :



Montrons que à tout passage au point A avec $x = x_o$, correspond un passage au point A' avec $x = x_o$ et b1 = .FALSE.
En effet, raisonnons par récurrence sur le nombre n de passages au point A (n est supposé fini).

- vrai pour le premier passage.

  Soit $x = x_i$ initialement Si $p1(x_i)$ est faux, alors le premier programme arrivera au point A avec $x = x_o$ après avoir exécuté S. Dans ce cas, le second programme arrivera au point B' avec $x = x_o$, b1 = .FALSE. et b = .FALSE.; on arrivera donc au point A' avec $x = x_o$ et b1 = .FALSE.

Si p1(x_i) est vrai, le premier programme arrivera au point A avec $x = x'_o$ après avoir exécuté P, S et X.
Dans ce cas, le second programme arrivera au point B' avec $x = x'_o$, b1 = .FALSE. et b = .FALSE.; on arrivera donc au point A' avec $x = x'_o$ et b1 = .FALSE.

- si vrai pour le $i^{ème}$ passage, alors vrai pour le $i+1^{ème}$ ($i<n$)
Au $i^{ème}$ passage au point A du programme P1 nous avons $x = x_o$ et $p(x_o)$ vrai car $i<n$. Au $i^{ème}$ passage au point A' du programme P2 , nous avons $x = x_o$, b1 = .FALSE. et $p(x_o)$ vrai
Le premier programme va revenir au point A avec $x = x''_o$ après avoir exécuté P, S et Q. Le second programme va revenir au point B' après avoir exécuté P et S car b1 = .FALSE.. Comme b = .TRUE., on exécutera Q avant de revenir au point A' avec $x = x''_o$ et b1 = .FALSE.

Avec un même raisonnement que celui-ci, nous pourrions montrer que à tout passage au point A' avec $x = x_o$ et b_1 = .FALSE., correspond un passage au point A avec $x = x_o$.

L'équivalence de P1 et P2 s'établit alors en prenant i = n. Les deux programmes se terminent donc avec la même valeur de x.

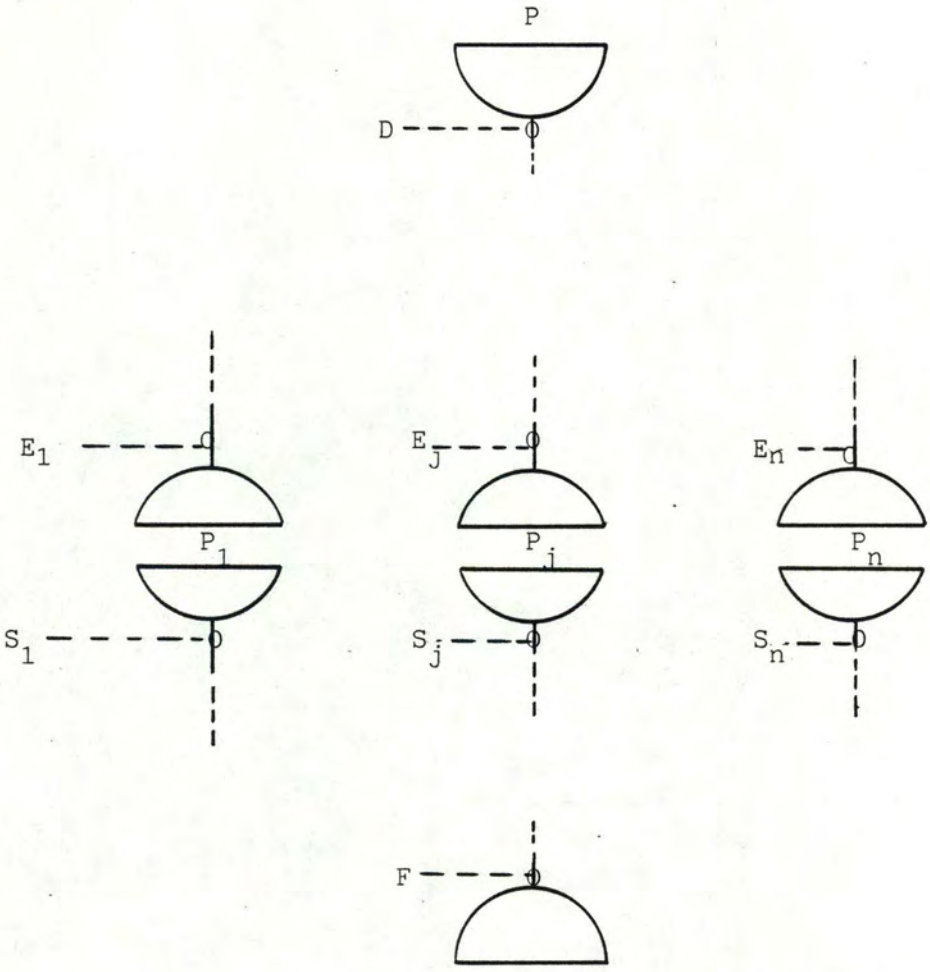
Cette équivalence nous permet de transformer l'algorithme ANAEXP où S représente l'appel à la procédure de niveau inférieur et b_1 la variable OPMADD. La nouvelle variable b sera appelée OPDADD et sera locale.

5.2. : Elimination des appels récursifs

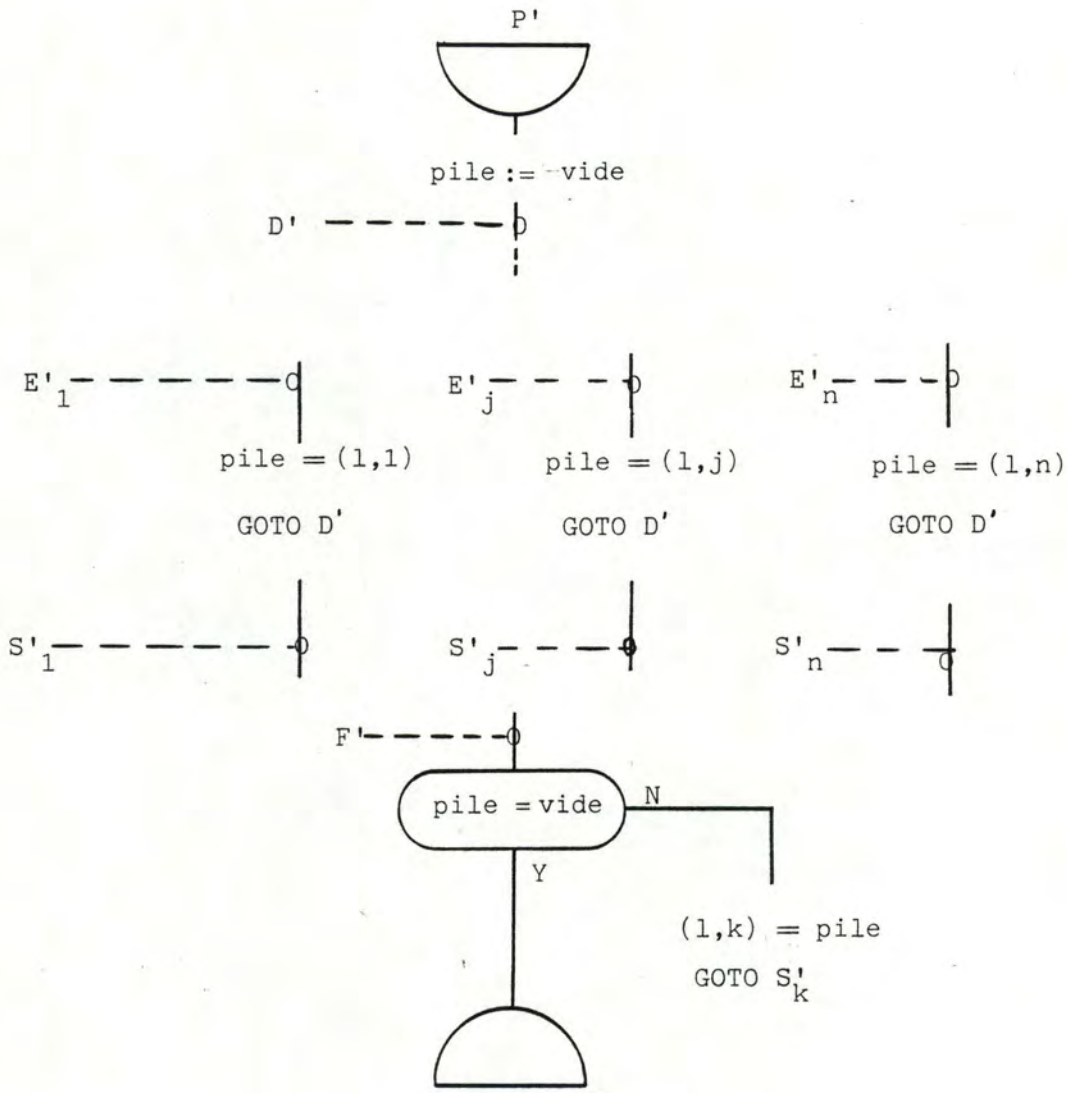
Nous avons donc un programme équivalent à ANLEXP contenant 3 appels récursifs. Nous allons tout d'abord étudier comment, d'une façon générale, on peut retirer les appels récursifs d'un programme et appliquer cette méthode pour obtenir l'algorithme final de ANEXPR.

5.2.1. : Principe du retrait des appels récursifs.

Soit P un programme contenant n appels récursifs (n>0)
g les variables globales de P
l les variables locales de P
P_i un appel récursif de P



Soit P' le programme non récursif avec pile, g, l et k comme variables globales. Pile est une variable de type pile dont chaque élément est (l,i) où i est un entier et l, l'ensemble des variables locales de P. k est une variable entière.



Si nous supposons que P et P' sont semblables pour tout ce qui n'a pas été représenté, nous allons montrer que

si initialement nous avons $g = g_o$ pour P et P'

et que l'exécution de P se termine avec $g = g_f$

alors l'exécution de P' se termine avec $g = g_f$.

Montrons tout d'abord que

si initialement $g = g_o$ pour P

$g = g_o$ et pile = $\boxed{p_o}$ pour P' au point D'

et que l'exécution de P se termine avec $g = g_f$

alors il existe un passage au point F' de P' avec $g = g_f$

et pile = $\boxed{p_o}$

Nous allons raisonner par récurrence sur le maximum de la profondeur des appels récursifs de P. Soit $m(g_o)$ ce nombre.

1. vrai pour $m(g_o) = 0$

Comme P et P' sont semblables en dehors des appels récursifs, le programme P arrivera au point F avec $g = g_f$ tout comme le programme P' au point F' avec $g = g_f$ et pile = $\boxed{p_o}$

2. si vrai pour $m(g_o) = i$, alors vrai pour $m(g_o) = i+1$ $(i \geqslant 0)$

Soit E_j, le point avant le premier appel récursif de P.

nous avons à ce point $g = g_1$

$l = l_1$

L'exécution de P' arrivera donc à E'_j avec $g = g_1$

$l = l_1$

pile = $\boxed{p_o}$

Après exécution des 2 instructions suivantes, on se retrouve au point D' avec $g = g_1$

pile = $\boxed{\begin{array}{|c|c|}\hline l_1 & j \\ \hline \multicolumn{2}{|c|}{p_o} \\ \hline \end{array}}$

Comme $m(g_1) < m(g_o)$ et par hypothèse de récurrence, si l'exécution de P après l'appel récursif P_j arrive au point S_j

avec $g = g_{1f}$

$l = l_1$

alors l'exécution de P' arrivera au point F'

avec $g = g_{1f}$

et pile = $\boxed{\begin{array}{|c|c|}\hline l_1 & j \\ \hline \multicolumn{2}{|c|}{p_o} \\ \hline \end{array}}$

Comme pile $\neq$ vide, on arrivera au point S'_j

avec $g = g_{1f}$

$l = l_1$

pile = $\boxed{p_o}$

S'il n'y a qu'un appel récursif de P au niveau 1, l'exécution de P arrivera au point F avec $g = g_f$ et comme les deux programmes sont semblables en dehors des appels récursifs, l'exécution de P' arrivera au point F'

avec $g = g_f$

pile = $\boxed{p_o}$

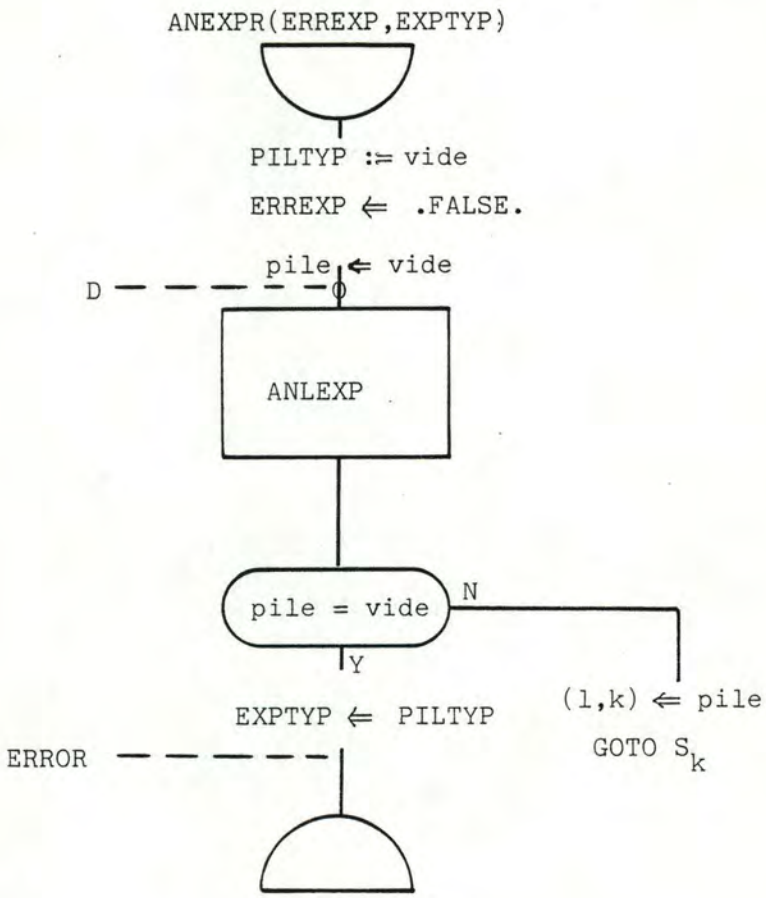
Dans le cas d'un nombre quelconque d'appels récursifs de niveau 1, il suffit d'effectuer un raisonnement par récurrence sur le nombre d'appel de niveau 1.

La propriété générale se déduit dans le cas particulier du premier passage au point D'. Nous avons alors initiallement $g = g_o$ pour P et P' et pile = vide pour P'. Si l'exécution de P se termine avec $g = g_f$, alors il existera un passage au point F' avec $g = g_f$ et pile = vide; le programme P' se terminera donc avec $g = g_f$.

C.Q.F.D

5.2.2. : Application à ANEXPR

L'ensemble des variables locales à sauvegarder avant branchement est : OPOR, OPAND, OPREL, OPDADD, OPMADD, OPMULT, NBSEXP et NBIND. Le programme ANEXPR devient



dans lequel ANLEXP est l'intégration des différents programmes où chaque appel récursif a été remplacé par

```
|
pile ⇐ (l , j )
GOTO D.
|
```

Lorsqu'une erreur est détectée, après avoir mis ERREXP à .TRUE. et rempli le message d'erreur, il faut arrêter immédiatement l'analyse. Pour cela, il suffira d'effectuer un branchement vers ERROR.

3EME PARTIE

CRITIQUE

Chapitre I : S P E C I F I C A T I O N

1. : MODULARISATION D'UN SYSTEME

Nous allons développer ici quelques principes exposés par Parnas [PAR,79] qui facilitent la structuration d'un système et comparer ceux-ci avec les principes exposés dans la première partie et appliqués dans la seconde.

Pour comprendre une chose complexe, nous devons la décomposer systématiquement et successivement en parties plus simples et en comprenant comment ces parties s'ajustent. Nous devons donc avoir différents niveaux de compréhension et chacun de ces niveaux correspond à une abstraction des détails d'implémentation des niveaux inférieurs.

1.1. : Structure hiérarchique UTILISE

1.1.1. : Relation UTILISE

Considérons un système décomposé en un ensemble de parties que nous appellerons composants, possédant chacun une spécification. Nous dirons de deux composants A et B que A UTILISE B si le fonctionnement correct de A dépend de la disponibilité d'une implémentation correcte de B.

1.1.2. : Hiérarchie UTILISE

Si l'on restreint la relation UTILISE de telle sorte qu'il n'y ait pas de circuit, nous obtenons alors une structure hiérarchique en niveaux correspondant aux différents niveaux d'abstraction. Le problème consiste donc à définir cette structure, ou encore comment déterminer que A UTILISE B. D'après Parnas, 4 conditions doivent être satisfaites :

1) A est beaucoup plus simple du fait d'utiliser B.
2) B n'est pas plus compliqué du fait de ne pouvoir utiliser A.
3) Il existe un sous-système utile pouvant contenir B et pas A.
4) Il n'existe pas de sous-système utile pouvant contenir A et pas B.

Une telle structure permet de factoriser le problème et donc de mieux le maîtriser tant au niveau de la spécification que de ceux de la conception, de la validation et de la maintenance. Les différents composants doivent "cacher un maximum d'informations"; cacher un "secret". Ce secret peut être, par exemple, un mode de réalisation ou de représentation. De plus, le degré d'interdépendance entre les composants doit être le plus faible possible; tandis que le degré d'interdépendance entre les traitements au sein d'un même composant doit être le plus élevé possible. C'est à ces différents critères que l'on jugera une structure hiérarchique en niveau appelée aussi modularisation.

1.2. : Critique

Les quatre conditions énoncées ci-dessus nous apparaissent comme extrêmement vagues et difficiles à utiliser pour effectuer une modularisation. Tout d'abord lorsqu'on dit que A est plus simple du fait d'utiliser B, qu'est-ce qui est plus simple; sa spécification ou son implémentation ? A notre avis, il ne peut s'agir que de l'implémentation. Cela veut donc dire qu'au stade de la modularisation, on a une idée suffisante de l'algorithme pour être capable de déterminer si celui-ci sera plus simple du fait d'utiliser l'autre. La phase de modularisation n'est donc pas indépendante de la phase de construction des programmes. De plus, si un composant peut utiliser un autre ou peut ne pas l'utiliser, comment déterminer s'il doit l'utiliser ?

Dans la définition de la relation UTILISE, que signifie que le fonctionnement correct de A dépend d'une implémentation correcte de B ? Ainsi, par exemple, si A UTILISE B et que B UTILISE C, peut-on dire que A UTILISE C ? Si tel était le cas, on perdrait alors un peu de pouvoir d'abstraction apporté par cette notion.

Enfin, qu'est-ce que l'interdépendance entre composants, l'interdépendance entre les traitements au sein d'un composant et surtout comment la mesurer ? S'il existait une métrique pour effectuer cette mesure, nous pensons qu'elle ne pourrait être qu'arbitraire. Si l'on admet la subjectivité dans cette mesure, celle-ci ne pourrait se faire qu'après la construction des différents programmes et non au moment où l'on effectue la modularisation.

Dans la démarche exposée ci-dessus, il y a un souci de rendre la démarche systématique, de fournir des critères effectifs et presque mesurables. Elle nous semble, en définitive, pseudo systématique et les critères proposés n'ont rien d'effectif .

1.3. : Structure hiérarchique UTILISE et structure induite par la notion de primitive

Nous pouvons faire quelques rapprochements entre notre démarche de spécification (1° partie chI.2) et celle exposée ci-dessus. Au niveau de la relation déterminant la structure générale des programmes, nous avons retenu la relation "est considéré comme primitive". Il nous semble que cette relation est semblable à la relation UTILISE et fournit donc le procédé d'abstraction nécessaire à la résolution d'un problème complexe en présentant l'avantage d'être définissable de façon plus précise. Dire que B "est considéré comme primitive" pour A, cela veut dire qu'on écrit le programme A dans une extension du langage, contenant B comme primitive supplémentaire.

Nous avons souligné que la spécification d'un problème nécessite l'identification de concepts adéquats pour en construire une solution. Ces concepts nous permettront de déduire une structure qui pourra être hiérarchique. Nous avons donc déplacé la difficulté dans la découverte de concepts adéquats, sachant que leur adéquation ne pourra être établie qu'a posteriori.

Cette façon de travailler, ajoutée au fait que nous avons décidé de raisonner sur les objets eux-mêmes plutôt que sur leurs représentations fera que chaque composant cachera un secret. On peut donc espérer que, si l'on change un composant sans modifier sa spécification, on ne devra pas modifier les autres composants.

Nous n'avons pas donné de condition pour qu'un composant soit une primitive d'un autre; nous pensons que cela n'a pas de sens absolu "en soi" et que c'est inutile. Au niveau des spécifications, de telles conditions n'auraient pas de sens car il est impossible, à ce niveau, de dire si un composant est une primitive d'un autre. Au niveau des algorithmes, une telle relation existera entre deux composants si cela simplifie les raisonnements, facilite la compréhension et la construction du programme.

2. : CARACTERISTIQUES D'UN "BON" LOGICIEL

Il est généralement reconnu qu'un bon logiciel se caractérise par les qualités ci-dessous. Nous allons briévement les développer et voir si notre démarche permet de les obtenir.

2.1. : Fiabilité

La notion de fiabilité correspond à la notion de correction; c'est à dire la validité par rapport aux spécifications. Ceci est notre première préoccupation lors de la résolution d'un problème, tant au niveau des spécifications techniques qu'au niveau de la construction des programmes.

2.2. : Facilité de maintenance et extensibilité

Nous pensons que la réalisation de ces objectifs dépend étroitement de la fiabilité, encore faut-il avoir une bonne documentation. Par bonne documentation, nous entendons tout d'abord une définition précise des concepts utilisés, de même que la théorie développée à leur sujet. Cette documentation devra ensuite justifier l'emploi de ces concepts et leur lien avec le problème. Elle devra encore contenir l'ensemble des spécifications des programmes, de même qu'une démonstration de ceux-ci et une justification des décisions prises lors de leur construction. La qualité d'une documentation sera jugée à sa simplicité, lisibilité et compréhensibilité.

2.3. : Portabilité

Le concept de portabilité correspond au fait de pouvoir exécuter les programmes sur des configurations différentes, avec le moins de modifications possibles. Comme nous l'avons dit précédemment, le choix de travailler sur les objets eux-mêmes permet de localiser les décisions de représentation de ces objets et donc, lorsqu'on modifiera un composant sans modifier sa spécification, on ne devra pas modifier les autres composants.

Le concept de portabilité nécessite la localisation des caractéristiques du langage utilisé susceptibles d'être interprétés différemment d'une configuration à l'autre (par exemple, les entrées/sorties.).

Nous pensons que cette notion exige une connaissance et une expérience permettant de déterminer ces caractéristiques. Cette connaissance permettra alors de les localiser dans des composants particuliers.

3. : CHOIX DU LANGAGE DE SPECIFICATION

3.1. : La vérité par définition

Comme le fait remarquer H. Leroy, la notion de programme correct risque de déboucher sur un cercle vicieux : celui de la vérité par définition [LER,80].

> "J'écris un nombre, et je le déclare correct par définition. Une définition étant, par nature même, irréfutable, il n'y a ni argument, ni expérience, qui pourrait me contraindre à admettre que mon "résultat" n'est pas correct. Il s'ensuit qu'il a aussi l'inconvénient de ne pouvoir servir à rien. Il faut d'ailleurs noter que ce n'est pas le nombre en tant que tel qui ne peut servir à rien (il serait absurde de dire que le nombre 2 ne peut servir à rien), mais l'affirmation que ce nombre est le résultat correct, par définition. Le résultat de quoi ? De la définition. Le cercle vicieux est tellement flagrant que personne n'oserait défendre une pareille conception du traitement de l'information.
>
> Compliquons donc un peu les choses. Au lieu de choisir mon nombre arbitrairement, et par la voie la plus directe, j'écris un programme long et compliqué. Pour fixer les idées, je suppose que ce programme n'accepte pas de données, donc qu'il n'est pas destiné à être exécuté plus d'une fois. Je m'assure que le programme terminera normalement son exécution en produisant un nombre comme résultat. Cela fait, je déclare le programme correct par définition. Je le fais exécuter par une machine, et je recueille le résultat, forcément correct, puisque le programme l'était. ... Le résultat que j'obtiens n'est encore correct que par définition. Il n'est donc ni plus, ni moins stérile que son prédécesseur. Qu'il ne soit pas une conséquence triviale ou immédiate de la définition ne lui confère rien d'autre que la supériorité, plutôt douteuse, d'avoir coûté plus cher. ...
>
> Dans la pratique, la vérité par définition n'est pas appliquée avec cette simplicité; elle est beaucoup moins apparente.

> Considérons par exemple un "analyste" qui produit un algorithme et l'appelle spécification. De deux choses l'une. Ou bien son algorithme est assez précis et complet pour déterminer exactement son exécution, donc ses résultats. Alors, pour ce qui nous occupe, il ne diffère en rien d'un programme. Ce programme, étant appelé spécification, est à lui-même sa propre norme de vérité. Ou bien l'algorithme est vague et incomplet, et il en est donc de même de la "spécification", qui ne sera complétée qu'en écrivant un programme au sens strict du terme. Il ne restera à la fin, comme spécification précise et complète de ce que doit faire le programme, que le programme lui-même qui, étant identique à lui-même, sera correct par définition. ... Bien sûr, l'analyste n'a pas produit son algorithme arbitrairement. Il a dû partir d'un vague énoncé. L'énoncé étant trop vague, il l'a remplacé par un algorithme plus précis qui, baptisé spécification, remplit désormais la double fonction de poser le problème et de le résoudre."

Une spécification devrait avoir une précision nécessaire et suffisante pour qu'on puisse s'assurer qu'un programme qui utilise cette primitive est correct, et que, lors de sa programmation, on n'ait pas besoin de se référer à autre chose qu'à sa spécification. Mais pour ne pas tomber dans le travers de la vérité par définition, il faudrait savoir s'il existe un moyen pour définir exactement une primitive, sans la programmer. Comment une spécification peut-elle être suffisamment précise sans se réduire à un algorithme plus ou moins déguisé ?

3.2. : Langage formel de spécification

Une réponse possible consiste à dire qu'il n'y a qu'à faire un langage formel de spécification permettant ainsi de spécifier sans ambiguïté une primitive. Un tel langage serait communicable et nettement supérieur au langage naturel car non équivoque. Ainsi, lorsque Parnas donne les objectifs d'une spécification, il dit entre autres qu'une spécification doit être suffisamment formelle pour qu'elle puisse être testée pour sa consistance, complétude et autres propriétés souhaitables pour une spécification. Pour atteindre cet objectif, il est nécessaire d'éviter les spécifications en langage naturel[PAR,72a]. Nous allons essayer de montrer qu'un langage formel de spécification est illusoire,

contraignant et ne permet pas de sortir du cercle vicieux de la vérité par définition.

3.2.1. : Un formalisme doit être défini

Une première remarque à faire est, qu'avant d'utiliser un langage formel, il faut le définir et sa définition ne pourra être énoncée sans recourir d'une façon ou d'une autre au langage naturel. Si l'on accepte que le langage naturel est capable de définir sans ambiguïté un langage formel, on admet déjà que le langage naturel peut être précis dans certains cas. Pour éviter le recours au langage naturel, on essaye alors de définir le langage formel à l'aide d'un langage formel de "définition du langage formel". Si le langage de définition est le même que celui que l'on veut définir, c'est un cercle vicieux; si c'est un langage formel différent, ce dernier doit lui aussi être défini et cela ne fait que reporter le problème.

3.2.2. : Faiblesse du pouvoir d'expression

Si l'on admet que la qualité d'un langage formel est la précision de ses énoncés, le prix à payer est une limitation excessive du pouvoir d'expression. Rien ne peut être dit comme on souhaiterait le dire. La variété des choses exprimables au moyen d'un système formel est ridiculement limitée en regard de tout ce que l'on peut imaginer et, d'autre part, ce qui peut être exprimé ne pourra l'être en général que très maladroitement et de façon incompréhensible. On pourra bien sûr trouver parfois des conventions permettant de représenter certains objets "indéfinissables" par d'autres qui le sont, mais ces conventions seront toujours extérieures au langage formel, inexprimables en ses termes et seront donc non formelles.

3.2.3. : Existera-t-il un "bon" langage formel de spécification ?

Ce qui rend un langage formel inutilisable, c'est sa pauvreté, son petit nombre de concepts. Deux solutions sont alors envisageables pour surmonter cette difficulté : soit fournir un formalisme contenant un beaucoup plus grand nombre de concepts, soit

fournir un formalisme "extensible".

Un formalisme contenant suffisamment de concepts pour être réellement pratique ne peut exister. Tout d'abord, sa définition serait insupportablement longue et comprendrait des risques énormes d'incohérence. La plus grande difficulté serait de délimiter l'ensemble des rapports possibles entre tous les concepts du formalisme. Ceci est une tâche énorme et impossible à effectuer sans arbitraire. Même si une telle définition était effectuée, aucun concept du formalisme obtenu ne correspondrait de manière adéquate aux concepts intuitifs visés. Cela signifie que les énoncés qu'on écrira dans ce formalisme ne pourront être compris directement, c'est à dire dans leur sens intuitif, mais devront être "décodés" au moyen des définitions alambiquées du langage.

Une autre solution consiste à définir un formalisme comportant peu de concepts primitifs, mais contenant des mécanismes d'extensions permettant, à tout moment, à partir des concepts et des notations déjà définies, d'en définir de nouveaux. Il faut remarquer que les mécanismes d'extension, faisant partie d'un formalisme, seront d'une nature très particulière et permettront uniquement de définir de nouveaux objets et de nouvelles notations toujours de la même façon. Or il existera beaucoup de cas où d'autres mécanismes auraient été préférables.

3.2.4. : Le cercle vicieux associé à l'utilisateur d'un langage formel de spécification

Si un langage formel de spécification n'est rien d'autre qu'un langage de programmation de plus, on tombe à nouveau dans le cercle vicieux de la vérité par définition. Mais pour masquer cette évidence, on dira que ce langage est non-algorithmique puisqu'il permet d'écrire les instructions dans n'importe quel ordre, ou encore qu'il n'est pas compilable. Mais l'essentiel n'est pas là, l'essentiel est que quelque soit la forme d'un tel langage, quelque savantes et complexes que soient les procédures, manuelles ou mécaniques, de passage de la spécification au programme ou de démonstration de conformité du programme à ses spécifications, on restera toujours enfermé dans le cercle vicieux de la vérité par définition. [LER,80]

Que la spécification en langage formel soit elle-même un algorithme ou pas, on restera toujours enfermé dans le cercle vicieux de la vérité par définition. Une telle spécification sera semblable à un programme, car l'effort consistant à écrire ou comprendre une propriété particulière énoncée dans un langage formel prédéfini, est de même nature et de même ordre de difficulté que celui que consiste à écrire ou à comprendre un programme. Dire qu'un programme est correct par rapport à une telle spécification équivaut à dire que le programme est correct par rapport à lui-même. Toute la difficulté consistera à montrer que la spécification formelle représente le problème à résoudre.

3.3. : Spécification en langage naturel

S'il est vrai que le langage naturel permet de construire des énoncés ambigus, il permet aussi de construire des énoncés précis. De plus, son pouvoir d'expression est pratiquement infini et coïncide avec la faculté que nous avons de percevoir et d'inventer; il peut d'ailleurs évoluer avec elle. S'en priver revient à renoncer à toutes possibilités de communication.

3.3.1. : Formalisme et notations

Les notations doivent être considérées comme des extensions du langage naturel, destinées à réaliser le meilleur compromis entre concision, précision et compréhensibilité, dans la rédaction d'énoncés. La principale différence entre un formalisme et une notation est qu'un formalisme est figé a priori, alors qu'un ensemble de notations ajouté au langage naturel est susceptible d'être étendu au delà de toute limite autre que celle de l'intelligence humaine, et de n'importe quelle façon, selon les besoins.

Il n'y a pas de système de notation universel et définitif. Sans doute existe-t-il des domaines où les concepts de base sont bien établis et très largement acceptés; dans ces domaines, il existe aussi des systèmes de notations utilisés et compris par une majorité de personnes. Mais ces systèmes ne constituent pas un formalisme; ils constituent seulement, avec l'appui du langage naturel, un langage spécialisé pour "parler" de certains objets .

Ses limites ne seront pas plus précises que celles du langage habituel; il sera seulement mieux adapté pour énoncer certaines classes de propriétés. Il ne suffit nullement d'utiliser certaines notations pour "faire du formalisme".

3.3.2. : Peut-on échapper au cercle vicieux de la vérité par définition ?

Si un langage formel ne permet pas d'échapper au cercle vicieux de la vérité par définition, le langage naturel le permet-il ?

La résolution d'un problème particulier relatif à un concept donné nécessite presque toujours la mise en lumière d'une certaine structure de ce concept; c'est à dire une façon de le voir ou de le décomposer qui rend possible les raisonnements nécessaires à la résolution du problème. La formulation d'une spécification sera alors facilitée par l'introduction de notations choisies en fonction des concepts du problème à résoudre, reflétant la structure de ces concepts.

Nous pensons que le langage naturel, augmenté de notations adéquates, offre plus de facilité pour définir de tel concepts et permettre ainsi de sortir du cercle vicieux de la vérité par définition. Tout d'abord parce que le langage naturel correspond le mieux à notre façon de penser; il y a donc plus de chances d'exprimer ce qu'on veut et pas autre chose. Mais il n'y a pas de critères absolus; dans la pratique, il faudra faire de son mieux pour qu'une spécification représente le plus fidèlement possible le problème à résoudre, et ce, sans contrainte arbitraire (langage formel). L'effort consistant à comprendre une telle spécification est d'un autre ordre que celui consistant à écrire un programme. Il n'existe pas de règles de vérification, mais dans chaque cas particulier, on pourra se convaincre avec rigueur que le programme correspond aux spécifications et que les spécifications correspondent au problème à résoudre.

Chapitre II : CONSTRUCTION ET VALIDATION

1. : GOTO LESS PROGRAMMING

1.1. : "Goto Statement Considered Harmful"

En 1968, Dijkstra écrivait que la qualité d'un programme est une fonction décroissante de la densité de "goto" dans les programmes qu'il produit [DIJ, 68a]. Essayons de voir pourquoi l'utilisation de branchements est désastreuse et comment faire pour les éviter.

L'observation pratique nous montre que les programmes contenant beaucoup d'étiquettes et de branchements sont généralement illisibles et souvent erronés. Remarquons aussi que l'activité d'un programmeur ne s'arrête pas lorsque son programme est écrit; le vrai sujet de son activité est de voir si le comportement dynamique de son programme est en accord avec ses spécifications. Remarquons encore que l'intelligence humaine est mieux adaptée à maîtriser des relations statiques et que notre pouvoir de visualiser l'évolution d'un processus dans le temps est assez peu développé.

Ainsi, il faudra rendre la plus évidente possible la correspondance entre le texte du programme et le déroulement dynamique de celui-ci. Or, l'utilisation de "goto" a pour effet de rendre beaucoup plus difficile la compréhension du lien entre le texte d'un programme et son exécution. De plus, la présence de branchements rend pratiquement impossible la formulation claire de la correspondance entre un point du texte et les différents passages de l'exécution par ce point.

On en conclut qu'un programme ne contenant que des instructions d' affectations, de conditions (IF ... THEN ... ELSE ... FI), d'appels de procédure et répétitives, (WHILE ... DO ... OD,REPEAT... UNTIL ...) permet de mieux suivre le déroulement de l'exécution parallèlement au texte du programme. Les branchements explicites ou "goto" doivent être donc entièrement exclus pour obtenir des programmes compréhensibles. L'abolition des branchements réduit l'effort intellectuel fourni par le programmeur pour comprendre ce que fait son programme et ce que la machine va faire lors de l'exécution de celui-ci. Un programme sans branchement sera appelé structuré et par opposition, un programme avec branchements sera appelé non structuré.

1.2. : Restructuration des programmes non structurés

Nous allons développer ici un ensemble de règles de transformation permettant, à partir d'un programme non structuré, d'obtenir un programme structuré équivalent. Nous considérons ici que deux programmes sont équivalents si pour les mêmes données, soit les deux programmes ne se terminent pas, soit ils terminent tous deux leur exécution avec les mêmes résultats. Cette transformation consiste à remplacer l'ensemble des branchements par des structures de contrôle adaptées. Nous résolvons ainsi le plus grand problème de la programmation: la suppression des "goto" !

1.2.1. : Définition des schémas de programmes avec branchements

Nous considérons un schéma de programmes comme un objet purement syntaxique et un programme comme un schéma de programmes muni d'une interprétation.

Les symboles élémentaires pour décrire les schémas de programmes avec branchements sont les suivants [LIV,78] :

- un ensemble X de variables
- un ensemble F de noms de fonctions
- un ensemble P de noms de prédicats
- un symbole := représentant l'affectation
- les symboles STOP et GOTO
- l'ensemble N des entiers positifs pour numéroter les instructions d'un schéma

Nous ne considérerons que 4 types d'instructions :

* l'instruction d'arrêt : STOP
* l'instruction d'affectation : $x_i := x_j$
  ou $x_i := f(y_1,\ldots,y_n)$ avec $x_i, y_j \in X$, $f \in F$
* l'instruction de test : $p(y_1,\ldots,y_m)$ h,k avec $p \in P$, $y_i \in X$, $h,k \in N$
* l'instruction de branchement : GOTO k avec $k \in N$

Un schéma de programme avec branchements est alors défini comme une suite finie Π de la forme

$(1,\varepsilon_1)\ (2,\varepsilon_2)\ \ldots\ \ldots\ (r,\varepsilon_r)$

où - chaque ε_i est une instruction d'un des types ci-dessus
- si ε_i est une instruction de test, on aura $1 \leq h_i, k_i \leq r$
- si ε_i est une instruction de branchement on aura $1 \leq k_i \leq r$
- toute instruction d'affectation est suivie d'une instruction.

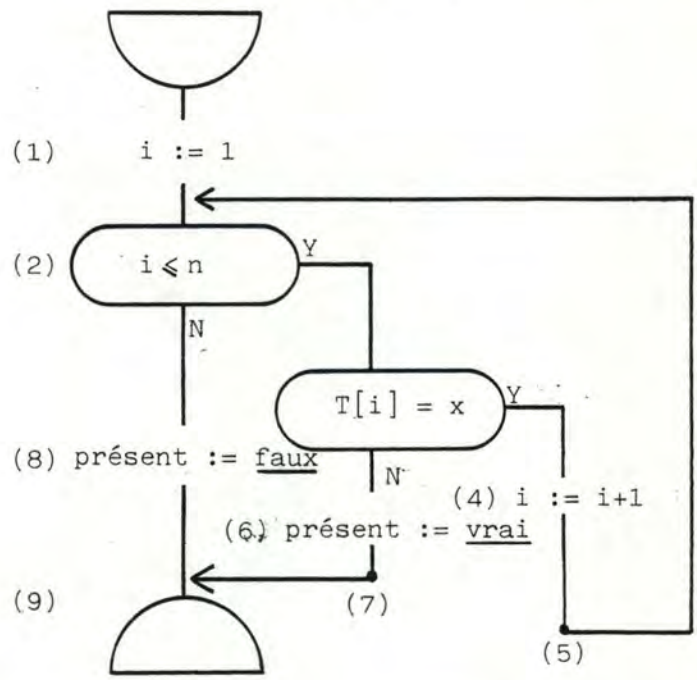
De façon évidente, les schémas de programmes avec branchements peuvent toujours être représentés par un organigramme et vice versa.

exemple

schéma de programme

```
1 i:=1
2 i≤n      3,8
3 T[i]=x   6,4
4 i:=i+1
5 GOTO     2
6 présent:=vrai
7 GOTO     9
8 présent:=faux
9 STOP
```

organigramme



Nous aurions dû, pour être complet, donner l'interprétation associée à un schéma. Nous supposerons que les règles d'interprétation d'un organigramme sont connues et que le passage d'un organigramme à un schéma avec branchements peut s'effectuer de manière intuitive. Signalons néanmoins pour interprèter un schéma avec branchements que l'exécution d'un test p(x) h,k est suivie de l'exécution de l'instruction h si le résultat du test est vrai et k s'il est égal à faux.

1.2.2. : Définition de schémas de programmes structurés

Nous considérons les 3 types d'instructions suivants :

* l'instruction d'affectation $x_i := x_j$

ou $x_i := f(y_1, \ldots, y_n)$ avec $x_i, y_j \in X$, $f \in F$

* l'instruction conditionnelle : IF $p(y_1,\ldots,y_n)$ THEN Π_1 ELSE Π_2 FI

avec $p \in P$

$y_j \in X$

Π_1, Π_2 deux suites d'instructions.

* l'instruction répétitive : WHILE $p(y_1,\ldots,y_n)$ DO Π OD

avec $p \in P$

$y_j \in X$

Π, une suite d'instructions.

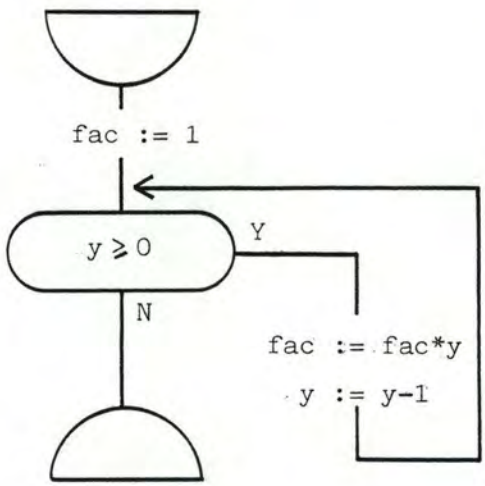
Un schéma de programme structuré est défini comme une suite finie d'instructions, de la forme $\varepsilon_1;\varepsilon_2;\ldots \quad \ldots;\varepsilon_r$
où chaque ε_i est une instruction d'un des 3 types ci-dessus.
A chaque schéma structuré correspond un organigramme, mais, cette fois, l'inverse n'est pas vrai.

exemple

schéma structure

```
fac:=1;
WHILE y≥0
DO
   fac:=fac*y;
   y:=y-1
OD
```

organigramme



1.2.3. : Règles de transformations

Notre but sera de transformer un organigramme quelconque en un schéma structuré.

étapes de transformation :

1. Traduire l'organigramme initial en un schéma avec branchement Π
2. Transformer Π en un schéma structuré Π'.

L'étape 1. étant supposée évidente, nous allons uniquement donner les règles de transformation de l'étape 2..

Définissons tout d'abord l'instruction

CASE x OF 1 : S_1
⋮
n : S_n

comme étant une abréviation de l'instruction structurée suivante

IF x = 1 THEN S_1
ELSE IF
⋮
ELSE IF x = n THEN S_n
FI
...
FI
...
FI

Soit P, un programme non structuré auquel correspond le schéma de programme avec branchements Π de la forme $(1,\varepsilon_1)$... ...(n,ε_n)

Soit X, l'ensemble des variables figurant dans Π

choisissons une variable $i \notin \Pi$.

Définissons le schéma structuré Π' avec X' = X U {i}

F' = F U {+,O,1}

P' = P' U {≠}

de la forme suivante :

```
i:=1
WHILE i ≠ O
DO CASE i OF 1:S1;
             ⋮
             j:Sj;
             ⋮
             n:Sn
OD
```

avec S_j, une suite d'instructions structurées d'une des formes :

- si ε_j est une instruction d'arrêt STOP,
  alors S_j sera : i:=O
- si ε_j est une instruction d'affectation $x_1:=f(y_1,...,y_n)$
  alors S_j sera : $x_1:=f(y_1,...,y_n)$;
  i:=i+1
- si ε_j est une instruction de test $p(y_1,...,y_m)h,k$
  alors S_j sera : IF $p(y_1,...,y_m)$ THEN i:=h
  ELSE i:=k FI
- si ε_j est une instruction de branchement GOTO k
  alors S_j sera : i:=k

Il faudrait maintenant prouver que les schémas Π et Π' sont équivalents; mais nous ne ferons pas ici cette démonstration [BOH,66].

1.3. : Utilité de la restructuration

La "goto less programming", quoique déjà appliquée dès 1960, pris son essor en 1968 lorsque Dijkstra publia son article "Goto Statement Considered Harmful" [DIJ,68a].
Ce n'est qu'en 1974, à notre avis, que fut clairement délimitée l'importance réelle de ce problème lorsque Knuth publia l'article "Structured Programming with goto Statements" [KNU,74].

Nous venons de montrer que tout programme sous forme d'organigramme peut être transformé systématiquement en un autre programme calculant les mêmes résultats que le premier et utilisant seulement les 3 structures de base : l'affectation, la condition et l'itération. Mais l'introduction d'une variable auxiliaire simule un compteur d'instructions du programme; compteur qui détermine quelle partie de l'organigramme va être exécutée. On élimine ainsi tous les branchements ; mais la structure du programme est perdue. Le programme résultat ne peut être alors beaucoup plus clair que le programme original. La traduction d'un programme quelconque plus ou moins mécaniquement en un programme tout aussi quelconque n'est donc pas recommandée.

L'élimination d'un branchement est un gaspillage d'énergie lorsqu'il figure dans un programme parfaitement compréhensible; de plus, le programme obtenu sans branchement ne sera pas conceptuellement plus simple. D'ailleurs, lorsqu'un branchement est utilisé judicieusement avec des commentaires établissant les raisonnements effectués, celui-ci peut être sémantiquement équivalent à une instruction "WHILE".

La notion de restructuration de programme apparait donc comme illusoire. La question n'est donc pas de retirer les branchements d'un programme pour le rendre "structuré", mais de savoir si l'utilisation de branchements est nuisible pour la compréhension d'un raisonnement accompagnant ce programme et s'il faut s'en interdire l'utilisation lors de sa construction.

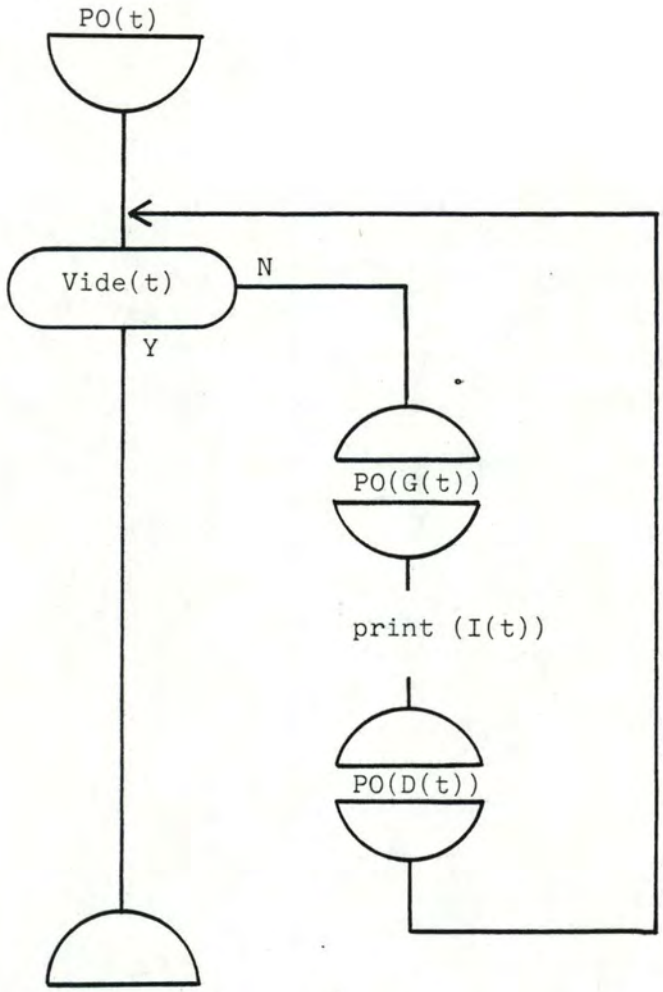
1.4. : Exemple d'utilisation de branchements

Nous allons construire ici un programme qui parcourt et imprime les informations liées aux noeuds d'un arbre binaire et ce, de façon infixée. On imprime donc d'abord les informations liées aux noeuds du sous-arbre de gauche; ensuite l'information liée à la racine, et enfin les informations liées aux noeuds du sous-arbre de droite.

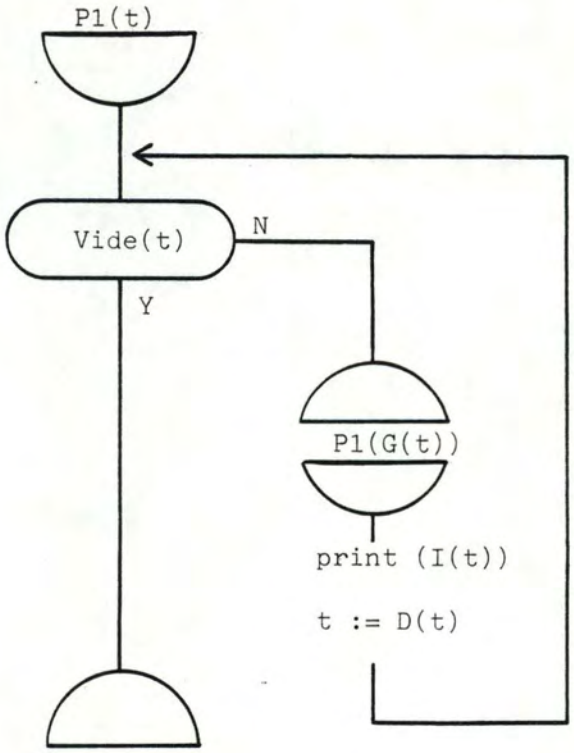
Soit t un arbre binaire. Supposons que nous disposons des primitives suivantes :

- vide(t) : fonction donnant le résultat vrai, si t est un arbre binaire vide. faux, sinon
- G(t) : fonction donnant le sous-arbre gauche de l'arbre binaire t.
- D(t) : fonction donnant le sous-arbre droit de l'arbre binaire t
- I(t) : fonction donnant l'information liée à la racine de l'arbre binaire t

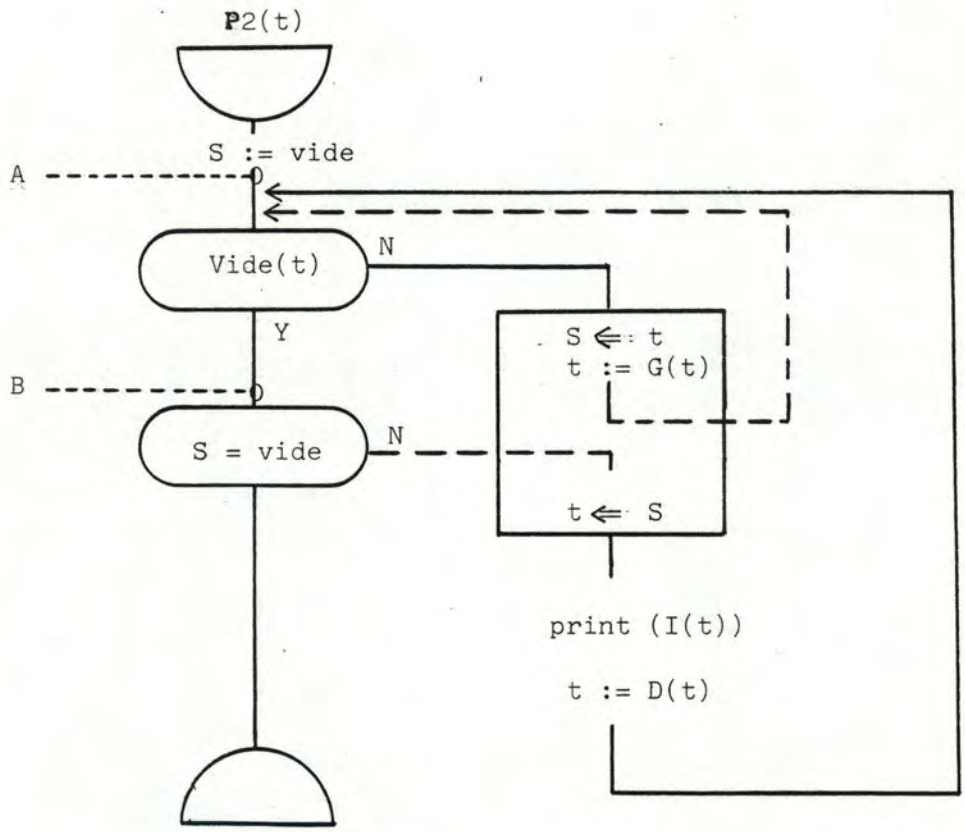
Par définition du parcours infixé, nous obtenons immédiatement le programme récursif suivant où les appels se font par valeur.



Supposons que le langage utilisé ne supporte pas la récursivité. Nous allons donc supprimer les 2 appels récursifs. Le second appel récursif peut être immédiatement retiré de la manière suivante.



Le retrait du premier appel récursif nécessite l'utilisation d'une pile S d'arbres binaires qui permettra de restaurer le contexte après exécution de l'appel.



Le programme obtenu contient un branchement qui ne peut se réduire à une répétitive sans l'ajout d'une nouvelle variable. Que penser de ce programme ? Les règles de transformation effectuées facilitent une démonstration du programme P2 par induction sur l'ensemble des arbres binaires bien fondé par la relation "est un composant de". Une telle démonstration établit d'abord : S'il se produit un passage en A

avec $S = \boxed{S_o}$, $f = \alpha$, $t = t_o$,

Alors il se produira un passage en B

avec $S = \boxed{S_o}$, $f = \alpha \,.\, rI(t_o)$

où f est le support d'impression

$rI(t_o)$ est la représentation imprimée de t_o.

1) vrai pour t_o vide

évident car $rI(t_o)$ est alors la chaîne de caractères vide

2) vrai pour t_o non vide

Soit tg = $G(t_o)$ et td = $D(t_o)$

Comme t_o est non vide, il se produira un nouveau passage au point A avec S = $\boxed{\begin{array}{|c|}\hline t_o \\ \hline S_o \\ \hline\end{array}}$ et t = tg

Par hypothèse d'induction et comme $tg<t_o$, il y aura un passage au point B avec S = $\boxed{\begin{array}{|c|}\hline t_o \\ \hline S_o \\ \hline\end{array}}$, f = $\alpha.rI(tg)$

Comme S ≠ vide, il se produira un nouveau passage au point A avec S = $\boxed{S_o}$, f = $\alpha.rI(tg).I(t_o)$ et t = td.

Par hypothèse d'induction et comme $td<t_o$, il y aura au passage au point B avec S = S_o et f = $\alpha.rI(tg).I(t_o).rI(td)$

La propriété est donc vraie pour t_o vu que

$rI(t_o) = rI(tg).I(t_o).rI(td)$

La correction du programme se montre en appliquant cette propriété lors du premier passage au point A.

Le programme P2 a été obtenu par un raisonnement descendant élémentaire donnant PO qui a été ensuite transformé.
Nous pensons qu'il aurait été plus difficile d'obtenir directement un programme et de le prouver sans utiliser implicitement ou explicitement la démarche effectuée ici.

Nous considérons que le branchement utilisé ici est acceptable parce que le raisonnement le plus simple conduit naturellement à un programme qui en contient un.

1.5. : Programmation avec "goto"

Les règles de transformation étudiées précédemment nous permettent de croire que, actuellement, la qualité d'un programmeur ne peut plus se juger à la densité des branchements écrits dans ses programmes. Il est en effet toujours possible de les retirer sans pour autant améliorer le programme. Le critère d'absence de branchements comme critère de qualité d'un programme apparaît donc en définitive arbitraire. Il n'y a pas la moindre raison de croire que le raisonnement le plus simple et le plus direct conduira automatiquement à un programme sans branchements. Ce qui compte, c'est que le raisonnement soit simple et que le programme en soit une traduction aussi fidèle

que possible. Or la traduction la plus fidèle et la plus directe peut fort bien avoir pour effet l'introduction de branchements.

Nous avons dit pour débuter qu'un programme sans branchement permet de mieux se représenter le déroulement de l'exécution de ce programme. Ce principe est valable si on admet que pour comprendre un programme, il n'y a pas d'autres moyens que d'essayer d'imaginer son exécution. Or comprendre un programme, ce n'est pas comprendre comment une machine l'exécute, c'est comprendre le raisonnement qui a été mené lors de sa construction, de même que le lien entre ce raisonnement et le texte du programme.

Nous n'avons pas abordé le problème de l'efficacité d'un programme. Signalons tout d'abord que le retrait des branchements entrave l'efficacité d'un programme. Ensuite, nous ne pensons pas qu'un programme simple et compréhensible soit en général nettement plus inefficace qu'une version optimisée mais incompréhensible. L'expérience montre que la plupart du temps d'exécution d'un programme non interactif est concentré dans près de 3% du texte source [KNU,74]. Ainsi, si le critère d'efficacité est important, il est préférable de construire un programme simple dont on optimisera, par transformation contrôlée et expliquée, les parties critiques.

La présence ou l'absence de "goto" ne peut être le centre de nos préoccupations. Le but réel est d'être capable de formuler un programme de telle façon qu'il soit facilement compréhensible. Nous terminerons avec une phrase de Dijkstra par qui la "goto less programming" a débuté.

> "Please don't fall into the trap of believing that I am terribly dogmatical about the go to statement. I have the uncomfortable feeling that others are making religion out of it, as if the conceptual problems of programming could be solved by a single trick, by a simple form of coding discipline."[KNU,74]

2. : PROGRAMMATION STRUCTUREE

2.1. : Objectifs

Toute méthode de programmation veut atteindre l'objectif de programmation structurée. Malheureusement, ce mot a été tellement utilisé à des fins diverses qu'il n'a actuellement plus beaucoup de sens comme tel. Nous allons essayer de voir ce que pourrait recouvrir ce concept et comment les idées exposées dans la première partie rencontrent celui-ci.

L'objectif de la programmation structurée veut être de proposer, plutôt que des règles, un ensemble d'idées qui, idéalement, permettraient la construction d'un programme sans erreurs.

2.2. : Idées principales

La notion de programmation structurée a été introduite par Dijkstra dans [DIJ,72a]. Selon lui, la question critique est de savoir pour quelles structures de programme l'on peut donner une preuve de correction sans travail excessif, même si le programme est grand. Par preuve de programme, Dijkstra entend non pas nécessairement dérivation formelle mais n'importe quelle sorte de preuves (formelles ou non formelles) qui soit suffisamment convainquante et qui permette de comprendre le programme [KNU,74].

Une chose est réellement claire : la programmation structurée ne se réduit pas au fait d'écrire des programmes et d'éliminer ensuite les instructions de branchements. On doit être capable de définir la programmation structurée sans référence aux instructions de branchements; le fait que leur utilisation sera rarement nécessaire n'en serait alors qu'une conséquence.

Une première idée [LER,75b] est de dégager des heuristiques, c'est à dire des procédés ou tours de mains grâce auxquels on pourra construire avec succès un programme résolvant un problème donné. Dans la majorité des cas, la résolution d'un problème de programmation présente peu de difficultés de principe. La vraie difficulté est de formuler explicitement, dans les moindre détails et sans erreurs, l'algorithme de résolution.

Une seconde idée est la structuration du processus de construction d'un programme. D'ailleurs, pour Hoare, la définition de la programmation structurée est l'utilisation systématique du processus d'abstraction pour contrôler l'ensemble des détails intervenant dans la construction d'un programme, de même qu'un moyen de documentation qui aide le "design" [KNU,74]. Dans notre optique, cette abstraction revient à exploiter systématiquement la notion de primitive . Nous sommes entièrement d'accord avec Dijkstra lorsqu'il dit : "A good way to express the abstract properties of an unwritten piece of program often helps us to write that program, and to "know" that it is correct as we write it" [KNU,74]. Ceci rejoint l'idée développée précédemment que pour construire un programme, on doit tout d'abord élaborer une "théorie" qui permettra de dégager certaines propriétés des concepts relatifs au problème à résoudre.

3. : FORMALISME ET CONSTRUCTION DE PROGRAMMES

3.1. : Formalisme de Hoare

3.1.1. : Nature de ce formalisme

Dans [HOA,69] et[HOA,71], Hoare propose un formalisme et une extension de la méthode des assertions inductives. Il introduit tout d'abord la notation {P} S {Q} déjà exposée p 1.14 . Ensuite,il présente les éléments fondamentaux d'un système formel dans lequel on pourrait démontrer des formules de la forme {P} S {Q}. La démonstration d'un tel théorème établit la correction partielle du programme S pour la précondition P et la postcondition Hoare ramène ainsi la démonstration de correction partielle d'un programme à une démonstration d'une formule dans son système formel.

3.1.2. : Limites de ce formalisme

1. Hoare reconnait que, comme toute preuve formelle, la démonstration formelle de correction partielle d'un programme est excessivement fatigante et pénible. Il propose alors de "dériver" des règles générales pour les démonstrations de correction, à partir des règles simples exposées dans ce formalisme. La validité de ces règles générales pourrait être établie en démontrant (non formellement) comment chaque théorème prouvé à l'aide de ces règles pourrait l'être parallèlement sans l'aide de ces dernières (beaucoup plus péniblement). Moyennant cet ensemble de règles "dérivées", une "preuve formelle" se réduira à un peu plus qu'une indication informelle de la façon de construire une preuve formelle [HOA,69].

   Ce point de vue contient un cercle vicieux La variété des propriétés possibles d'un programme est telle qu'on devra pratiquement définir pour chaque nouveau programme, de nouvelles règles "dérivées". La justification de ces règles reviendra alors à donner une preuve non formelle du programme.

2. Les axiomes définis par Hoare ne sont adéquats que s'il n'y a pas d'effet de bord dans l'évaluation des expressions. Il faudra donc prouver, en dehors du système formel, qu'il n'y a pas

d'effets de bord dans le programme donné. De plus, ce formalisme ne permet pas d'exprimer certaines notions comme la manipulation de caractères et de fichiers, les déclarations, les pointeurs, les exécutions parallèles, ... L'introduction de ces notions dans le formalisme se répercuterait sur la complexité des axiomes de base.

3. Cette technique de démonstration peut être utilisée pour prouver la correction d'un programme, pourvu que ses spécifications puissent être décrites rigoureusement sous forme de formules de ce système. D'une manière générale, comment savoir qu'une formule de ce système exprime bien ce que l'on a voulu y mettre ? Si l'on dit qu'une spécification est une formule de ce système, on retombe alors dans les spécifications formelles et le cercle vicieux de la vérité par définition exposé au ch I.3.

3.1.3. : Ambiguïté liée à ce formalisme

Considérons, par exemple, le programme de division entière proposé dans [HOA,69] :
(r:=x; q:=0; WHILE y $\leq$ r DO(r:=r-y; q:=q+1))
Notons le S. Il faut montrer qu'il effectue l'opération
(q,r):= (x DIV y, x MOD y).
Pour démontrer la correction partielle, Hoare établit :
{vrai} S {x = q*y+r et r<y}.
Mais ce résultat n'exprime pas la correction partielle du programme car rien n'affirme dans cette formule, que le programme ne modifie pas les valeurs de x et y. On pourrait démontrer la même formule pour le programme
(r:=0; q:=x; y:=1).
Or ce dernier programme n'effectue généralement pas l'opération désirée [LEC,81a].

Pour prouver la correction partielle, il faudrait démontrer par exemple que pour tous x_o, y_o entiers :
{x=x_o et y=y_o} S {x=x_o et y=y_o et x=q*y+r et r<y}
Il est donc nécessaire d'écrire des formules contenant des variables x_o et y_o désignant les mêmes valeurs avant et après exécution. Or Hoare ne donne pas de réponse à la question suivante : les formules P et Q peuvent-elles, oui ou non, contenir d'autres variables que celles du programme S et, si oui, quelle interprétation donner à la formule {P} S {Q} ?

Les règles de Hoare peuvent ainsi être interprétées différemment selon la réponse apportée à cette question [LEC,81a].

On pourrait dire, concernant cet exemple, que le fait que x et y ne changent pas de valeurs est évident au vu du programme. C'est vrai, mais en l'affirmant, on utilise la sémantique intuitive du programme, et ce raisonnement ne peut être exprimé dans le formalisme. Donc, pour démontrer la correction partielle de ce programme, en plus d'une démonstration formelle, il faut effectuer une seconde démonstration, non formelle, qui prouve l'invariance de certaines variables.

3.2. : Définition de la sémantique d'un langage de programmation

3.2.1. : Algorithme, sémantique et langage de programmation

3.2.1.1. : Algorithme et langage de programmation

La notion d'algorithme est intuitive et il est difficile d'en donner une définition satisfaisante. Nous dirons qu'un algorithme est un procédé de calcul suffisamment bien défini pour qu'il soit théoriquement possible de réaliser une machine capable d'exécuter ce procédé avec une précision absolue [LEC,81b]. Ceci dit, pour communiquer un algorithme, on doit en donner une description ou représentation dans un langage et se pose alors la question du "bon langage". Le langage naturel est parfaitement adapté pourvu que la description de l'algorithme soit accompagnée d'un raisonnement établissant sa validité. Cela signifie, en particulier, que l'utilisateur de l'algorithme doit être capable de comprendre à quoi sert l'algorithme (sans quoi la notion de validation n'aurait pour lui aucun sens). Mais le fait d'utiliser le langage naturel n'interdit pas d'introduire des conventions ou abréviations du langage. De telles conventions sont particulièrement utiles dans le cas de la description d'algorithmes car la variété des opérations qu'on est amené à spécifier est généralement très restreinte. Il est donc possible d'exprimer complètement de vastes classes d'algorithmes en utilisant uniquement un ensemble d'abréviations fixées une fois pour toute. On a alors défini un langage de programmation.

Un langage de programmation peut donc être considéré comme une extension du langage naturel constitué d'abréviations qui se sont révélées particulièrement utiles pour la description des algorithmes. Si un tel langage est formel, c'est parce qu'on a fixé l'ensemble des abréviations utilisables pour qu'une machine puisse exécuter tous les algorithmes exprimables dans ce langage. Une telle formalisation n'apporte rien de plus sur la nature des algorithmes eux-mêmes. C'est la syntaxe qui est formalisée; pas la sémantique.

Le désavantage de cette formalisation de la syntaxe est que, très rapidement, on aura envie d'introduire de nouvelles abréviations; et faute de pouvoir le faire, il faudra se résoudre à écrire des périphrases dans le langage de programmation. C'est le prix à payer pour qu'un algorithme puisse être exécuté par une machine.

3.2.1.2. : Sémantique d'un langage de programmation

Selon cette optique, un texte écrit dans un langage de programmation (ou programme) désigne un algorithme. Définir la sémantique d'un tel langage, c'est donc fournir un moyen de connaître l'algorithme représenté par un programme quelconque.

La définition de la sémantique d'un langage de programmation peut se faire de plusieurs façons. On peut tout d'abord décrire comment le programme est exécuté; on décrit ainsi l'algorithme représenté par celui-ci. On obtient une sémantique opérationnelle du langage de programmation.

Une autre idée part du désir d'obtenir une définition de la sémantique d'un langage de programmation qui permette de raisonner rigoureusement. On attache donc à chaque programme, non un algorithme, mais un "objet mathématique" qui sera, par exemple, une fonction partielle ou un transformateur de prédicat. On obtient ainsi ce que nous appellerons une sémantique mathématique du langage de programmation. Une telle définition est insuffisante pour que ce langage soit exécuté par une machine. On devra alors ajouter un ensemble de règles d'interprétation pour qu'il puisse être implémenté.

3.2.2. : Sémantique opérationnelle

La sémantique opérationnelle d'un programme est une description de " ce qui se passe à l'intérieur de la machine" durant son fonctionnement; c'est-à-dire la suite des modifications effectuées sur le contenu des variables entre l'instant initial et l'arrêt du programme.

Pour exprimer cette sémantique, il faut utiliser un langage en dehors du langage de programmation. On peut utiliser soit le langage naturel, soit un langage formel. Une sémantique opérationnelle formelle consiste à décrire une machine abstraite et à montrer comment se passe l'exécution du programme sur cette machine abstraite. En tant que langage formel, il devra donc être défini finalement à l'aide du langage naturel. La sémantique du langage PL/I a, par exemple, été entièrement décrite à l'aide d'un langage formel. Il semble que cela n'ait pas été une réussite; très peu d'utilisateurs exploitent cette définition pour comprendre ce langage, de même, il n'est pas certain que les compilateurs respectent cette sémantique formelle.

Avec le langage naturel, la sémantique d'un programme sera fixée en lui associant un texte du langage naturel que l'on comprend et qui désigne le même algorithme. Ce texte ne constitue pas la "vraie" signification de programme, mais il est seulement une façon parmi d'autres d'énoncer l'algorithme qu'il représente. Quelqu'un n'aura bien compris la sémantique d'un langage de programmation que lorsqu'il ne sera plus obligé de reformuler explicitement l'énoncé équivalent en langage naturel pour pouvoir raisonner au sujet d'un programme. Un bel exemple de définition de la sémantique en langage naturel est la définition du langage Algol [NAU,63], où la syntaxe est définie formellement selon une grammaire BNF et où la sémantique est exprimée en langage naturel.

3.2.3. : Sémantique mathématique

3.2.3.1. : Définition de la sémantique

Le système formel de Hoare permet de déterminer l'assertion que l'on peut attacher au point d'entrée d'une instruction connaissant celle que l'on a déjà attachée au point de sortie.

On peut considérer que ces règles déterminent la sémantique du langage. On obtient ainsi une des définitions mathématiques possibles d'un langage de programmation. Une autre définition plus complète car elle inclut la terminaison, est celle des transformateurs de prédicats telle que proposée par Dijkstra. Il déclare d'ailleurs [DIJ,75] qu'il prend la position que l'on connait suffisamment bien la sémantique d'un mécanisme S si l'on connait son transformateur de prédicat; c'est à dire que l'on peut dériver wp(S,R) pour toute postcondition R.

Nous allons essayer de montrer que cette définition de la sémantique est équivalente à la sémantique opérationnelle.
Soit x, une variable
Y, l'ensemble des autres variables
f, une fonction
$\beta(x,Y)$, une assertion
x:=f(x,Y), une affectation.
La sémantique mathématique de cette instruction d'affectation est définie par le fait que, pour que l'assertion $\beta(x,Y)$ soit vraie après son exécution, il suffit que l'assertion $\beta(f(x,Y),Y)$ soit vraie avant. La sémantique opérationnelle de cette affectation décrit que, si avant exécution de cette instruction nous avons l'état (x_o,Y_o), alors après exécution nous aurons l'état $(f(x_o,Y_o),Y_o)$.

Supposons que cette instruction soit exécutée selon la sémantique opérationnelle. Soit (x_o,Y_o), un état initial tel que l'on ait $\beta(x,Y)$ après exécution. Il faut donc que l'état $(f(x_o,Y_o),Y_o)$ vérifie $\beta(x,Y)$: ce qui est équivalent à dire que l'état (x_o,Y_o) vérifie $\beta(f(x,Y),Y)$ avant exécution. La sémantique opérationnelle implique donc cette sémantique mathématique.

Montrons maintenant que cette sémantique mathématique implique la sémantique opérationnelle. Pour le voir, il suffit de poser :
$\beta(x,Y) \leftrightarrow \{x = f(x_o,Y_o) \text{ et } Y = Y_o\}$
d'où on déduit l'assertion la plus faible à l'entrée :
$\{f(x,Y) = f(x_o,Y_o) \text{ et } Y = Y_o\}$,
assertion qui est impliquée par l'assertion plus forte (en général) :
$\{x = x_o \text{ et } Y = Y_o\}$.
Si donc cette assertion est vérifiée à l'entrée, on aura surement, à la sortie, quel que soient x_o et Y_o :
$\{x = f(x_o,Y_o) \text{ et } Y = Y_o\}$.

3.2.3.2. : critique

L'équivalence de ces deux sémantiques n'est pas fortuite; elle provient du fait que la sémantique mathématique est déduite de la connaissance qu'on a de la sémantique opérationnelle des différentes instructions. On peut se demander ce qu'on a gagné à cette opération. Notons qu'on peut considérer une assertion comme caractérisant un ensemble d'états. La définition de la sémantique opérationnelle d'une instruction est celle d'une relation entre état initial et état final; donc une relation binaire sur l'ensemble des états. La définition de la sémantique par la relation que doivent vérifier les assertions à l'entrée et à la sortie est celle d'une relation entre deux sous-ensembles d'états: c'est donc une relation binaire sur l'ensemble des parties de l'ensemble des états. Le gain est plutôt douteux [LER,78].

Avec cette sémantique mathématique, une démonstration d'un programme S consistera à "calculer" wp(S,POST) pour une proposition POST fixée par l'énoncé de la spécification de S, et de vérifier que la précondition fixée par l'énoncé de la spécification entraine wp(S,POST). L'illusion serait de croire que la sémantique mathématique facilite une démonstration car les règles de calcul ressemblent d'avantage à un algorithme qu'à une simple substitution de symboles. Ainsi, par exemple, le calcul de wp(a[i*k]:=a[j*l], a[i+j]=a[k+l]) ne peut s'effectuer à partir d'une règle de subtitution simple et si une telle règle existe, il ne peut s'agir que d'un algorithme compliqué de manipulation de formules, fortement lié au système formel choisi [LEC,81b]. Une définition mathématique de la sémantique ne pourra aider substantiellement à faciliter une démonstration de programme. L'optimisme injustifié consiste à croire qu'en exprimant une sémantique entre termes de relations entre prédicats, on rendra plus facile à établir, dans chaque cas particulier, le lien mental entre l'exécution et les spécifications. Puisque, de toutes façons, on ne peut rien définir d'autre , en toute généralité, que des sémantiques rigoureusement et trivialement équivalentes à des règles d'exécution, mieux vaut donner directement ces règles [LER,78].

3.3. : Méthode "formelle" de Dijkstra

Nous allons donner ici quelques aperçus des différences significatives entre la méthode de construction que nous avons exposée dans la première partie et la méthode de construction de programmes proposée par Dijkstra dans [DIJ,75] et [DIJ,76]. D'une façon générale, nous procédons d'une manière moins formelle à tous les niveaux.

3.3.1. : Choix du langage de programmation

Dijkstra utilise le langage des "guarded commands"; nous avons préféré utiliser ici les organigrammes pour le problème que nous avions à résoudre. Néanmoins nous n'affirmons pas que l'organigramme est la meilleure façon d'exprimer un algorithme en général. Pour certains problèmes, un autre langage pourrait d'ailleurs être préférable.

Nous pensons que les organigrammes sont agréables pour exprimer un algorithme. De plus, le passage d'un algorithme à un langage de programmation ne pose pas de problème. Comme les organigrammes forment un langage non implémenté, nous avons toute liberté de représentation, de convention et de notation. Enfin, nous estimons que les organigrammes permettent d'établir facilement le lien entre l'algorithme et le raisonnement effectué lors de la construction ou la démonstration de celui-ci.

3.3.2. : Choix du langage de spécification et d'expression des raisonnements.

Dans notre optique, il n'y a pas de restrictions à faire sur le langage d'expression des spécifications ni sur celui d'expression des raisonnements (logique des prédicats pour Dijkstra). Nous n'imposons donc pas de restrictions arbitraires sur les propriétés exprimables ni sur la manière de les démontrer. Nous pensons que c'est une grande illusion de croire qu'en essayant de formaliser les raisonnements, on y verra plus clair dans la structure de ceux-ci et qu'il n'y a pas de formalisation, surtout quand il s'agit de quelque chose d'aussi délicat que le raisonnement, sans déformations, mutilations et limitations arbitraires [LER,78].

Nous croyons que les avantages de la formalisation (absence d'ambiguité,...) ne justifie pas ces désavantages. Mais cela ne nous empêche pas d'emprunter certaines notations à ces systèmes lorsqu'elles nous sembleront utiles. De plus, ces notations pourront toujours être étendues n'importe quand et n'importe comment. La seule règle, en définitive est "Sois rigoureux et fais ce qu'il te plait" [LEC,79b]. La philosophie est que, sachant que l'on est pas infaillible, il faut faire tout ce qui est humainement possible pour ne pas se tromper et que, si un raisonnement non formel ne donne pas de garantie absolue, il vaut immensément mieux que pas de raisonnement du tout, ou qu'un raisonnement formel illisible et incompréhensible qui, aussi mécanisé qu'on le suppose, ne donnera pas non plus de garanties absolues [LER,78].

3.3.3. : Méthode de construction et de démonstration

Dijkstra raisonne théoriquement par invariants; c'est à dire par récurrence sur la structure de l'exécution. En général, comme nous l'avons exposé dans la première partie, nous raisonnerons plutôt par raisonnement descendant sur la structure des données tout en admettant que pour certains problèmes, il est plus simple de raisonner par invariants ou encore par raisonnement ascendant sur les données. Ici encore, nous ne nous limiterons pas à une méthode déterminée, mais on préférera utiliser une méthode adaptée au problème à résoudre. Pour s'en convaincre, il suffira de comparer deux constructions d'un même programme dont l'une très simple est réalisée par raisonnement descendant et l'autre, beaucoup plus complexe est construite par invariants. Ces deux constructions sont présentées dans [MAR,79] page 78. Signalons que la construction par invariants est effectuée par Dijkstra.

Enfin, nous ne croyons pas qu'il soit possible de déduire les invariants à partir des spécifications et les programmes à partir de ces invariants. Nous admettons qu'il est nécessaire d'avoir au départ une idée intuitive de résolution qu'on essaye alors d'exprimer rigoureusement en un programme qui résout le problème en toute généralité.

C O N C L U S I O N

Sous ce titre bien ambitieux, nous rappellerons quelques idées qui nous paraissent importantes et que nous voulons souligner une dernière fois.

1. Un programme est l'aboutissement d'un raisonnement. C'est ce raisonnement, et non le fait que le programme ait passé avec succès un petit nombre de tests, qui fonde la confiance que l'on peut en avoir et qui garantit tant bien que mal sa validité. La démonstration de correction d'un programme n'est que l'expression de la conviction que l'on a de la correction de celui-ci. Que les essais empiriques restent indispensables montre simplement que les raisonnements comportent souvent des lacunes.

2. Nous pensons que la spécification et la résolution d'un problème de programmation nécessite l'élaboration d'un ensemble de concepts ou d'une "théorie" propre à ce problème. Une spécification exprimera alors une propriété du programme qui permettra de l'utiliser comme une primitive.

3. Le problème de la construction d'un programme se pose au moment où l'on a non seulement l'énoncé du problème à résoudre, mais également une idée intuitive de résolution qu'on essayera alors d'exprimer rigoureusement en un programme résolvant le problème en toute généralité.

4. Nous rejetons deux types de formalisation : celle des raisonnements et celle des spécifications. La formalisation des démonstrations implique, entre autres choses, la formalisation des spécifications. Un tel langage de spécification devrait permettre d'exprimer des concepts aussi variés que les applications informatiques. Nous pensons qu'il est impossible d'obtenir un langage de spécification à la fois complètement formalisé et bien adapté à tous les problèmes. De plus, l'expression formelle d'une spécification devrait être suffisamment immédiate pour ne pas donner lieu à autant d'erreurs que la programmation elle-même.

5. Parmi les méthodes de construction et de validation, nous croyons que les raisonnements sur les données sont en général plus simples, plus clairs et plus faciles à utiliser pour construire un programme que la méthode des assertions inductives, bien que cette dernière soit beaucoup plus répandue. Nous regrettons que les méthodes de raisonnement sur les données soient si peu développées dans la littérature.

Nous terminerons en citant H.Leroy qui résume, en quelques mots, les idées que nous avons développées tout au long de ce travail :

> "Le raisonnement est l'âme de la programmation. Ce n'est pas une opinion, c'est une évidence ... La seule chose qui compte en définitive c'est (à part l'efficacité d'exécution) la simplicité, la clarté et la rigueur des raisonnements ... Or, si on veut atteindre la maîtrise du raisonnement, il faut le pratiquer explicitement, surtout s'il est peu intuitif, ce qui est généralement le cas en programmation ... Ma philosophie est que, sachant qu'on n'est pas infaillible, il faut faire tout ce qui est humainement possible pour ne pas se tromper ..."[LER,78].

*

* *

REFERENCES

[ASH,72] ASHCROFT, E., MANNA, Z. : *The Translation of "GO TO" Programs to "WHILE" Programs.* Information Processing 71, North-Holland Publishing Company, 1972, 250-255.

[BER,81] BERGLAND, G.D. : *A Guided tour of Program Design Methodologies.* IEEE computer, October 1981, 13-37.

[BOH,66] BOHM, C., JACOPINI, G. : *Flow-diagrams, Turing machines, and languages with only two formation rules.* Comm. ACM 9(5), May 1966, 366-371.

[BOY,81] BOYER, R.S., MOORE, J.S. : *The correctness problem in Computer science.* International Lecture Series in Computer Science, Academic Press, 1981.

[BUR,69] BURSTALL, R.M. : *Proving properties of programs by structural induction.* Computer Journal, 72, 1969, 41-48.

[BUR,74] BURSTALL, R.M. : *Program proving as hand simulation with a little induction.* Proceeding IFIP congress, 1974, 308-312.

[DEM,79] DE MARNEFFE, P.A. (Ed.) : *Dijkstra's Lectures on the Design of Correct Programs.* Chaire internationnale d'informatique FNRS-IBM, 1973.

[DIJ,68a] DIJKSTRA, E.W. : *Go To statement Considered Harmful.* Comm. ACM 11(3), March 1968, 147-148.

[DIJ,68b] DIJKSTRA, E.W. : *The Structure ot the "THE"-Multiprogramming System.* Comm.ACM 11(5), May 1968, 341-346.

[DIJ,72a] DIJKSTRA, E.W. : *Notes on Structured Programming.* A.P.I.C. Studies in Data Processing, No 8, Academic Press, London and New-York, 1972.

[DIJ,72b] DIJKSTRA, E.W. : *The Humble Programmer.* Comm. ACM 15(10), October 1972, 859-866.

[DIJ,75] DIJKSTA, E.W. : *Guarded Commands, Nondeterminancy and Formal Derivation of Programs.* Comm. ACM 18(8), August 1975, 453-457.

[DIJ,76] DIJKSTRA, E.W. : *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, N-J, 1976.

[FLO,67] FLOYD, R.W. : *Assigning Meanings to Programms.* Proceedings of a Symposium in Applied Mathematics, Vol 19, Mathematical Aspect in Computer Science. Providence, Rhode Island, American Society, 1967, 19-32.

[GOO,75] GOODENOUGH, J.B., GERHART, S.L. : *Toward a Theory of Test Data Selection.* IEEE Trans. on Softw. Eng., Vol SE-1(2), June 1975, 156-160.

[GRI,76] GRIES, D. : *An illustration of Current Ideas on the Derivation of Correctness Proofs and Correct Programs.* IEEE Trans. on Softw. Eng., Vol SE-2(4), December 1976, 238-244.

[HOA,69] HOARE, C.A.R. : *An Axiomatic Basis for Computer Programming.* Comm. ACM 12(10), October 1969, 576-583.

[HOA,71] HOARE, C.A.R. : *Procedures and Parameters : an Axiomatic Approach.* Symposium on Semantics of Algorithmic Languages, Lecture Notes in Mathematics, Springer Verlag , 1971, 102-115.

[KER,76] KERNIGHAN, B.W., PLAUGER, P.S. : *Software Tools.* Addison-Wesley, Reading, Mass., 1976.

[KNU,74] KNUTH, D.E. : *Structured Programming with* go to *Statements.* Computing Surveys, Vol 6(4), December 1974, 261-301.

[LAM,78] LAMOITIER, S.P. : *Le langage Fortran* IV Dunod, Paris 1978.

[LEC,79a] LE CHARLIER, B. : *Raisonnement non formel en programmation.* Communication présentée à la réunion du Groupe de Contact "Computer Science-Metatheories and Theories", FNRS, Mars 1979.

[LEC,79b] LE CHARLIER, B. : *Correspondance avec* SINTZOFF, M. Mars 1979.

[LEC,80] LE CHARLIER, B. : *Définition du langage* LSD 80 et *Spécifications des procédures de l'interpréteur du langage* LSD 80 . Non publié, 1980.

[LEC,81a] LE CHARLIER, B. : *La formalisation, source d'ambiguïtés et d'obscurités : quelques exemples relatifs au formalisme de Hoare.* Non publié, 1981.

[LEC,81b] LE CHARLIER, B. : *Sémantique des langages de programmation.* Non publié, 1981.

[LEC,82] LE CHARLIER, B. : *Séminaire de programmation.* Notes de cours, 1982.

[LEC,83] LE CHARLIER, B. : *L'ordinateur est-il infaillible ?* La revue nouvelle, à paraître.

[LER,75a] LEROY, H. : *Soms problems with the Design of Large Languages.* Institut d'informatique, Fac. Univ. Namur, February 1975.

[LER,75b] LEROY, H. : *La fiabilité des programmes.* Travaux de l'Institut d'Informatique de Namur No 3, Namur, 1975.

[LER,78] LEROY. H. : *La fiabilité des programmes.* Notes de cours, Ecole d'été de l'AFCET, 1978.

[LER,80] LEROY, H. : *Qu'est-ce qu'un programma correct ?* Non publié, 1980.

[LER,81a] LEROY, H. : *Théorie des langages.* Notes de cours,1981.

[LER,81b] LEROY, H. : *Méthodologie de la programmation.* Notes de cours, 1981.

[LIV,78] LIVERCY,C. : *Théorie des programmes : schémas, preuves, sémantiques.* Dunod informatique, Paris 1978.

[MEY,82] MEYER,B. : *Principles of Package Design.* Comm. ACM 25(7),419-428.

[MOR,77] MORRIS, J.H., WEGBREIT,B. : *Subgoal Induction.* Comm. ACM 20(4), April 1977, 209-222.

[NAU,63] NAUR,P. (Ed.) : *Revised Report on the Algorithmic Language* ALGOL 60. Comm. ACM 6(1), January 1963, 1-17.

[NAU,69] NAUR, P. : *Programming by Action Clusters.* BIT 9, 1969, 250-258.

[PAR,72a] PARNAS, D.L. : *A Technique for Software Module Specification with examples.* Comm. ACM 15(5), May 1972, 330-336.

[PAR,72b] PARNAS, D.L. : *On the Criteria Be Used in Decomposing Systems into Modules.* Comm. ACM 15(12), December 1972, 1053-1058.

[PAR,79] PARNAS, D.L. : *Designing Software for Ease of Extension and Contraction.* IEEE Trans. on Softw. Eng., Vol SE-5(2), March 1979, 128-138.

[ROB,81a] ROBILLARD, P.N., PLAMONDON, R. : *Harness a Computer to write Better Software, Faster.* Electronic Design 29, July 1981, 125-129.

[ROB,81b] ROBILLARD, P.N., PLAMONDON, R. : *An Interactive Tool for Descriptive, Operational and Structural Documentation.* Proc. 23th IEEE Computer Society Intl. Conf., September 1981, 291-295.

[ROB,82a] ROBILLARD, P.N., PLAMONDON, R. :*Planning for software tool implémentation : experience with Schemacode.* National Computer Conference, 1982, 750-757.

[ROB,82b] ROBILLARD, P.N., PLAMONDON, R. : *Outil pour la création automatique de programme structuré en Fortran.* Ecole Polytechnique de Montréal, 1982.

[ROB,77] ROBINSON, L., LEVITT, K.N. : *Proof Techniques for Hierarchically Structured Programs.* Comm. ACM 20(4), April 1977, 271-283.

[VAN,81] VAN LAMSWEERDE, A. : *Quelques principes pour le développement d'une architecture logicielle.* Institut d'informatique, Fac. Univ. Namur, 1981.

[VAN,82a] VAN LAMSWEERDE, A. : *Les outils d'aide au développement de logiciels : un apperçu des tendances actuelles.* XV^{ème} journées internationales de l'informatique et de l'automatisme, Juin 1982.

[VAN,82b] VAN LAMSWEERDE, A. : *Automatisation de la production de logiciels d'application : quelques approches.* 1ère partie. T.S.I. -Technique et Science Informatiques 1(6), Nov-Dec 1982, 475-494.

[VAN,83] VAN LAMSWEERDE, A. : *Automatisation de la production de logiciels d'application : quelques approches.* 2ème partie. T.S.I. -Technique et Science Informatique 2(1), Janv-Fev 1983, 5-20.

[WIR,71] WIRTH, N. : *Program Development by Stepwise Refinement.* Comm. ACM 14(4), April 1971, 221-227.