UNIVERSITE DE METZ

INSTITUT NATIONAL POLYTECHNIQUE DE LORRAINE

# THESE

## TROISIEME CYCLE INFORMATIQUE

# UN PROTOTYPE D'ANALYSEUR SEMANTIQUE POUR UN SOUS-ENSEMBLE DE PASCAL.



PAR J.P. JUNG



THESE SOUTENUE LE 31 MAI 1983 DEVANT LE JURY :

PRESIDENT P. COUSOT

EXAMINATEURS B. LOHRO

M. SINTZOFF

J.P. THOMESSE

Je remercie M. P. COUSOT (Professeur à la Faculté des Sciences de Metz) de m'avoir accepté dans son équipe de recherche et de m'avoir permis de travailler avec lui ; je lui suis reconnaissant de m'avoir donné le présent sujet de thèse et de m'avoir suivi et si bien conseillé tout au long de ce travail ; je le remercie pour toute son aide et tout son dévouement à mon égard ; je le remercie également d'avoir présidé le jury de ma thèse.

Je remercie

M. B. LOHRO (Professeur à l'Université d'Orléans)

M. H. SINTZOFF (Professeur à l'Université Catholique de Louvain [B])

M. J.P. THOMESSE (Professeur à l'ENSEM)

d'avoir accepté de faire partie du jury de la présente thèse ainsi que MM APT et NIVAT d'avoir assisté à la soutenance.

Je remercie M. C. PAIR qui, en tant que Président de l'Institut National Polytechnique de Lorraine, m'a permis de commencer mes études de 3ème cycle en Informatique.

Je remercie M. P. BOUCHET (Assistant à l'Université de Nancy I), M. J. DUCLOY (Président de l'A.N.L.) et M. D. HERY (Assistant à la Faculté des Sciences de Metz) de m'avoir aidé lors de l'implantation du prototype d'analyseur sémantique.

Tous mes remerciements vont également à tous mes collègues de l'I.U.T. de Metz qui m'ont aidé et soutenu, et en particulier à Mme M. JUNG pour son excellent travail de dactylographie.

Jean-Pierre JUNG

TABLE DES MATIERES
==========

7. binary - search

8. pged - tableau

VI Etude du graphe

1. introduction

2. parcours avec priorité à la profondeur
   - définitions
   - parcours en profondeur
   - algorithme de parcours en profondeur

3. recherche des ensembles minimaux de coupure dans un graphe réductible
   - introduction
   - définitions
   - ensembles de coupure
   - ensembles de coupure minimaux dans les graphes réductibles

4. ensembles de coupure dans les graphes quelconques
   - introduction
   - modifications
   - algorithme
   - exemple

5. composantes fortement connexes
   - introduction
   - connection forte
   - algorithme

6. décomposition d'un graphe en composantes fortement connexes
   - introduction
   - problème
   - algorithme

7. remarques

8. graphe - inverse du graphe

9. algorithme récursif
   - introduction
   - méthode
   - algorithme

10. algorithme itératif
    - introduction
    - ensemble
    - exemple
    - algorithme final
    - conclusion
    - exemples

VII Pointeurs

1. introduction

2. variables de type pointeur
   - initialisation
   - valeurs abstraites
   - description
   - exemple à l'étape 1

3. collections de pointeurs
   - introduction
   - notion de collection
   - description
   - exemple

4. description

5. application à l'exemple

6. généralisation
   - notations et définitions
   - affectations
   - conditionnelles
   - jonction de chemins

7. analyse en arrière

8. conclusion

VIII conclusions

Bibliographie

## I Motivations

Les vérifications statiques de la conformité des programmes à la spécification semantique du langage qui sont faites par un compilateur sont souvent insuffisantes. Pour y remédier, des tests peuvent être placés par le compilateur dans le code objet afin de détecter les erreurs au moment de l'exécution d'un programme. Ces tests, s'ils ne donnent pas toujours satisfaction, occasionnent un supplément de temps non négligeable aussi bien à la compilation (jusqu'à 50 % du temps de compilation peut être consacré au placement des tests dans le code objet) qu'à l'exécution (le temps d'exécution peut être augmenté de 5 à 30 %).
C'est la raison pour laquelle de nombreux utilisateurs évitent leur mise en place, ce qui conduit fréquemment à des résultats très surprenants, comme pour cet exemple classique, écrit en langage Pascal :

```
program    essai ;
var   x :  record a, b, : 0.. 255 end ;
begin
      x.a : = 1 ;
      while   x.a<>0 do   x.a : = x.a + 1 ;
      write (x.a) ;
end.
```

Selon les normes ISO de définition du langage Pascal, le résultat de ce programme est indéfini. Cependant en pratique, il peut être parfaitement être exécuté par une machine et pourra conduite à des résultats différents , pour une même machine et un même compilateur, selon les directives de compilation. L'exécution de ce programme sans l'utilisation des tests à l'exécution se termine par une détection par le matériel d'un débordement arithmétique de la valeur du champ a de la variable x. La mise en place des tests à l'exécution modifie radicalement le résultat
puisque le programme se termine avec la valeur 255 affectée à x.a.
Modifions la déclaration de la variable x par :

```
x : packed record a, b : 0.. 255 end ;
```

Les champs a et b sont à présent compactés et ont chacun une occupation mémoire d'un octet.

> L'exécution avec tests de ce nouveau programme ne modifie pas le résultat obtenu précédemment c'est-à-dire que le programme s'arrête anormalement. Par contre sans test à l'exécution, le programme s'arrête normalement avec la valeur 0 pour x. a ; en effet, le champ a est initialisé à 1 et est incrémenté de 1 jusqu'à la valeur 255, alors la valeur est rangée dans un registre de plus d'un octet , la valeur est incrémentée de 1, atteind 256 (c'est-à-dire 1 suivi de huit chiffres 0 dans le registre) ; la valeur d'un octet (soit les huit chiffres 0) est rangée dans le champ x.a ; ce dernier vaut alors 0 et le programme s'arrête sans erreur. Il est pourtant clair que ce programme est incorrect.

Dans un tout autre ordre d'idée, le compilateur Pascal dû à Wirth place les tests à l'exécution lors de l'affectation d'une variable et non lors de son utilisation. Cette stratégie est économique mais incorrecte. Par exemple, pour le programme suivant :

```
programme   essai ;
var   x : 0.. 10 ;
begin
      write (x) ;
end.
```

La valeur finale de x n'a aucune raison de se trouver dans l'intervalle numérique [0,10].

En pratique, la plupart des tests placés à l'exécution sont redondants. Certaines stratégies naïves de compilation (Welsh [77]) permettent d'en éliminer une partie.

(par exemple, supposons la variable i déclarée entre les bornes 0 et 10, et supposons l'affectation i : = 5 ; les deux tests qui apparaissent pour vérifier que la valeur 5 est bien d'une part plus grande que 0, d'autre part plus petite que 10 peuvent être aisément éliminés).

Pour des exemples moins naïfs, le compilateur ne garde aucune trace du lien qui peut exister entre les valeurs des différentes variables du programme et de ce fait, impose la présence de tests qui s'avèrent rapidement inutiles et coûteux.

Une analyse sémantique permet de fournir au compilateur des informations permettant d'éviter la mise en place de tests coûteux et inutiles.

Si les informations fournies au compilateur permettent déviter de placer des tests à l'exécution inutilement, elles peuvent également permettre d'ajouter des tests que les compilateurs classiques ne placent pas dans le code objet, ou de déplacer la position de ceux qui y seraient placés. Les tests à l'exécution permettent de découvrir des erreurs une fois qu'elles se sont produites, trop tard pour observer le comportement du programme avant l'erreur. Une analyse sémantique peut permettre de déplacer un test au point le plus en amont où l'erreur est fatale. Enfin, certaines situations d'erreurs à l'exécution, comme la non-terminaison, ne peuvent être repérées par des tests à l'exécution. Une solution partielle peut être apportée par une analyse sémantique qui permet de déterminer des conditions nécessaires (si ce n'est suffisantes) de terminaison.

Pour terminer ces motivations concernant l'application de l'analyse sémantique des programmes à la conception de compilateurs plus évolués (sinon intelligents), indiquons qu'à chaque fois qu'un test à l'exécution peut être évalué statiquement, le compilateur a découvert une erreur et agit comme une aide à la mise au point statique.

## II Introduction

Nous avons orienté nos recherches vers l'analyse sémantique des programmes écrits en langage Pascal et avons, dans le cadre de cette thèse, réduit cette étude à un sous ensemble simple du langage Pascal.
Dans un premier chapitre, un paragraphe sera consacré à l'analyse sémantique décrite dans Cousot P[78] ; cette méthode est la base de toute l'étude que nous ferons ; nous allons expliciter et développer des fonctions sémantiques qui seront appliquées en vue de l'analyse de programmes dont les variables seront des varibales simples entières, ainsi que des tableaux d'entiers à indices entiers.

Nous avons implanté la méthode et construit ainsi un prototype que nous décrirons dans le second chapitre ; nous y verrons également divers exemples de programmes, l'explication des résultats fournis par l'analyseur ainsi que l'utilisation de ces résultats lors de la construction de cet analyseur, nous avons rencontré les problèmes que posent certaines contraintes dûes à la machine (par ex., le temps d'exécution), dûes aussi à la méthode (par ex., certaines approximations de résultats) ; pour ces raisons, nous avons, un peu en marge de l'analyse sémantique proprement dite, tenté de faire une étude de graphe de dépendance associé à un programme devant être analysé sémantiquement ; cette étude de graphes fait l'objet d'un quatrième chapitre.
Le cinquième chapitre traite de l'étude semantique des variables de type pointeur sur le tas.

## III - Analyse semantique .

Nous nous proposons dans cette thèse de reprendre l'analyse sémantique telle qu'elle a été définie par P. Cousot et R. Cousot et de l'automatiser pour le langage Pascal, en limitant toutefois notre étude à l'analyse des programmes écrits avec un sous-ensemble strict du langage Pascal, ce sous-ensemble sera progressivement étendu à Pascal tout entier (normes ISO).
Le choix du langage Pascal a été influencé par sa grande diffusion, par la lisibilité des programmes écrits dans ce langage, par la simplicité du compilateur ainsi que par les possibilités offertes par le langage de déclarer de manière assez précise le type des variables.

### 1. Méthode

Les détails de la théorie sur laquelle s'appuie cette méthode pourront être trouvés dans [ Cousot 78 ] . Dans ce même ouvrage, la méthode elle-même est explicitée et nous allons donner dans ce paragraphe un rapide résumé de tout ce dont nous aurons besoin dans la suite.

def : analyse sémantique

> L'analyse sémantique est une analyse statique (i.c. sans exécuter le programme) des propriétés dynamiques (i.e. à l'exécution) de ce programme.

Un programme que nous analyserons sera défini par l'ensemble E des états, par une relation t de transition: $t : E \times E \to B$ (où $B = \{ vrai, faux \}$ ) ainsi que trois fonctions caractérisant les états d'entrée, les états de sortie et les états d'erreur respectivement. La spécification d'un tel système comporte la donnée d'une spécification d'entrée $\phi$ et d'une spécification de sortie $\psi$ . Ceci exprimera l'intention que tout état d'entrée satisfaisant à la spécification d'entrée $\phi$ entraîne l'évolution du système vers un état satisfaisant à la spécification de sortie $\Psi$.

Pour faire l'étude du comportement d'un tel système, il faudra chercher à caractériser d'une part l'ensemble des ascendants des états satisfaisant

à une condition $C \in (E \to B)$ :

$pre\ (t^*)\ (c) : e_1 \to \{ e_2 \in E : t^*(e_1, e_2)$ et $c\ (e_2) \}$

d'autre part l'ensemble des descendants des états satisfaisant à une condition $c \in (E \to B)$

post $(t^*)$ $(c) : e_1 \longrightarrow \{ \exists e_2 \in E : c(e_2)$ et $t^*(e_1, e_2) \}$

pre $(t^*)$ et post $(t^*)$ seront obtenus comme point fixes d'une équation

Analyse sémantique en avant : descendants des états satisfaisant à une condition

Pour faire une analyse "en avant", nous devons caractériser l'ensemble Post $(t^*)$ $(\emptyset)$ des ascendants des états d'entrée satisfaisant à une condition d'entrée $\emptyset$ qui est isomorphe à la plus petite solution de l'équation

$X = \emptyset \vee$ post $(t)$ $(X)$

sur le treillis complet $(\Longrightarrow, \vee)$. Cette équation n'est pas aisément représentable en machine et sa plus petite solution n'est généralement pas calculable C'est pourquoi, en pratique, il faut se contenter de calculer une approximation supérieure I de post $(t^*)$ $(\emptyset)$ qui est invariante car post $(t^*)$ $(\emptyset) \Longrightarrow I$.

Pour ce faire, on imagine d'associer à chaque point de programme (dont les états sont des couples $< c, m >$ où $c \in C$ est un état de contrôle et $m \in M$ un état mémoire) une assertion $i_c$ $(m)$ choisie dans une classe d'assertions qui seront à priori jugées intéressantes et que l'on représente par un treillis. Une connection de Galois $(\alpha, \gamma)$ - où $\alpha \in (E \to \{tt, ff\}) \longrightarrow \prod_{c \in C} A$ est la fonction d'abstraction et $\gamma \in \prod_{c \in C} A \longrightarrow (E \to \{tt, ff\})$ est la fonction de concrétisation - définit la sémantique des assertions approchées $i_c$ $c \in C$ associées au programme.

Posant $F \in \prod_{c \in C} A \longrightarrow \prod A$, isotone tel que :

$\lambda X.[\alpha (\emptyset \vee$ post $(t)$ $(X))] \subseteq F$

la plus petite solution I du système d'équations sémantiques $X = F(X)$ satisfait la condition post $(t^*)$ $(\emptyset) \Longrightarrow \gamma(I)$.

Si le treillis A satisfait la condition de chaîne ascendante, cette solution est calculable itérativement $I = \bigcup_{n \geq 0} F^n (\alpha(\emptyset))$.

Sinon, on utilisera une itération chaotique avec stratégie d'extrapolation pour déterminer une approximation supérieure de I. On a alors obtenu un invariant en chaque point du programme.

Analyse sémantique en arrière : ascendants des états satisfaisant à une condition

Pour faire une analyse en arrière, nous devons caractériser l'ensemble des ascendants des états de sortie satisfaisant à une condition de sortie . soit pre $(t^*)$ $(\Psi)$, qui est isomorphe à la plus petite solution d'un système d'équations sémantiques en arrière $X = B(X)$ associé au programme étudié où $B = \lambda X.[\alpha(\Psi \vee pre(t)(\gamma(X)))$ .

Toute approximation supérieure de la plus petite solution du système d'équation $X = B(X)$ va caractériser en chaque point du programme les conditions nécessaires pour que l'exécution du programme se termine, ne se termine pas ou conduise à une erreur sémantique (selon le choix de $\Psi$ ).

Techniques d'extrapolation

Pour calculer une approximation supérieure I de la plus petite solution lfp (F) d'un système d'équation $X = F(X)$, on calcule la limite $I = \bigcup_{n \geq o} (\lambda X.\ X \nabla F(X))^n (\alpha(\emptyset))$ d'une itération croissante avec élargissement $\nabla$ satisfaisant :

$$\forall X\ Y \in \prod_{c \in C} A\ ,\quad \forall c \in C\ ,\quad (X \nabla Y)_c \supseteq X_c \sqcup Y_c$$

on s'assure que lfp $(F) \subseteq I$.

Ayant choisi $\nabla$ satisfaisant la condition qu'il n'existe pas de chaîne strictement croissante de la forme $X^\circ, \ldots X^{i+1} = X^i \nabla F(X^i), \ldots$, on s'assure de la convergence.

Il est ensuite possible d'améliorer l'approximation I de lfp (F) en calculant la limite $J = \bigcap_{n \geq o} (\lambda X.\ X \Delta F(X))^n (I)$ d'une itération décroissante avec rétrécissement $\Delta$ . Ayant choisi $\Delta$ satisfaisant :

$$\forall X, Y \in \prod_{c \in C} A\ ,\ \forall c \in C,\quad X_c \cap Y_c \subseteq (X \Delta Y)_c \subseteq Y$$

on s'assure que lfp $(F) \subsetneq J \subseteq I$

Ayant choisi $\Delta$ satisfaisant la condition qu'il n'existe pas de chaîne strictement décroissante de la forme $X^\circ, \ldots X^{i+1} = X^i \Delta F(X^i), \ldots$ On s'assure de la convergence.

En pratique, on réduit les temps de calcul en remplaçant la stratégie d'itération de Jacobi par une stratégie d'itération chaotique dont l'idée est de "suivre l'exécution du programme analysé". De plus, les extrapolations $\Delta$ et $\nabla$ ne sont nécessaires que pour les têtes de cycle du graphe de dépendance

du système d'équations.
Ceci fera l'objet d'une étude détaillée au chapitre VI de cette thèse.

Analyse sémantique combinée : descendants des états d'entrée qui sont ascendants des états sortie.

Nous voulons à présent trouver une approximation supérieure post $(t^*)$ $(\emptyset) \wedge$ pre $(t^*)$ $(\Psi)$ c'est-à-dire déterminer en chaque point du programme un sur-ensemble des valeurs des variables que l'on peut obtenir sur un chemin d'exécution quelconque du programme, partant du point d'entrée dans un état satisfaisant la condition d'entrée $\emptyset$ et qui se termine dans un état satisfaisant le condition de sortie $\Psi$ .
Il est proposé ⌜Cousot 78⌝ de considérer la limite d'une suite de la forme

$$p^1 = \text{lfp } (F\ (\emptyset))$$
$$p^2 = \text{lfp } (\lambda\ X.\ P^1 \ \underline{\text{et}}\ B\ (\Psi)\ (X).$$
$$\vdots$$
$$p^{2k+1} = \text{lfp } (\lambda\ X.\ p^{2k} \ \underline{\text{et}}\ F\ (\emptyset)\ (X))$$
$$p^{2k+2} = \text{lfp } (\lambda\ X.\ p^{2b+1} \ \underline{\text{et}}\ B\ (\Psi)\ (X))$$

Cette suite est décroissante et, si elle ne satisfait pas le condition de chaîne descendante, il est possible de forcer la converge au prix d'une perte d'information par un opérateur de rétrécissement :

$$p^1 \supseteq \text{lfp } (F\ (\emptyset))$$
$$p^2 \supseteq p_1 \vartriangle X2 \qquad \text{où } X^2 \supseteq \text{lfp } (\lambda\ X.P^1 \sqcap B\ (\Psi)\ (X))$$
$$\vdots$$
(1)
$$\vdots$$
$$p^{2k+1} \supseteq p^{2k} \vartriangle X^{2k+} \qquad \text{où } X^{2k+1} \supseteq \text{llfp } (\lambda x.P^{2k} \sqcap F\ (\emptyset)\ (X))$$
$$p^{2k+2} \supseteq p^{2k+1} \vartriangle x^{2k+2} \qquad \text{où } X^{2k+2} \supseteq \text{lfp } (XX.\ P^{2k+1} \sqcap B\ (\Psi)\ (X)$$

Pour calculer les $X^k$, on utilise une itération chaotique croissante avec élargissement supérieur, puis, si la solution obtenue n'est pas un point fixe, on l'améliore par une itération chaotique décroissante avec rétrécissement inférieur. Il est clair que la résolution séparée des deux systèmes d'équation X = F (X) et X = ( B (X) fournirait de bien moins bons réultats que la méthode (1).
Le lecteur qui désire de plus amples détails concernant la méthode que nous venons de résumer, est invité à lire ⌜Cousot 78⌝ .

## 2. Représentation intermédiaire des programmes

. Nous allons à présent décrire la forme des programmes qui pourront être analysés. Il est entendu que cette forme syntaxique simple n'est qu'une écriture intermédiaire et que tout programme pourra se transformer rapidement dans cette écriture intermédiaire.

Considérons : les affectations
les conditionnelles
les sauts et les étiquettes

Cette forme rudimentaire nous permet dans un premier stade de nous consacrer exclusivement à la sémantique. Plus tard, nous complèterons cette syntaxe.

. Notre but est l'étude des programmes écrits dans le langage Pascal : pour cela nous allons dans ce chapitre commencer par l'étude des variables entières simples ; dans le paragraphe suivant, nous verrons comment étendre cette étude à celle des tableaux d'entiers. Ultérieurement dans cette thèse, nous consacrerons un chapitre à l'étude sémantique des variables dynamiques à savoir les pointeurs sur le tas. Nous pouvons donner ici une des raisons qui nous ont poussé à choisir le langage Pascal : outre le fait qu'il existe dans ce langage des structures dynamiques ou des procédures récursives, nous avons été influencé par les déclarations des variables qui permettent au programmeur de modifier ou d'améliorer son programme en changeant les valeurs possibles d'une ou plusieurs variables que son programme utilise. Cette propriété nous permettra de constater les différences que vont pouvoir apporter ces modifications des déclarations sur les tests que l'analyse sémantique va fournir à l'exécution (ces tests seront le résultat de comparaison de résultats obtenus sur les différentes variables et seront détaillés plus tard). Nous allons maintenant décrire la forme syntaxique simplifiée que nous allons utiliser dans tous les exemples que nous prendrons.

. Une affectation s'écrira comme suit :

< variable > := < variable > / < constante >

ou

< variable > := < variable > < opérateur > < variable > /
< variable > < opérateur > < constante > /
< constante > < opérateur > < variable > /
< constante > < opérateur > < constante > .

Nous évitons d'accepter les expressions arithmétiques afin de ne pas surcharger l'étude par l'introduction d'arbres qui représenteraient ces expressions.

Les opérateurs arithmétiques qui vont être traités seront : +, -, * , / (division entière notée div en Pascal) et la fonction modulo (notée mod en Pascal).

. Une conditionnelle s'écrira comme suit :

if condition then goto étiquette ;

ou < condition > = < variable > < opérateur logique > < variable > /
< variable > < opérateur logique > < constante > /
< constante > < opérateur logique > < variable > /
< constante > < opérateur logique > < constante > .

Comme dans le cas des affectations, nous ne considèrerons pas les conditions composées afin de ne pas avoir à traiter les arbres qui représenteraient ces conditions. Notons que la difficulté est d'ordre pratique et non théorique.

L'opérateur logique pourra prendre les valeurs : $=, \neq, \leq, <, \geq, >$

L'étiquette est un entier déclaré dans le paragraphe label au début du programme.

. Un saut s'écrira naturellement goto < étiquette > ;

3. Informations à déterminer pour les variables simples

Il nous reste à définir le type d'informations à propager à travers ces instructions du graphe de dépendance du programme, et aussi de quelle manière les propager.

Les informations sont multiples, même si l'on considère les variables entières simples comme seules variables du programme.

Il est difficile, voire impossible, pour un compilateur actuel de savoir si une variable a été initialisée : en effet, si l'on utilise une variable sans lui avoir précédemment affecté de valeur, la variable a alors pour valeur une valeur résiduelle de la mémoire ; ceci ne conduit pas à l'abandon de l'exécution par le compilateur mais conduit assez sûrement à une erreur qui bien souvent n'est même pas détectée par les tests mis à l'exécution par le compilateur. Pour pallier à cette carence, nous allons nous intéresser à l'initialisation des variables d'un programme.

La seconde propriété des variables qui nous préoccupe est de connaître (exactement ou en approximation) les valeur de ces variables ; or la meilleure façon de faire une approximation des valeurs à l'exécution est de les encadrer. C'est la raison pour laquelle il paraît naturel de chercher à découvrir lors d'une analyse sémantique les valeurs possibles des variables d'un programme.

Ce sont donc ces deux aspects des propriétés des variables (initialisation et valeurs possibles) qui vont être traités tout au long des prochains chapitres.

## 4. Sémantique opérationnelle

Définissons à présent la sémantique opérationnelle du langage que nous allons utiliser. Nous précisons que ce langage simple n'est que la représentation interne que nous donnons à tout programme étudié.
Le graphe de dépendance associé au programme sera supposé connexe et n'aura qu'un seul point d'entrée et un seul point de sortie.

Soit un programme P :

soit $x_k$, $k \in K$ les variables de P

$a_i$, $i \in [1,n]$ les points de P

La sémantique opérationnelle doit impérativement tenir compte de l'initialisation et des noeuds de jonction comme suit :

- on définit un état $e \in E$ comme un couple (c, m) où
  $c \in C = \{ a_1, \ldots a_n, \text{erreur} \}$ est l'état de contrôle, $m \in M^n = (D \times \{i, \neg i\})^n$ est l'état mémoire, i est la marque d'initialisation,
  $\neg i$ celle de non initialisation.

- la fonction partielle $\bar{t}$ de transition induite par le programme définit la relation de transition $t(e,e') = [e \in d(f) \wedge é = \bar{t}(e)]$
  où d(f) est le domaine de définition de f.
  Elle est définie par cas comme suit :

  . affectation

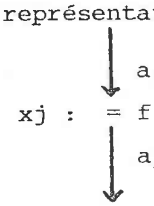
  si $m = ((d_1, i_1), \ldots (dn, in)) \in M^P$, on note $dj = \underline{val}\ (mj)$
  et $ij = \underline{init}\ (mj)$

  la syntaxe de l'affectation est :

  $a_1$ :
  $xj := f\ (x_k, k \in K)$

  $a_2$ :

  la représentation graphique correspondante est :



sémantique : $\bar{t}(a., m) =$ si $\forall k \in K$, init $(m_k) = i_\wedge$ $(val(m_k), k \in K) \in d(f)$

alors

$(a_2, m[j/(f(val(m_k), k \in K), i)])$

sinon

(erreur, m)

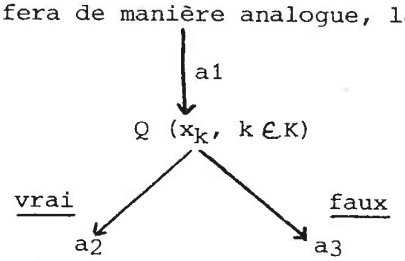
fsi

où $m[j/v]_k = m_k$ pour $k \in [1, n] - [j]$

$m[j/v]j = v$

. test

il se fera de manière analogue, la représentation schématique est :




sémantique :

$\bar{t}(a_1, m) =$ si $\forall k \in K$, init $(mk) = i_\wedge$ $(val(mk), k \in K) \in d(Q)$

alors

si $Q(val(m_k), k \in K)$ alors

$(a_2, m)$

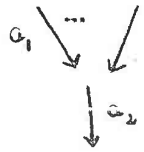
sinon $(a_3, m)$

fsi

sinon (erreur, m)

fsi

. jonction de chemins



la représentation schématique est :

la fonction partielle est définie

par $\bar{t}(a, m) = (a_2, m)$

- dans l'état initial, aucune variable du programme n'est initialisée

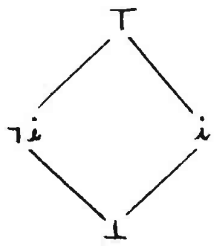
$\emptyset = \lambda(c,m).[c=a_e \wedge \forall k \in [1,n],$ init $(mk) = \neg i$

où $a_e$ l'unique point d'entrée du programme

## 5. Initialisation

### a) Treillis de propriétés

Pour étudier l'initialisation des variables d'un programme P, il faut déterminer en chaque point du programme si des valeurs ont été affectées aux différentes variables de P. Pour cela, nous construisons le treillis L suivant :




pour définir le sens de cette approximation, on définit la fonction de concrétisation :

$$\gamma_L \in L \longrightarrow (D \times \{\neg i, i\} \longrightarrow B)$$

$$\gamma_L(\bot) = (val, ini).\ \underline{false}$$

$$\gamma_L(i) = (val, ini).\ (ini = i)$$

$$\gamma_L(\neg i) = (val, ini).\ (ini = \neg i)$$

$$\gamma_L(T) = (val, ini).\ true$$

$$A = (L^n - \{ (v, \ldots v_n) / \exists j \in [1,n],\ vj = \bot \}) \cup (\bot, \ldots, \bot) ;$$

en chaque point du programme, on associe la valeur abstraite vj à la variable Xj ; l'assertion "faux" a une représentation unique par le p-uplet $(\bot, \ldots, \bot) = \lambda j.\ \bot$

$$\gamma_A \in (A \longrightarrow ((DX\{i, \neg i\})^n \longrightarrow B))$$

$$\gamma_A = \lambda P.\ \lambda m.\ [\forall j \in [1,n],\ \gamma_L(P(j))\ \ (mj)] ;$$

le sens de l'assertion associée à chaque point du programme est la conjonction des assertions sur la marque d'initialisation des variables

$$\gamma \in (\prod_{c \in C} A \longrightarrow (E \longrightarrow \{tt, ff\}))$$

$$\gamma = \lambda I.\ \lambda (c,m).\ [\gamma_A\ (I_c)\ \ (m)] ;$$

quand le contrôle est au point c du programme, l'état mémoire satisfait l'assertion $I_c$ associée à ce point.

La fonction d'abstraction est définie comme suit :

$$\alpha \in (E \longrightarrow B) \longrightarrow \prod_{c \in C} A$$

$$\alpha = \lambda I.\ \lambda c.\ [\alpha_A\ (\lambda m.\ [I\ (c,m)])] ;$$

définissons la projection

$$\pi j \in ((D \times \{ i, \neg i \})^{n} \longrightarrow B) \longrightarrow (D \times \{i, \neg i\} \longrightarrow B))$$

pour $j \in [1,p]$ par

$$\pi j = \lambda\ P.\ \lambda\ (val, ini).\ (\exists\ m \in (D \times \{ i, \neg i \})^{P}\ ,\ P\ (m[j/(val,ini)]).$$

$$\alpha_A \in (((D \times \{ i, \neg i \})^{n} \longrightarrow B) \longrightarrow A)$$

$$\alpha_A = \lambda\ P\ [\ \underline{si}\ \exists\ j \in [1,n]\ ,\ \alpha_L(\pi j\ (P)) = \bot\ \underline{alors}\ \lambda j.\ \bot$$

$$\underline{sinon}\qquad \lambda\ j.[\alpha_L(\pi j(P))]$$

$$\underline{fsi}\ .$$

Les relations entre variables sont ignorées.

$$\alpha_L \in ((D \times \{ i, \neg i \}) \longrightarrow B) \longrightarrow L$$

$$\alpha_L = \lambda\ P.\ [\ \underline{si}\ \forall\ val \in D,\ \forall\ ini \in \{ i, \neg i \},\ \neg\ P\ (val, ini)\ \underline{alors}\ \bot$$

$$\underline{sinonsi}\ \forall val \in D,\ \forall\ ini \in \{ i, \neg i \},\ P(val, ini) \Rightarrow (ini = i)\ \underline{alors}\ i$$

$$\underline{sinonsi}\ \forall val \in D,\ \forall\ ini \in \{ i, \neg i \},\ P(val, ini) \Rightarrow (ini = ri)\ \underline{alors}\ \neg i$$

$$\underline{sinon}\qquad T$$

$$\underline{fsi}\ .$$

La seule information retenue sur les valeurs possibles d'une variable concerne sa marque d'initialisation.

Le treillis L est muni des opérations $\cup, \cap, \nabla, \Delta$ définies précédemment.

b) <u>détermination des régles de construction du système d'équations en avant</u>

Nous cherchons à déterminer l'initialisation des variables au cours d'une exécution d'un programme P commençant dans un état satisfaisant $\emptyset$ ; pour cela établissons formellement le système d'équation à partir de la sémantique opérationnelle.

Soit $a_e$ le point d'entrée du programme ;

on choisit $\emptyset = \lambda\ (c,m).\ [\ c = a_e\ \wedge\ \forall j \in [1,n]\ ,\ \underline{init}\ (m) = \neg i\ ]$ ;

le système d'équation à la forme :

$$X = F\ (X)$$

c'est-à-dire :

$$X = \alpha\ (\emptyset\ v\ post\ (t)\ (\gamma\ (X)))$$

$$= \alpha\ (\emptyset)\ \sqcup\ \alpha\ (post\ (t)\ (\gamma\ (X)))\ ;$$

il vient

$$\alpha(\emptyset) = \lambda c.\ \alpha_A\ (\lambda m.\ [\emptyset\ (c,m)])$$
$$= \lambda c.\ \underline{si}\ c = a_e\ \underline{alors}\ \lambda j.\ \alpha_\perp\ (\lambda(val, ini).\ [ini = \neg i])$$
$$\underline{sinon}\quad \lambda j.\ \perp\ \underline{fsi}$$
$$= \lambda c.\ \underline{si}\ c = a_e\ \underline{alors}\ \lambda j.\ \neg i\ \underline{sinon}\ \lambda j.\perp\ ;$$

il reste maintenant à simplifier le terme $\alpha$ (post (t) ($\gamma$(X)))

$$\alpha(\text{post}(t)(\gamma(X))) = \alpha(\lambda(c,m).[\exists (c',m') \in E,\ \gamma(X)\ (c',m') \wedge t((c'm'), (c,m))])$$
$$= \lambda c.\ \alpha_A\ (\lambda m.\ [\exists (c',m') \in E,\ \gamma\ (X)\ (c',m') \wedge t((c',m'), (c,m))])\ ;$$

pour aller plus loin, on procède par cas, selon la nature du point de contrôle C :

. Point d'entrée du programme

si $c = a_e$, alors les états d'entrée n'ayant pas de prédécesseur $\forall c', m', m\ \neg t((c', m'),(c,m))$, le terme se réduit à

$\alpha_A$ ( m. false $= \perp$

l'équation correspondant au point d'entrée est donc :

$X_{ae} = \lambda j.\ (\neg i)$

. Affectation :

$C = a_2$ et $\downarrow a_1$

$xj = f\ (X_k\ ,\ k\ K)$

$\downarrow a_2$

$$\alpha_A\ (\lambda m.\ [\exists m',\ \gamma(X)\ (a1,m') \wedge \forall k \in K,\ \underline{init}\ (m'_k) = i \wedge (\underline{val}\ (m_k),\ k \in K) \in d(f) \wedge m = m'\ [j\ /\ (f(\underline{val}\ (m_k),\ k \in K),\ i)]])$$

$$= \alpha_A\ (\lambda m.[\exists (val,ini),\ \gamma_A\ (X_{a1})\ (m[j\ /\ (val, ini)]) \wedge \forall k \in K.\ \underline{init}\ (m\ [j/\ (val, ini)]_k) = i \wedge (val\ (m[j/\ (val, ini)]_k), k \in K) \in d(f) \wedge mj = (f\ (\underline{val}\ (m[j/(val, ini)]\ k, k \in K, i))])$$

$$= \alpha_A\ (\lambda m.[\exists\ (val,ini),\ \forall\ k \in (1,n) - \{j\},\ \gamma_\perp(Xa_a\ (k))\ (mk)$$
$$\wedge\ \gamma_A\ (X_{a1}\ (j))\ (ini)$$
$$\wedge \forall k \in K - \{j\},\ \underline{init}\ (mk) = i$$
$$\wedge\ \underline{si}\ j \in K\ \underline{alors}\ \underline{init}\ (ini) = i$$

$$\wedge\ (\underline{val}\ (m\,]j\ /\ (val, ini)]\ k\ ),\ k \in K) \in d\ (f)$$

$$\wedge\ mj = (f\ (\underline{val}\ (m\,[j/\ (val, ini)]\ k),\ k \in K),\ i\ ))]\ )$$

* si $\exists\ k \in [1,n]$ , $Xa_1\ (k) = \perp$ ou $\exists\ k \in K$, $X_{a1}\ (k) = \neg\ i$
ou si l'opérateur f a un domaine de définition vide,
alors le terme se réduit à $\alpha_A$ ( $\lambda$ m. false $= \lambda j. \perp$

* sinon, on remarque que l'on peut choisir de manière équivalente
ini = i et il vient :

$$\alpha_A\ (\lambda\, m.[\ \exists\ val, \forall\ k \in (1,n) - \{ j \}\ ,\ \gamma_L (Xa_1\ (k)\ (mk)$$

$$\wedge\ \forall\ k \in KV\{ j \}\ \underline{init}\ (mk) = i$$

$$\wedge\ (\underline{val}\ (m\,[j/\ (val, i)]\ k,\ k \in K) \in d\ (f)$$

$$\wedge\ (\underline{val}\ (mj) = f\ (\underline{val}\ (m\ [\ j/\ (val, i)]\ k), k \in K)\ ).$$

Comme $\alpha_A$ est isotone, ce terme est inférieur à :

$$\alpha_A\ (\lambda\ m.[\ \forall\ k \in [1,n] - \{ j \}\ ,\ \gamma_L\ (X_{a1}\ (k))\ (mk)$$

$$\wedge\ \forall\ k \in KU\{ j \}\ ,\ \underline{init}\ (m_k) = i\ ])$$

$$= \lambda\, 1.[\alpha_L\ (\lambda\ (val, ini).[\ \exists\ m, \forall\ k \in [1,n] - \{ j \} - K,$$

$$\gamma_L\ (X_{a1}\ (k))\ (m\,[1/(val,ini)\ )]_k,$$

$$\wedge\ \forall\ k \in KU\{ j \}\ ,\ \gamma_L\ (X_{a1}(k) \sqcap i)$$

$$(m\,[1/(val, ini)]_k)$$

$$= \lambda 1.\ \ \underline{si}\ l \in KU\{ j \}\ \underline{alors}\ \alpha_L (\gamma_L\ (X_{a1}\ (1) \sqcap i))\ \underline{sinon}\ \alpha_L\ (\gamma_L (x_{a1}\ (1))).$$

Comme dans le cas considéré, on a $X_{a1}\ (1) \sqsupseteq i$, il vient :

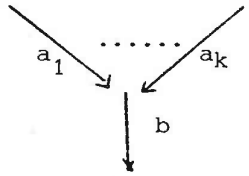
$\lambda$ 1. si $1 \in KU\{ j \}$ alors i sinon $X_{a1}\ (1)$

. Test

Le test se présente sous la forme schématique de deux branches et pour chacune le développement fait ci-dessus est identique.

. Jonction de chemin

La représentation schématique est la suivante :




$$\alpha_A\ (\lambda\ m.[\exists (c',m') \in E\ ,\ \gamma(X)\ (c'm')\ \mathbf{A}\ t\ ((c',m'),(b,m))\ .$$

$$= \alpha_A\ (\lambda\ m.[\exists\ 1 \in [1,k]\ ,\ \gamma(X)\ (a_e,\ m)])$$

$$= \alpha_A\ (\lambda\ m.[\exists\ 1 \in [1,k]\ ,\ \gamma_A\ (X_{ae}\ (m)])$$

$$= \alpha_A\ (\overset{k}{\underset{l\ =\ 1}{\forall}}\ \gamma_A\ (X_{ae}))$$

$$= \bigsqcup_{l=1}^{k} (\alpha_A\ (\gamma_A\ (X_{ae}))\quad \text{car } \alpha_A \text{ est un morphisme complet pour l'union}$$

$$= \bigsqcup_{l=1}^{k} X_{ae} \qquad \text{car } \alpha_A \circ \gamma_A = \mathbb{1}$$

L'équation correspondant au cas où c = erreur ne sert à rien car aucune autre n'en dépend ;
elle donnerait un renseignement global sur la présence d'une erreur mais on préfère les renseignements locaux qui sont donnés de manière plus précise par l'algorithme d'insertion des tests à l'exécution.

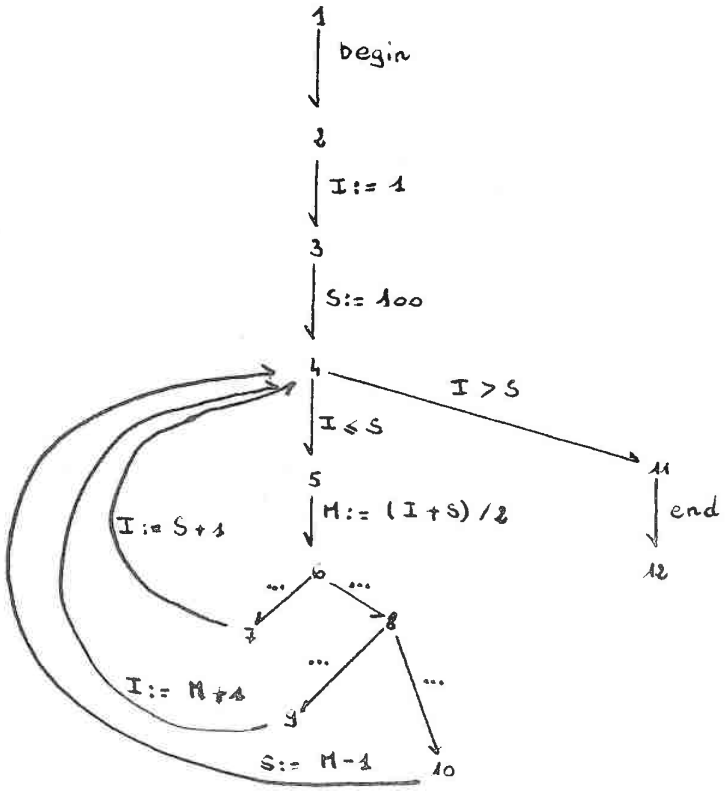
c) Résolution itérative du système d'équation en avant

Donnons maintenant l'exemple de la procédure de recherche dichotomique d'une clé dans une table comportant n éléments dont les clés sont rangés par ordre croissant. Fixons n à 100 et écrivons le programme dans lequel nous aurons préalablement retiré les instructions concernant le tableau (ces instructions ne nous apportent aucune information dans le cas qui nous intéresse et de toutes façons un paragraphe sera consacré plus loin sur les tableaux en particulier)

```
program  dichotomie ;
var      I, S, M : integer ;
begin
         I : = 1 ;
         S : = 100 ;
```

```
        while    I ⩽ S   do
        begin
              M : = (I + S) div 2 ;
              if  ...  then   I : = S + 1
              else if  ...  then  S : = M - 1
              else                I : = M + 1 ;
        end
end.
```

Les instructions remplacées par ... correspondent à des comparaisons d'éléments du tableau considéré.
Décrivons tout d'abord le graphe de dépendance de ce programme.




Les règles que nous venons d'établir permettent d'associer automatiquement le système d'équations suivant à ce programme :

P1 = (I : ⊤ i, S : ⊤ i, M : ⊤ i)
P2 = P1
P3 = P2 (I⟵i)
P4 = P3 (S⟵i) U P6 (I⟵i, S⟵i) U P8 (I⟵i, M⟵i)
U P9 (S⟵i, M⟵i)

P5 = P4 (I⟵i, S⟵i)

P6 = P5 (M⟵i, I⟵i, S⟵i)

P7 = P6

P8 = P6

P9 = P8

P10 = P8

P11 = P4 (I⟵i, S⟵i)

P12 = P11

Comme le treillis L est fini, le calcul par approximations successives du plus petit point fixe lfp $(F_p)$ convergera en un nombre fini de pas.

Intuitivement, l'itération parcourt ce graphe en suivant l'ordre naturel, soit à dire que l'analyse de l'arc 4 ⟶ 11 n'aura lieu que l'orsque les informations qui arriveront au noeud 4 seront complètes.

Au départ, la spécification d'entrée Ø est qu'aucune variable n'est initialisée ; toute utilisation dans une affectation ou un test d'une variable non initialisée conduit à une erreur sémantique qui sera représentée par l'infimum du treillis à savoir $\perp$ .

Le résultat final est le suivant :

| pas \ var | I | S | M |
|---|---|---|---|
| 1 | ⊤i | ⊤i | ⊤i |
| 2 | ⊤i | ⊤i | ⊤i |
| 3 | i | ⊤i | ⊤i |
| 4 | i | i | T |
| 5 | i | i | T |
| 6 | i | i | i |
| 7 | i | i | i |
| 8 | i | i | i |
| 9 | i | i | i |
| 10 | i | i | i |
| 11 | i | i | T |
| 12 | i | i | T |

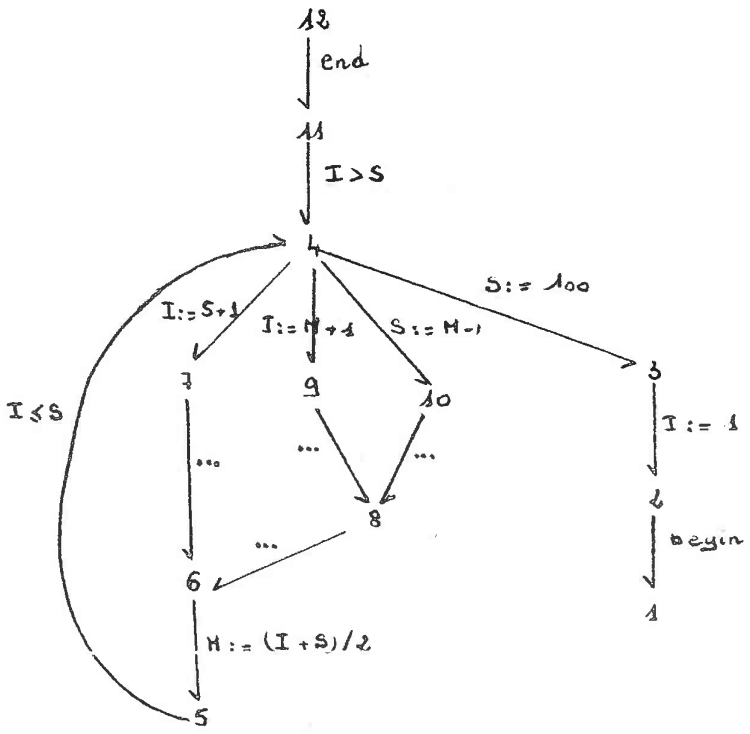
A la sortie de la boucle, l'analyse affirme que les variables I et S sont bien initialisées mais ne peut rien dire sur l'initialisation de la variable M. En effet, comme on fait une analyse statique du problème, il est impos-

sible de savoir si l'arc 4 ⟶ 11 a été traversé alors qu'aucun tour de boucle n'a été effectué (ce qui conduit la variable M à n'avoir aucune valeur), ou bien si un tour de boucle a été fait (ce qui a initialisé M au moins une fois) : nous résumons ceci par le supremum du treillis L : T

Introduisons volontairement une erreur dans le programme en oubliant d'initialiser la variable I par exemple. Le test au noeud 4 utilise la variable I, ce qui conduit à une erreur dans tous les points de programme de 4 à 12. Si nous introduisons la même faute sur la variable M, il sera moins aisé de repérer l'erreur : mais les tests aux noeuds 6 et 8 utilisent M : nous verrons donc dans le paragraphe consacré aux tableaux comment cette erreur sera repérée.

d) Résolution itérative du système d'équation en arrière

Le graphe de dépendance associé au même programme est le suivant :




Des règles similaires aux règles établies pour le système d'équations en avant permettent d'associer le système d'équation approché en arrière :

P12 = $\Psi$

P11 = P12

P10 = P4 (M$\leftarrow$i, S$\leftarrow$i)

P9 = P4 (M$\leftarrow$i, I$\leftarrow$i)

P8 = P9 U P10

P7 = P4 (S$\leftarrow$i, I$\leftarrow$i)

P6 = P7 u P8

P5 = P6 (I$\leftarrow$i, M$\leftarrow$i, S$\leftarrow$i)

P4 = P11 (I$\leftarrow$i, S$\leftarrow$i)U P5 (I$\leftarrow$i, S$\leftarrow$i)

P3 = P4 (S$\leftarrow$T)

P2 = P3 (I$\leftarrow$T)

P1 = P2

Comme précédemment, le treillis L étant fini, le calcul par approximations successives du plus petit point fixe lfp (Bp) convergera en un nombre fini de pas.

La spécification de sortie du système d'équations en arrière sera choisie telle qu'on ne sache rien sur aucune variable, ie pour toute variable v, v = T

Le résultat final est le suivant :

| pas \ var | I | S | M |
|---|---|---|---|
| 1 | T | T | T |
| 2 | T | T | T |
| 3 | i | T | T |
| 4 | i | i | T |
| 5 | i | i | T |
| 6 | i | i | i |
| 7 | i | i | i |
| 8 | i | i | i |
| 9 | i | i | i |
| 10 | i | i | i |
| 11 | T | T | T |
| 12 | T | T | T |

Les résultats sont qualitativement différents si l'on compare les analyses en avant et en arrière. Mais on remarque toutefois que, lors de l'analyse en arrière, nous avons supposé que la spécification de sortie $\Psi$ était

choisie de telle sorte qu'on ne savait rien sur les variables ; toutefois, l'analyse en avant a apporté des informations qu'il est dommage de ne pas utiliser. On peut donc prendre pour spécification de sortie $\Psi$ les résultats obtenus par l'analyse en avant. Partant de ceci, l'analyse en arrière fournit des résultats qui coïncident avec ceux obtenus par l'analyse en avant ; ceci prouve que l'analyse en arrière n'apporte dans ce cas rien de plus que l'analyse en avant ; mais nous verrons tout au long de cet exposé des exemples pour lesquels l'analyse en arrière apporte des renseignements sur l'état des variables qui se révèleront indispensables.

Une remarque simple mais importante : dans cet exemple, chaque fois qu'une variable est utilisée en membre droit, on est assuré qu'elle est initialisée.

Il serait intéressant de combiner les résultats obtenus par l'analyse en avant et par l'analyse en arrière ; en effet, dans l'analyse en arrière, nous prenons comme spécification $\Psi$ le fait qu'on ne sait rien sur les variables du programme : mais l'analyse en avant a apporté une information qui peut être très facilement utilisée. Comme spécification $\Psi$, nous allons donc prendre les résultats de l'analyse en avant.

La combinaison des deux analyses a été explicité en III.1.

Pour le programme présent, les résultats seront donnés par les tableaux suivants :

$P^1 \supseteq$ lfp $(F(\emptyset))$

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| I | ⊤i | ⊤i | i | i | i | i | i | i | i | i | i | i |
| S | ⊤i | ⊤i | ⊤i | i | i | i | i | i | i | i | i | i |
| M | ⊤i | ⊤i | ⊤i | T | T | i | i | i | i | i | T | T |

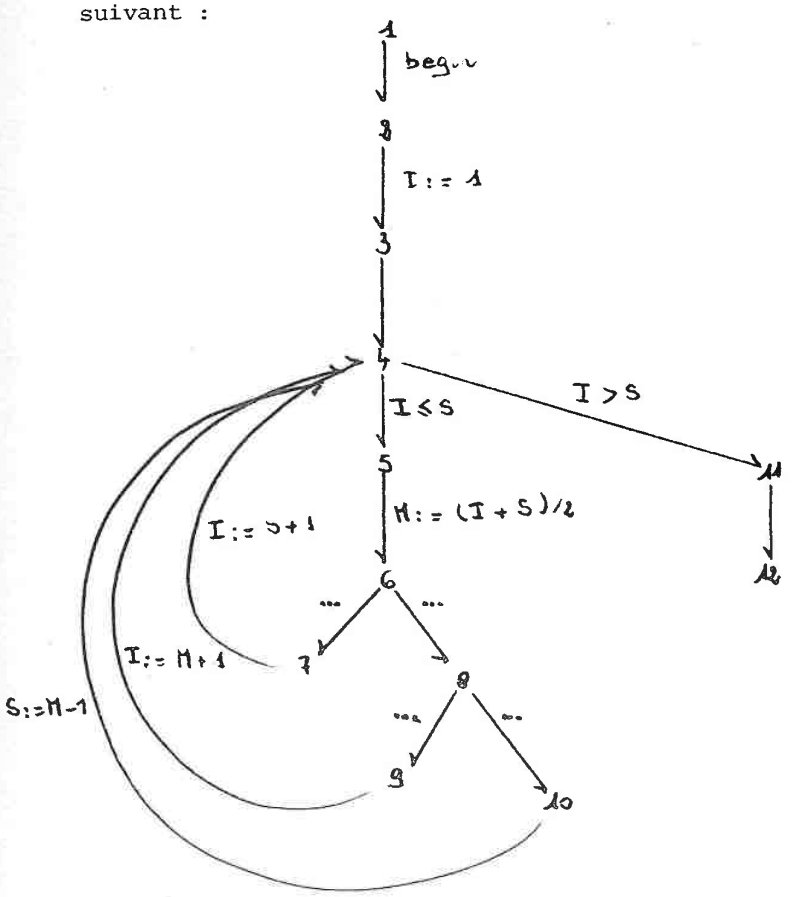
$P^2 = P^1 \Delta X^2$ où $X^2 \supseteq$ lfp $(\lambda X.\ P^1 \cap B(\Psi)\ (X))$

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| I | ⊤i | ⊤i | i | i | i | i | i | i | i | i | i | i |
| S | ⊤i | ⊤i | ⊤i | i | i | i | i | i | i | i | i | i |
| M | ⊤i | ⊤i | ⊤i | T | T | i | i | i | i | i | T | T |

$P^3 = P^2 \Delta X^3$ où $X^3 \supseteq$ lfp $(\lambda X.P^2 \cap F\ (\emptyset)\ (X))$

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| I | ⊤i | ⊤i | i | i | i | i | i | i | i | i | i | i |
| S | ⊤i | ⊤i | ⊤i | i | i | i | i | i | i | i | i | i |
| M | ⊤i | ⊤i | ⊤i | T | T | i | i | i | i | i | T | T |

Dans cet exemple précis, l'analyse en arrière n'apporte aucune information mais elle pourrait imposer la présence d'un test à l'exécution si la variable M était utilisée à partir du noeud 11 ; si une erreur est détectée par l'analyse en avant, l'analyse en arrière permet de déplacer le test d'arrêt de façon à arrêter le programme le plus tôt possible comme le montre l'exemple modifié suivant :




| P1:: | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| I | ri | ri | i | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ |
| S | ri | ri | ri | ri | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ |
| M | ri | ri | ri | ri | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ |

| P2:: | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| I | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ |
| S | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ |
| M | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ |

L'analyse en arrière affirme que ce programme s'il commence, conduira fatalement à une erreur.

Nous verrons, dans le chapitre consacré aux exemples, que tous ne sont pas aussi triviaux que celui-ci.

## 6. Analyse des propriétés d'initialisation et intervalles de valeurs

### a) Treillis des propriétés

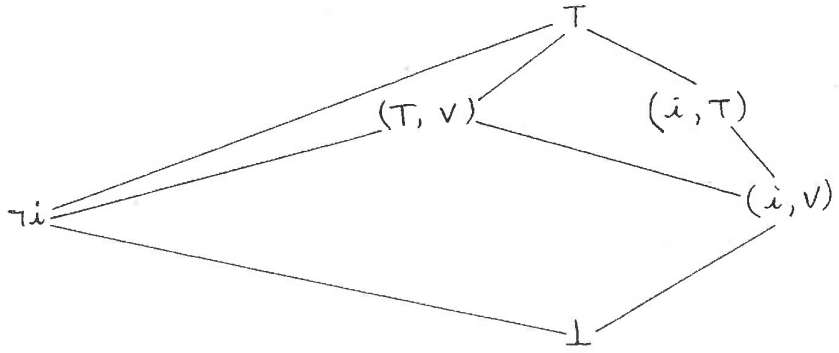
Nous rappelons que P. Cousot et R. cousot ont proposé de choisir comme treillis des valeurs : $L = \{\bot\} \cup \{ [a,b] \ / -\infty \leq a \leq b \leq +\infty \}$

où $-\infty$ ($+\infty$) est le plus petit (grand) entier représentable en machine avec l'interprétation :

$$\gamma_L(\bot) = \lambda\,(val,ini).\ \underline{false}$$
$$\gamma_L(a,b) = \lambda\,(val,ini).\ [\ a \leq val \leq b\ ]\ ;$$

plutôt que d'étudier les propriétés de bonne initialisation et l'intervalle de valeurs des variables séparément, il a été prouvé (Cousot [78]) que les deux analyses séparées apportent de moins bons résultats que ceux obtenus en utilisant une unique analyse, que ce soit sur le plan de la rapidité des calculs ou sur le plan de la précision de ces résultats.

Le treillis sera obtenu en prenant la réunion de tous les éléments des deux treillis correspondant à l'initialisation et à l'intervalle numérique ; on construit la plus petite famille de Moore contenant ces éléments et en rajoutant toutes les intersections qui ne sont pas dans l'ensemble. Le treillis obtenu sera bien évidemment infini et il sera difficile de le représenter schématiquement ; prenons l'exemple d'un seul intervalle V du treillis des intervalles numériques et essayons de construire le sous-ensemble fini du treillis composé des ensembles suivants : $\{\bot, i, \neg i, T\}$ et $\{\bot, V, T\}$



Il suffit de le généraliser pour tout intervalle V
Le supremum T correspond au cas (T,T) c'est-à-dire que l'initialisation est indéterminée et que l'intervalle numérique est l'ensemble complet des valeurs numériques soit $[-\infty , +\infty]$. Les éléments de ce treillis seront composés d'un couple de valeurs : le premier champ sera une valeur d'initialisation (i, $\top$i T), le second champ sera l'intervalle des valeurs possibles.

On choisit donc :

$$L = (\{\bot, i, \top i, T\} \times \{[a,b] / -\infty \leq a \leq b \leq +\infty\}) / \equiv$$

où $\equiv$ est une relation d'équivalence définie par :

$$(ini_1, int_1) \equiv (ini_2, int_2) \text{ ssi } (ini_1 = ini_2 = \bot)$$
$$\text{ou } (ini_1 = ini_2 = \top i)$$
$$\text{ou } (ini_1 = ini_2 \text{ et } int_1 = int_2)$$

on définit la fonction de concrétisation par :

$$\gamma_L((\bot, [a,b])) = \lambda(val, ini).\ \underline{false}$$
$$\gamma_L((i, [a,b])) = \lambda(val, ini).\ [\ ini = i \wedge a \leq val \leq b\ ]$$
$$\gamma_L((\top i, [a,b])) = \lambda(val, ini).\ [\ ini = \top i\ ]$$
$$\gamma_L((T, [a,b])) = \lambda(val, ini).\ [\ \underline{si}\ ini = i\ \underline{alors}\ a \leq val \leq b\ ]$$

Remarque

L a un supremum (T, $[-\infty, +\infty]$) et $\gamma_L((T, [-\infty, +\infty])) = \lambda(val, ini).\ \underline{true}$

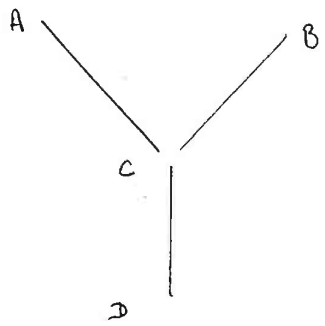
à condition d'avoir choisi dans la sémantique opérationnelle :

$$E = \{a, \ldots ap, \text{erreur}\} \times (D \times \{i, \top i\})^P$$
$$\text{et } D = \{x \in \mathbb{N} / -\infty \leq x \leq +\infty\}.$$

Lorsque la varisable sera initialisée, il importera de connaître V de valeurs numériques possibles ; lorsque, par contre, elle ne sera pas initialisée, les valeurs possibles de la variable seront quelconques, mais surtout d'aucun intérêt, car nous allons considérer que l'utilisation d'une variable qui n'a pas été initialisée constitue un cas d'erreur. Ceci est d'autant plus vrai pour un langage comme Pascal dans lequel une variable à qui aucune valeur n'a été affectée contient une valeur résiduelle de la mémoire qui peut être quelconque : l'utilisation de cette valeur arbitraire entraîne inévitablement des résultats erronés. Donc, lorsque la variable n'a été pas initialisée, aucun intervalle numérique ne peut lui être associé.

Dans le cas où l'initialisation est indéterminée, cela signifie qu'il y a eu jonction de deux chemins du programme, dans l'un, la variable a été initialisée, dans l'autre non. Comme l'analyse sémantique est une analyse statique, il n'est pas possible de savoir quel chemin a été emprunté, donc il n'est pas possible de vérifier l'initialisation de la variable ; mais dans ce cas, il est astucieux de conserver l'intervalle numérique de la variable dans le (ou les) chemin(s) où elle est initialisée.

Par ex.




sur Ac, la variable x n'est pas initialisée et n'a pas d'intervalle numérique significatif

sur Bc, la variable x est initialisée et a V pour intervalle nuémrique

alors posons : sur CD, on ne sait pas si x est initialisée mais si elle l'a été, alors son intervalle est V,

soit : $(\neg i) \ U \ (i, V) = (T, V)$.

Et l'on peut se servir de l'élément $\perp$ pour marquer les erreurs, quitte à associer une marque spéciale à $\perp$ pour distinguer les différentes erreurs possibles, comme nous le verrons dans le chapitre suivant. Nous pouvons dès à présent citer :

- l'utilisation d'une variable non initialisée
- la comparaison (dans un test) de deux quantités qui ne sont pas comparables
- la division par zéro

De plus il est clair qu'une erreur provenant d'une variable ou d'un groupe de variables induit une erreur au point de programme lui-même.

L'arithmétique peut être réécrite sur J x V où $J = \{ \perp, i, \neg i, T \}$

$V = N \times N = \{ [a, b] / a, b \in N \}$

b) Interprétation abstraite

Nous ne présentons pas ici le détail de la détermination formelle des règles de construction des systèmes d'équations sémantiques approchées pour les intervalles et pour l'initialisation ; intuitivement la résolution itérative des équations peut se comprendre comme une interprétation symbolique du programme sur des valeurs abstraites (ini, [ min, max ] ). Le détail sera trouvé dans Cousot [75]. Nous nous contenterons de donner les opérateurs sur les valeurs abstraites.

Soit x et y deux variables représentées par des couples (j, [ a,b ] )
ou $j \in J$ , $[a,b] \in V$
et soit deux fonctions $\bar{i}$, $\bar{v}$ qui sont respectivement la première projection sur J et la seconde projection sur V.

Addition

$\star$ x $\oplus$ y = cas
. $\bar{i}$ (x) = $\bot$ ou $\bar{i}$ (y) = $\bot$ $\Rightarrow \bot$
. $\bar{i}$ (x) = $\neg$i ou $\bar{i}$ (y) = $\dot{\top}$i $\Rightarrow \bot$
. sinon (i, [ min ($\bar{v}$ (x)) + min ($\bar{v}$ (y)),
max ($\bar{v}$ (x)) + max ($\bar{v}$ (y)) ] )
fcas .

Soustraction

$\star$ x $\ominus$ y = cas
. $\bar{i}$ (x) = $\bot$ ou $\dot{\top}$ (y) = $\bot$ $\Rightarrow \bot$
. $\bar{i}$ (x) = $\dot{\top}$i ou $\bar{\top}$ (y) = $\dot{\top}$i $\Rightarrow \bot$
. sinon (i, [ min ($\bar{v}$ (x)) - max ($\bar{v}$ (y)) ,
max ($\bar{v}$ (x)) - min ($\bar{v}$ (y))] )
fcas .

Multiplication

$\star$ x $\otimes$ y = cas
. $\bar{i}$ (x) = ou $\bar{i}$ (y) =
. $\bar{i}$ (x) = ri ou $\bar{i}$ (y) = ri
. sinon/$\star$ soit bi = min ($\bar{v}$ (x)) ; bs = max ($\bar{v}$ (x)) ;
bj = min ($\bar{v}$ (y)) ; bt = max ($\bar{v}$ (y)) $\star$/
(i, [ inf (bi $\star$ bj, bi $\star$ bt, bs $\star$ bj, bs $\star$ bt),
sup (bi $\star$ bj, bi $\star$ bt, bs $\star$ bj, bs $\star$ bt))]
fcas.

Division

```
* x div y = cas
              . ī (x) = ⊥ ou ī (y) = ⊥      ⟹ ⊥
              . ī (x) = ↑i ou ī (y) = ↑i    ⟹ ⊥
              . sinon/* soit bi = min v̄ (x) ; bs = max v̄ (x) ;
                             bj = min v̄ (y) ; bt = max v̄ (y) */
            cas
              . bj = bt = 0                 ⟹ ⊥
              . bi ⩾ 0° et bj > 0  ⟹ (i, [ bi div bt, bs div bj ] )
              . bi ⩾ 0 et bt < 0   ⟹ (i, [ bs div bt, bi div bj ] )
              . bi ⩾ 0 et 0 ∈ [bj, bt] ⟹ (i, [ - bs, bs ] )
              . bs < 0 et bj ⩾ 0   (i, [ bi div bj, bs div bt ] )
              . bs < 0 et bt < 0   (i, [ bs div bj, bi div bt ] )
              . bs < 0 et 0 ∈ [bj, bt] ⟹ (i, [ bi, - bi ] )
              . 0 ∈ [bi, bs] et bj > 0 ⟹ (i, [ bi div bj,
                                               bi div bt ] )
              . 0 ∈ [bi, bs] et bt < 0 ⟹ (i, [ bs div bt,
                                               bs div bj ] )
              . 0 ∈ [bi, bs] et 0 ∈ [bj, bt] ⟹ (i, [ inf (bi, - bs),
                                                     sup (bs, - bi)]
            fcas
      fcas .
```

Modulo

```
* x mod y = cas
             . ī (x) = ⊥ ou ī (y) = ⊥   ⟹ ⊥
             . ī (x) = ↑i ou ī (y) = ↑i ⟹ ⊥
             . sinon  (i, [ 0, max (v̄ (y)) - 1 ] )

           fcas
```

Les prédicats peuvent s'écrire de façon similaire :

```
* Relation (x ⊖ y) = cas
             . ī(x) = ⊥ ou ī (y) = ⊥   ⟹ ⊥
             . ī(x) = ↑i ou ī (y) = ↑i ⟹ ⊥
             . sinon/* soit bi = min v̄ (x) ; bs = max v̄ (x) ;
                            bj = min v̄ (y) ; bt = max v̄ (y) */
           cas
             . bi > bt   ⟹ ⊥
             . bj > bs   ⟹ ⊥
```

. sinon x : ($\bar{i}$ (x), [ sup (bi, bj) , inf (bs, bt)])
y : ($\bar{i}$ (y), [ sup (bi, bj) , inf (bs, bt)])

fcas

fcas.

* Relation (x $\not\emptyset$ y) = cas

. $\bar{i}$ (x) = $\perp$ ou $\bar{i}$ (y) = $\perp$ $\Rightarrow$ $\perp$
. $\bar{i}$ (x) = $\top i$ ou $\bar{i}$ (xy) = $\top i$ $\Rightarrow$ $\perp$
. sinon/* soit bi = min $\bar{v}$ (x) ; bs = max $\bar{v}$ (x) ;
bj = min $\bar{v}$ (y) ; bt = max $\bar{v}$ (y) * /

cas
. bi = bj = bs = bt $\Rightarrow$ $\perp$
. bi = bj = bs $\Rightarrow$ x : ($\bar{i}$(r), [bi, bs ] )
y : ($\bar{i}$(y), [bj + 1, bt])
. bi = bt = bs $\Rightarrow$ x : ($\bar{i}$(x), [bi, bs] )
y : ($\bar{i}$(y), [bj, bt - 1])
. bi = bj = bt $\Rightarrow$ x : ($\bar{i}$(x), [bi + 1, bs])
y : ($\bar{i}$(y), [bj, bt ] )
. bs = bs = bt $\Rightarrow$ x : ($\bar{i}$(x), [bi, bs - 1] )
y : ($\bar{i}$(y), [bj, bt] )
. sinon x : ($\bar{i}$ (x), [bi, bs] )
y : ($\bar{i}$ (y), [bj, bt ] )

* Relation (x $\leq$ y) = cas

. $\bar{i}$ (x) = $\perp$ ou $\bar{i}$ (y) = $\perp$ $\Rightarrow$ $\perp$
. $\bar{i}$ (x) = $\top i$ ou $\bar{i}$ (y) = $\top i$ $\Rightarrow$ $\perp$
. sinon /* soit bi = min ($\bar{v}$ (x) ; bs = max $\bar{v}$ (x) ;
bj = min ($\bar{v}$ (y) : bt = max $\bar{v}$ (y) */

cas
. bi > bt $\Rightarrow$ $\perp$
. sinon x : ($\bar{i}$ (x), [ bi, inf (bs, bt)])
Y : ($\bar{i}$ (y), [ sup (bi, bj), bt ] )

fcas

fcas.

* Relation (x $\mathcal{R}$ y) = cas

. $\bar{i}$ (x) = $\perp$ ou $\bar{i}$ (y) = $\perp$ $\Rightarrow$ $\perp$

. $\bar{i}$ (x) = ri ou $\bar{i}$ (y) = ri $\Rightarrow$ $\perp$

. sinon /* soit bi = min $\bar{v}$ (x) ; bs = max $\bar{v}$ (x)
bj = min $\bar{v}$ (y) ; bt = max $\bar{v}$ (y) */

cas

. bi $\geqslant$ bt $\Rightarrow$ $\perp$

. sinon x : ($\bar{i}$ (x), [ bi, inf (bs,bt-1 )])
y : ($\bar{i}$ (y), [sup (bi + 1, bj), bt]

fcas

fcas.

L'étude des relations $\geqslant$ et $>$ se fait par analogie.

Pendant l'analyse déductive, nous allons être amenés à considérer la réunion de contextes (chaque contexte étant un produit cartésien des "valeurs" des différentes variables, chaque "valeur" étant elle-même le produit cartésien d'une valeur d'initialisation et d'un intervalle numérique). De même, l'intersection, l'élargissement et le rétrécissement de tels contextes seront à considérer : pour cela, il faudra savoir comment se comporteront ces quatre fonctions avec chaque "valeur" des variables.

* réunion x U y = cas

. $\bar{i}$ (x) = $\perp$ $\Rightarrow$ x

. $\bar{i}$ (y) = $\perp$ $\Rightarrow$ y

. $\bar{i}$ (x) = $\top$i et $\bar{i}$ (y) = $\top$i $\Rightarrow$ $\top$i

. $\bar{i}$ (x) = $\top$i $\Rightarrow$ (T, $\bar{v}$ (y))

. $\bar{i}$ (y) = $\top$i $\Rightarrow$ (T, $\bar{v}$ (x))

. sinon ($\bar{i}$(x) u $\bar{i}$ (y), [ inf (min $\bar{v}$ (x) ,
(min $\bar{v}$ (y))) ,
sup (max $\bar{v}$ (x), max $\bar{v}$ (y)] )

fcas.

* intersection x $\cap$ y = cas

. $\bar{i}$ (x) = $\perp$ ou $\bar{i}$ (y) = $\perp$ $\Rightarrow$ $\perp$

. sinon

cas

. $\bar{i}$ (x) = $\bar{i}$ (y) $\Rightarrow$ $\bar{i}$ = $\bar{i}$ (x)

. $\bar{i}$ (x) = T $\Rightarrow$ $\bar{i}$ = $\bar{i}$ (y)

. $\bar{i}$ (y) = T $\Rightarrow$ $\bar{i}$ = $\bar{i}$ (x)

. sinon $\bar{i} = \perp$

fcas

cas

. $\max \bar{v}(x) < \min \bar{v}(y) \Rightarrow \perp$

. $\min \bar{v}(x) > \max \bar{v}(y) \Rightarrow \perp$

. sinon $\bar{v} = [\sup(\min \bar{v}(x), \min \bar{v}(y)),$
$\inf(\max \bar{v}(x), \max(v\ y))]$

fcas

si $\bar{i} \neq \perp$ et $\bar{i} \neq$ Ti alors $(\bar{i}, \bar{v})$

fcas.

* élargissement $x \nabla y$ = cas

. $\bar{i}(x) = \perp \Rightarrow y$

. $\bar{i}(y) = \perp \Rightarrow x$

. $\bar{i}(x) =$ ri et $\bar{i}(y) =$ ri $\Rightarrow$ Ti

. $\bar{i}(x) =$ ri $\Rightarrow (T, \bar{v}(y)$

. $\bar{i}(y) =$ ri $\Rightarrow (T, \bar{v}(x)$

. sinon $\bar{i} = \bar{i}(x) \cup \bar{i}(y)$

$\min \bar{v} =$ si $\min(\bar{v}(y) < \min \bar{v}(x)$ alors $-\infty$
sinon $\min \bar{v}(x)$

$\max \bar{v} =$ si $\max \bar{v}(y) > \max \bar{v}(x)$ alors $+\infty$
sinon $\max \bar{v}(x)$

$\Rightarrow (\bar{i}, \bar{v})$

fcas.

* rétrécissement $x \Delta y$ = cas

. $\bar{i}(x) = \perp$ ou $\bar{i}(y) = \perp \Rightarrow \perp$

. sinon

cas

. $\bar{i}(x) = \bar{i}(y) \Rightarrow \bar{i} = \bar{i}(x)$

. $\bar{i}(x) = T \Rightarrow \bar{i} = \bar{i}(y)$

. $\bar{i}(y) = T \Rightarrow \bar{i} = \bar{i}(x)$

fcas

cas

. $\max \bar{v}(x) < \min \bar{v}(y) \Rightarrow \perp$

. $\min \bar{v}(x) > \max \bar{v}(y) \Rightarrow \perp$

. sinon

$\min \bar{v} =$ si $\min \bar{v}(x) = -\infty$ alors $\min \bar{v}(y)$
sinon $\inf(\min \bar{v}(x), \min \bar{v}(y))$

$\max \bar{v} =$ si $\max \bar{v}(x) = +\infty$ alors $\max \bar{v}$ (y
sinon $\sup(\max \bar{v}(x), \max \bar{v}(y))$

fcas
si $\bar{i} \neq \perp$ et $\bar{i} \neq \top i$ alors $(\bar{i}, \bar{v})$
fcas.

C. Application à l'exemple

Reprenons le programme de la recherche dichotomique et appliquons l'analyse sémantique déductive en avant à cet exemple. Les résultats vont être qualitativement différents.

Ecrivons les équations sémantiques :

$P1 = \emptyset$

$P2 = P1$

$P3 = P2\ (I \longleftarrow 1)$

$P4 = P3\ (S \longleftarrow 100)\ U\ P7\ (I \longleftarrow S+1)\ U\ P9\ (I \longleftarrow M+1)\ U\ P10\ (S \longleftarrow M-1)$

$P5 = P4\ (I \longleftarrow S)$

$P6 : P5\ (M \longleftarrow \frac{I+S}{2})$

$P7 = P6$

$P8 = P6$

$P9 = P8$

$P10 = P8$

$P11 = P4\ (I \longleftarrow S)$

$P12 = P11$

La spécification d'entrée $\emptyset$ est telle qu'aucune variable n'est initialisée et par conséquent, aucun intervalle ne leur est associé. Au départ de l'analyse, on a $\forall i \geq 2 \quad P_i^0 = \perp$

Les variables I, S, M seront rangées dans cet ordre :

$P1' = \{ \top i, \top i, \top i \}$

$P2' = P1$

$= \{ \top i, \top i, \top i \}$

$P3' = P2\ (I \longleftarrow 1)$

$= \{ (i, [1,1]), \top i, \top i \}$

$P4' = P4^\circ \nabla (P3'\ (S \longleftarrow 100)\ U\ P7^\circ\ (I \longleftarrow S+1)\ U\ P9^\circ\ (I \longleftarrow M+1)$

$U\ P10^\circ\ (S \longleftarrow M-1))$

$= P4^\circ \nabla (P3'\ (S \longleftarrow 100))$

$= \perp \nabla \{(i, [1,1]), (i\ [100, 100]), \top i \}$

$= \{ (i, [1,1]\ ,\ (i, [100, 100]\ , \top i \}$

$P'_5 = P^1_4 \ ( I \leqslant S)$

$= \{ (i, [1, 1] ), (i, [100, 100] ), \forall i \}$

$P^1_6 = P'_5 \ (M \longleftarrow (I + S) / 2)$

$= \{ (i, [1,1] ) , (i, [100, 100]), (i, [50, 50] )$

$P'_7 = P'_8 = P'_9 = P'_{10} = P_6{}^1$

$P^2_4 = P'_4 \nabla (P'_3 \ (S \longleftarrow 100) \ U \ P'_7 \ (I \longleftarrow S + 1) \ U \ P'_9 \ (I \longleftarrow M + 1)$

$U \ P'_{10} (S \longleftarrow M - 1))$

$= \{ (i, [1,1]), (i, [100, 100]), \forall i \} \nabla \{ (i, [1,101] ),$

$(i, [49, 100]), i, [50,50] ) \}$

$= \{ (i, [1, +\infty] ), (i, [-\infty , 100] ), (T, [50,50] ) \}$

$P^2_5 = \{ (i, [1, 100] ), (i, [1, 100] ), (T, [50,50] ) \}$

$P^2_6 = \{ (i, [1, 100]), (i, [1, 100] ), (i, [1, 100 ] ) \}$

$P^2_7 = P^2_8 = P^2_9 = P^2_{10} = P^2_6$

$P^3_4 = P^2_4 \ \nabla \ (P'_3 \ (S \longleftarrow 100) \ u \ P^2_7 \ (I \longleftarrow S + 1) \ U \ P^2_9 \ (I \longleftarrow M + 1)$

$U \ P^2_{10} (S \longleftarrow M - 1))$

$= \{ (i, [1, +\infty]) , (i, [-\infty , 100]), (T, [50,50] ) \}$

$\nabla \{ (i, [1, 101] ), (i, [0,100]), (T, [1,100]$

$= \{ (i, [1, +\infty]), (i, [-\infty, 100]), (T, [-\infty, +\infty] ) \}$

$P^3_5 = \{ (i, [1,100]), (i, [1,100]), (T, [-\infty, +\infty]) \}$

$P^3_6 = \{ (i, [1,100]), (i, [1,100]), (i, [1,100]) \}$

$P^3_7 = P^3_8 = P^3_9 = P^3_{10} = P^3_6$

$P^4_4 \subseteq P^3_4$ donc le système est stable

on peut maintenant calculer :

$P^4_{11} = P^4_4 \ (I \leftrightarrows S)$

$= \{ (i, [1, +\infty]), (i, [-\infty , 100]), (T, [-\infty, +\infty]) \}$

$P^4_{12} = P^4_{11}$

$= \{ (i, [1, +\infty] ), (i, [-\infty , 100] ) , (T , [-\infty , +\infty] ) \}$

A présent, nous allons améliorer les résultats obtenus par une analyse avec rétrécissement.

$P_1^5 = \{ \top i, \top i, \top i \}$

$P_2^5 = \{ \top i, \top i, \top i \}$

$P_3^5 = P_2^5 \ (I \longleftarrow 1)$

$= \{ (i, [1,1]), \top i, \top i \}$

$P_4^5 = P_4^4 \ \Delta \ (P_3^5 \ (S \leftarrow 100) \ u \ P_7^4 \ (I \leftarrow S + 1) \ U \ P_9^4 \ (I \leftarrow M + 1)$

$U \ P_{10}^4 \ (S \leftarrow M - 1))$

$= \{ (i, [1, +\infty]), (i, [-\infty, 100]), (T, [-\infty, +\infty]) \}$

$\Delta \{ (i, [1, 101]), (i, [0,100]), (T, [1,100]) \}$

$= \{ (i, [1, 101]), (i, [0,100]), (T, [1,100) \}$

$P_5^5 = P_4^5 \ (I \leqslant S)$

$= \{ (i, [1,101]), (i,[1,100]), (T,[1,100]) \}$

$P_6^5 = P_5^5 \ (M \longleftarrow (I + S) / 2)$

$= \{ (i, [1,101]), (i, [1,100]), (i, [1,100]) \}$

$P_7^5 = P_8^5 = P_9^5 = P_{10}^5 = P_6^5$

$P_4^6 \supseteq P_4^5$ donc la stabilité est atteinte

$P_{11}^6 = P_4^6 \ (I > S)$

$= \{ (i, [1,101]), (i, [à,100]), (T, [1,100]) \}$

$P_{12}^6 = P_{11}^6$

$= \{ (i, [1,101]), (i, [0,100]), (T, [1,100]) \}$

Plusieurs constatations :

- aucune variable utilisée n'est dans le cas non initialisée (sinon une erreur serait apparue)
- toutes les variables utilisées sont initialisées (si une variable utilisée a une marque d'initialisation à T, cela veut dire qu'en ce point de programme, on sait pas si la variable a été initialisée ; donc, afin de ne pas utiliser cette variable si elle n'a pas été initialisée, il faudra à l'exécution mettre un test sur l'initialisation).

d) Analyse en arrière

L'analyse en arrière recherche les ascendants des états de sortie, c'est-à-dire en chaque point les conditions nécessaires avant l'action pour terminer l'action dans la situation attendue et sans erreur.

ex. pour pouvoir effectuer l'instruction a : = b, il fallait qu'avant, la variable b ait été initialisée ;
l'analyse en avant a déjà étudié les cas où b n'a pas été initialisée (ce qui conduit à une erreur à l'exécution) et où une erreur se propage ;
si la marque d'initialisation de la variable b est T, cela signifie qu'en arrivant à l'instruction, le contrôle ne peut pas savoir si b a été initialisée il faudra donc ajouter avant cette instruction un test qui permettra l'arrêt du programme en cas d'erreur (i.e. en cas de non-initialisation) ;
ce test va être déplacé de façon à arrêter le programme le plus tôt possible.

Il nous faut donc définir la sémantique en arrière de notre langage et en particulier des instructions élémentaires que nous utilisons :

a) initialisation :
pour une affection, toutes les variables qui sont en membre droit doivent avoir été initialisées avant l'affectation ;
pour un test, toutes les variables doivent être initialisées avant le test ;

b) intervalle numérique :
pour une affectation, l'analyse en avant n'apporte aucune information sur les valeurs abstraites des variables du membre droit de l'affectation ;
l'analyse en arrière va pouvoir apporter des informations concernant les variables du membre droit afin que la variable du membre gauche soit telle que l'analyse en avant l'a définie ;
la relation qui va lier les valeurs des différentes variables va dépendre de l'opérateur op défini ;

nous allons les étudier cas par cas ;

pour les tests, nous ne rechercherons aucune information particulière.

Les instructions du type z : = z + x ou z : = z - x sont des cas particuliers de l'addition et de la soustraction ; de même l'instruction z : = x est un cas particulier de l'addition (z : = x + o)

* addition ( z = x ⊞ y) = (x' = z θ y et y' = z θ x et z' quelconque)

soit . $\bar{i}$ (x') = i

$\underline{min}\ \bar{v}$ (x') = $\underline{min}\ \bar{v}$ (z) - $\underline{max}\ \bar{v}$ (y)

$\underline{max}\ \bar{v}$ (x') = $\underline{max}\ \bar{v}$ (z) - $\underline{min}\ \bar{v}$ (y)

. $\bar{i}$ (y') = i

$\underline{min}\ \bar{v}$ (y') = $\underline{min}\ \bar{v}$ (z) - $\underline{max}\ \bar{v}$ (x)

$\underline{max}\ \bar{v}$ (y') = $\underline{max}\ \bar{v}$ (z) - $\underline{min}\ \bar{v}$ (x)

. $\bar{i}$ (z') = T

$\underline{min}\ \bar{v}$ (z') = - ∞

$\underline{max}\ \bar{v}$ (z') = + ∞

* soustraction (z = x ⊟ y) = x' = z θ y et y' = x θ z et z' quelconque)

soit . $\bar{i}$ (x') = i

$\underline{min}\ \bar{v}$ (x') = $\underline{min}\ \bar{v}$ (z) + $\underline{min}\ \bar{v}$ (y)

$\underline{max}\ \bar{v}$ (x') = $\underline{max}\ \bar{v}$ (z) + $\underline{max}\ \bar{v}$ (y)

. $\bar{i}$ (y') = i

$\underline{min}\ \bar{v}$ (y') = $\underline{min}\ \bar{v}$ (x) - $\underline{max}\ \bar{v}$ (z)

$\underline{max}\ \bar{v}$ (y') = $\underline{max}\ \bar{v}$ (x) - $\underline{min}\ \bar{v}$ (z)

. $\bar{i}$ (z') = T

$\underline{min}\ \bar{v}$ (z') = - ∞

$\underline{max}\ \bar{v}$ (z') = + ∞

* multiplication (z = x ⊠ y) = (x' = z div y et y' = z div x et z' quelconque)

soit /* xi = $\underline{min}\ \bar{v}$ (x) ; xs = $\underline{max}\ \bar{v}$ (x) ; yi = $\underline{min}\ \bar{v}$ (y) ; ys = $\underline{max}\ \bar{v}$ (y)
zi = $\underline{min}\ \bar{v}$ (z) ; zs = $\underline{max}\ \bar{v}$ (z) * /

. x' = cas

. yi = ys = o ⇒ ⊥

. zi ≥ 0 et yi > 0 ⇒ (i, [ zi div ys, zs div yi ] )

. zi ≥ 0 et ys < 0 ⇒ (i, [ zs div ys, zi div yi 1 )

. zi ⩾ 0 et 0 ∈ [yi, ys] ⇒ (i, [- zs, zs] )
. zi < 0 et yi > 0 ⇒ (i, [zi div yi, zs div ys])
. zi < 0 et ys < 0 ⇒ (i, [zs div yi, zi div ys])
. zi < 0 et 0 ∈ [yi, ys] ⇒ (i, [ zi, - zi] )
. 0 ∈ [zi, zs] et yi > 0 ⇒ (i, [zi div yi, zi div ys] )
. 0 ∈ [zi, zs] et ys < 0 ⇒ (i, [zs div ys, zs div yi] )
. 0 ∈ [zi, zs] et 0 ∈ [yi, ys] ⇒ (i, [ inf (zi, - zs), sup (zs, - zi) ] )

fcas .
. y' = cas
. xi = xs = 0 ⇒ ⊥
. zi ⩾ 0 et xi > 0 ⇒ (i, [zi div xs, zs div xi] )
. zi ⩾ 0 et xs < 0 ⇒ (i, [zs div xs, zi div xi] )
. zi ⩾ 0 et 0 ∈ [xi, xs] ⇒ (i, [ - zs, zs])
. zi < 0 et xi > 0 ⇒ (i, [zi div xi, zs div xi] )
. zi < 0 et xs < 0 ⇒ (i, [zs div xi, zi div xs] )
. zi < 0 et 0 ∈ [xi, xs] ⇒ (i, [zi, - zi] )
. 0 ∈ [zi, zs] et xi > 0 ⇒ (i, [zi div xi, zi div xs] )
. 0 ∈ [zi, zs] et xs < 0 ⇒ (i, [zs div xs, zs div xi] )
. 0 ∈ [zi, zs] et 0 ∈ [xi, xs] ⇒ (i, inf 'zi, - zs), sup (zs - zi)])

fcas
. ī (z') = T
min v̄ (z') = - ∞
max v̄ (z') = + ∞

* division (z = x div y ) = (x' = z ⊛ y et y' = x div z et z' quelconque)
soit /* xi=min v̄ (x) ; xs = max v̄ (x) ; yi = min v̄ (y) ;
ys=max v̄ (y) ; zi = min v̄ (z) ; zs = max v̄ (z) * /
. ī (x') = i
min v̄ (x') = inf (zi * yi, zi * ys, zs * yi, zs * ys)
max v̄ (x') = sup (zi * yi, zi * ys, zs * yi, zs * ys)

. y' = cas
. zi = zs = 0 ⇒ ⊥
. xi ⩾ 0 et zi > 0 ⇒ (i, [xi div zs, xs div zi])
. xi ⩾ 0 et zs < 0 ⇒ (i, [xs div zs, xi div zi])
. xi ⩾ 0 et 0 ∈ [zi, zs] ⇒ (i, [ - xs, xs] )
. xs < 0 et zi > 0 ⇒ (i, [ xi div zi, xs div zs])

. $xs < 0$ et $zs < 0 \Rightarrow (i, [xs \text{ div } zi, xi \text{ div } zs])$

. $xs < 0$ et $0 \in (zi, zs) \Rightarrow (i, [xi, -xi])$

. $0 \in [xi, xs]$ et $zi > 0 \Rightarrow (i, [xi \text{ div } zi, xi \text{ div } zs])$

. $0 \in [xi, xs]$ et $zs < 0 \Rightarrow (i, [xs \text{ div } zs, xs \text{ div } zi])$

. $0 \in [xi, xs]$ et $0 \in [zi, zs] \Rightarrow (i, [\inf(xi, -xs), \sup(xs, -xi)])$

fcas .

. $\bar{i}(z') = T$

min $\bar{v}(z') = -\infty$

max $\bar{v}(z') = +\infty$

* modulo z = x mod y

les informations apportées par l'analyse en arrière dépendent de la définition de la fonction modulo ; nous avons choisi celle donnée par la norme ISO:AFNOR ;

c'est-à-dire y doit être un nombre supérieur ou égal à 1.

. $\bar{i}(x') = i$ ; $\bar{v}(x') = \bar{v}(x)$

. $\bar{i}(y') = i$ ; min $\bar{v}(y') = 1$ ; max $\bar{v}(y') = +\infty$

. $\bar{i}(z') = T$ ; min $\bar{v}(z') = -\infty$ ; max $\bar{v}(z') = +\infty$

Les valeurs que nous venons de décrire pour x', y' ne sont toutefois pas les meilleurs possibles ; on peut en effet trouver une meilleure approximation des intervalles en signalant que l'analyse en avant ne modifie pas les variables x et y. Cela signifie que d'un autre côté, les variables x et y ont des intervalles après instruction qui sont inclus dans les intervalles avant instruction et vice versa. Donc les intervalles définitifs que nous allons trouver pour les variables x et y sont des intervalles qui sont d'une part inclus dans les intervalles avant instruction, d'autre part inclus dans les nouvelles valeurs que nous venons de décrire dans les pages précédentes.

Appelons x" et y" les valeurs finales ; on a : $x'' = x \cap x'$ et $y'' = y \cap y'$

Les tests s'effectuent plus simplement puisque l'analyse en arrière ne calcule aucune information concernant les valeurs numériques. Nous avons donc : $x'' = (i, \bar{v}(x))$ et $y'' = (i, \bar{v}(y))$.

Remarque

Nous n'avons pas parlé des variables qui ne sont pas initialisées, ni des cas où une erreur apparaît (par la présence de $\perp$).

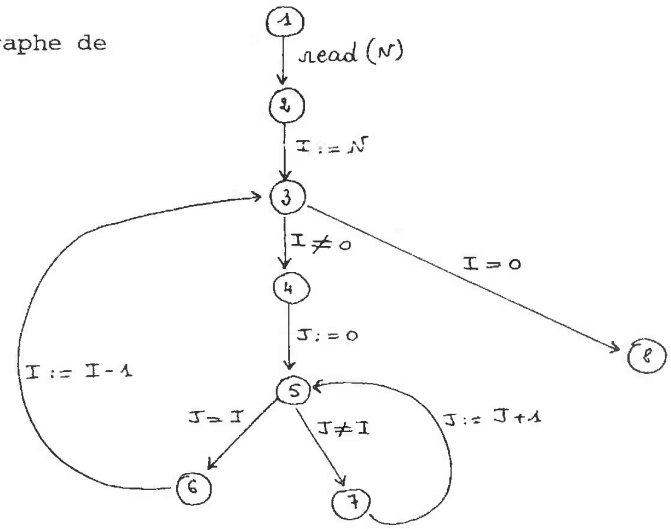
Si une erreur est apparue après une instruction, l'analyse en arrière va chercher à remonter le plus haut possible dans le programme cet indicateur d'erreur afin de pouvoir arrêter l'exécution de ce programme le plus tôt possible.

e. exemple

Nous reprenons ici l'exemple du tri par remontée des bulles donné dans Cousot 78 ; tous les éléments de tableaux et les instructions y faisant référence ont été enlevés du programme :

```
program   tri ;
   :
begin
      read (N) ;
      I : = N ;
      while  I<>0  do
      begin
            J : = 0 ;
            while  J<>I  do   J : = J + 1 ;
            I : = I - 1 ;
      end ;
end .
```

dont voici le graphe de dépendance :



Nous n'avons jusqu'à présent pas parlé de l'instruction de lecture dont la sémantique est simple :

read (x) $\Longrightarrow$ x = (i, $[-\infty , +\infty]$) ;

en effet x est bien initialisé et ses valeurs sont quelconques puisque choisies en lecture (un format, possible en Pascal, apporterait des précisions supplémentaires) l'analyse en arrière ne peut que supposer que x a une initialisation indéterminée avant la lecture.

Les équations sémantiques en avant sont :

P1 = $\emptyset$

P2 = P1 (lire (N))

P3 = P2 (I$\longleftarrow$N) U P6 (I$\longleftarrow$I -1)

P4 = P3 (I <> O)

P5 = P4 (J$\longleftarrow$ O) U P7 (J$\longleftarrow$J + 1)

P6 = P5 (J = I)

P7 = P5 (J <> I)

P8 = P3 (I = O)

Les équations sémantiques en arrière sont :

P8 = $\Psi$

P7 = P5 (J$\longleftarrow$J - 1)

P6 = P3 (I$\longleftarrow$I + 1)

P5 = P6 (J = I) U P7 (J <> I)

P4 = P9 (J$\longleftarrow$T)

P3 = P8 (I = O) U P4 (I <> O)

P2 = P3 (I$\longleftarrow$T)

P1 = P2 (N$\longleftarrow$T)

Nous prendrons comme précédemment . $\emptyset = \{ \top i, \top i, \top i \}$ chaque composante correspondant à I, J et N rangés dans cet ordre

. $\Psi$ = le résultat de l'analyse en avant en P8

L'étude que nous ferons sur le graphe de dépendance dans un chapitre ultérieur donne le noeud 5 comme point où va se faire l'élargissement (et le rétrécissement) ainsi que l'ordre de parcours des ares de ce graphe.

Nous ne donnerons pas le détail de tous les calculs et obtenons :

/ $Q^1 \supseteq$ lfp (F (Ø))

P1 = { $\top$i, $\top$i, $\top$i }

P2 = { $\top$i, $\top$i, (i, [ $-\infty$ , $+\infty$ ]) }

P3 = { (i, [ $-\infty$ , $+\infty$ ]), (T, [ $-\infty$ , $+\infty$ ]), (i, [ $-\infty$ , $+\infty$ ]) }

P4 = { (i, [ $-\infty$ , $+\infty$ ]), (T, [ $-\infty$ , $+\infty$ ]), (i, [ $-\infty$ , $+\infty$ ]) }

P5 = { (i, [ $-\infty$ , $+\infty$ ]), (i, [ $-\infty$ , $+\infty$ ]), (i, [ $-\infty$ , $+\infty$ ]) }

P6 = { (i, [ $-\infty$ , $+\infty$ ]), (i, [ $-\infty$ , $+\infty$ ]), (i, [ $-\infty$ , $+\infty$ ]) }

P7 = { (i, [ $-\infty$ , $+\infty$ ]), (i, [ $-\infty$ , $+\infty$ ]), (i, [ $-\infty$ , $+\infty$ ]) }

P8 = { (i, [ 0,0 ]), (T, [ $-\infty$, $+\infty$ ]), (i, [ $-\infty$ , $+\infty$ ]) }

/ $Q^2 = Q^1 \Delta X^2$ où $X^2 \supseteq$ lfp ($Q^1 \cap$ B ($\Psi$))

P1 = { $\top$i, $\top$i, $\top$i }

P2 = { $\top$i, $\top$i, (i, [0, $+\infty$ ]) }

P3 = { (i, [ 0, $+\infty$ ]), (T, [ $-\infty$ , $+\infty$ ]), (i, [ $-\infty$ , $+\infty$ ]) }

P4 = { (i, [ 0, $+\infty$ ]), (T, [ $-\infty$ , $+\infty$ ]), (i, [ $-\infty$ , $+\infty$ ]) }

P5 = { (i, [ 0, $+\infty$ ]), (i, [ $-\infty$ , $+\infty$ ]), (i, [ $-\infty$ , $+\infty$ ]) }

P6 = { (i, [ 0, $+\infty$ ]), (i, [ $-\infty$ , $+\infty$ ]), (i, [ $-\infty$ , $+\infty$ ]) }

P7 = { (i, [ 0, $+\infty$ ]), (i, [ $-\infty$ , $+\infty$ ]), (i, [ $-\infty$ , $+\infty$ ]) }

P8 = { (i, [0,0 ]), (T, [ $-\infty$ , $+\infty$ ]), (i, [ $-\infty$ , $+\infty$ ]) }

/ $Q^3 = Q^2 \Delta X^3$ où $X^3 \supseteq$ lfp ($Q^2 \cap$ F (Ø))

P1 = { $\top$i, $\top$i, $\top$i }

P2 = { $\top$i, $\top$i, (i, [0, $+\infty$ ]) }

P3 = { (i, [ 0, $+\infty$ ]), (T, [ 1, $+\infty$ ]), (i, [ 0, $+\infty$ ]) }

P4 = { (i, [ 1, $+\infty$ ]), (T, [ 1, $+\infty$ ]), (i, [ 0, $+\infty$ ]) }

P5 = { (i, [ 1, $+\infty$ ]), (i, [ 0, $+\infty$ ]), (i, [ 0, $+\infty$ ]) }

P6 = { (i, [ 1, $+\infty$ ]), (i, [ 1, $+\infty$ ]), (i, [ 0, $+\infty$ ]) }

P7 = { (i, [ 1, $+\infty$ ]), (i, [ 0, $+\infty$ ]), (i, [ 0, $+\infty$ ]) }

P8 = { (i, [ 0,0 ]), (T, [ 1, $+\infty$ ]), (i, [0, $+\infty$ ]) }

La suite des calculs montre la stabilisation des résultats.

f. Placement des tests à l'exécution

Soit $Q \in \Pi$ A, le résultat d'une analyse sémantique satisfaisant :

$$\forall c \in C,\ \forall m \in M^n,\ [(\text{poste } (t^*)\ (\emptyset) \wedge \text{pre } (t^*)\ (\Psi)]_{c \in C}\ (c, m) \Rightarrow \gamma(Q)\ (c, m)\ ;$$

soit la sémantique opérationnelle de la forme :

$t((c, m), (c', m')) = \underline{si}\ Te_c\ (m)\ \underline{alors}\ (c'\ m') = \bar{t}\ (c, m)\ \underline{sinon}\ (erreur, m)$ ;

si le programme est compilé, non pas pour réaliser t avec les tests à l'exécution $Te_c$ , $c \in C$, mais pour réaliser t' tel que

$t'((c, m), (c', m')) = \underline{si}\ Te'_c\ (m)\ \underline{alors}\ (c',m') = \bar{t}\ (c, m)\ \underline{sinon}\ (erreur, m)$ ;

appelons exécution correcte de t (t') une suite finie $e_1, \ldots e_k$ d'états tels que $k \geq 1$, $\phi(e_1)$, $\forall j \in [1, k-1]$, $t(e_j, e_{j+1})$ , $\psi(e_k)$

et $\forall j \in [1, k]$ , $\forall m \in M^p$ , $e_j \neq (erreur, m)$

th1

si $\forall c \in C$, $m \in M^n$, $[Te_c\ (m) \wedge \gamma(Q)\ (c,m)] \Rightarrow Te'_c\ (m)$,

alors toute exécution correcte de t est une exécution correcte de t'

dem

pour toute exécution correcte $e_1, \ldots e_k$ de t, on a :

$\forall j \in [1, k]$ , $[post\ (t^*)\ (\phi) \wedge pre\ (t^*)\ (\psi)]\ (e_j)$ et par conséquent $\gamma(Q)\ (e_j)$; de plus, si $e_j = (c_j, m_j)$ et $j < k$, on a

$t((c_j, m_j), (c_j + 1, m_j + 1))$

et $c_j + 1 \neq$ erreur donc $Te_{c_j}\ (m_j)$ ; il vient $Te'_{c_j}\ (m_j)$ et donc on a $t'((c_j, m_j), (c_j + 1, m_j))$.

La signification de ce théorème est que l'on peut rajouter des tests à l'exécution, sans que rien ne soit changé pour les exécutions correctes.

th2

si $\forall c \in C$, $m \in M^n$, $Te'_c(m) \Rightarrow [\gamma(F\ (Q))\ (c,m) \Rightarrow Te_c\ (m) \wedge \gamma(Q)\ (c,m)]$

alors toute exécution correcte de t' est une exécution correcte de t

dem

par définition de F :

$\alpha\ (\phi \vee post\ (t)\ (\gamma\ (Q))) \sqsubseteq F\ (Q)$,

donc $\gamma\ (\alpha\ (\phi \vee post\ (t)\ (\gamma\ Q)))) \Rightarrow \gamma(F(Q))$ car $\gamma$ est isotone,

donc $\phi \vee post\ (t)\ (\gamma\ (Q)) \Rightarrow \gamma\ (F(Q))$ car $1 \Rightarrow \gamma \circ \alpha$ ;

il vient

$Te'_c(m) \Rightarrow [[(\phi \vee post\ (t)\ (\gamma\ (Q))]\ (c,m) \Rightarrow [Te_c\ (m) \wedge \gamma(Q)\ (c,m)]]$

rem

c'est cette condition qui est vraiment nécessaire pour que toute exécution correcte de t' soit une exécution correcte de t, mais on

lui préfère la condition suffisante du théorème 2, car F a déjà été implanté en machine pour les besoins de l'analyse sémantique ;

soit maintenant $e_1, \ldots, e_k$ une exécution correcte pour t' où $\forall j \in [1,k]$,

on a ej = (cj, mj)

on a $\emptyset$ $(e_1)$,

donc $[\emptyset$ v post (t) ( $\gamma$ (Q)) $]$ $(e_1)$ ,

comme t' $(e_1, e_2)$ $\wedge$ $\forall m, e_2 \neq$ (erreur, m), on a $T'e_{c1}$ $(m_1)$ ;

on en déduit : $Te_{c1}$ $(m_1)$ $\wedge$ $\gamma(Q)$ (c1, m1) ;

et, si $k > 1$, on a t $(e_1, e_2)$, donc post (t) ( $\gamma$ (Q)) $(e_2)$ ;

supposons : par induction $j < k$ et post (t) ( $\gamma$ (Q)) (ej) ;

comme t' (ej, ej + 1) et cj + 1 $\neq$ erreur, on a $T'e_{cj}$ (mj)

et donc $Te_{cj}$ (mj) $\wedge$ $\gamma(Q)$ (cj, mj) ;

on a t (ej, ej + 1) et donc post (t) ( $\gamma$ (Q)) (ej + 1) ;

par induction, $\forall j \in [1, k]$ , t (ej, ej +1) donc $e_1, \ldots e_k$ est un chemin d'exécution correcte.

La signification de ce théorème est que l'on peut enlever des tests à l'exécution sans que rien ne soit changé pour les exécutions correctes. Les théorèmes 1 et 2 montrent que les exécutions correctes sont les mêmes pour t et t'.

Il resterait à montrer que toute exécution incorrecte de t' (c'est-à-dire une trace finie $e_1, \ldots$ ek avec ck = erreur ou une trace infinie $e_1 \ldots . e_j \ldots .$ avec $\emptyset$ $(e_1)$ et $\neg \Psi$ (ej) pour $j \geq 1$) est un préfixe d'une exécution incorrecte de t.

Dans la pratique, nous placerons des tests à l'exécution en tout point du programme ; les théorèmes 1 et 2 nous permettent d'enlever parmi ces tests ceux qui ne modifient en rien une exécution correcte ;

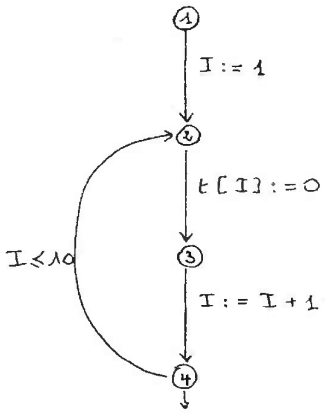
dans l'exemple précédent, il reste ainsi un test : il faut au point 2 du programme (c'est-à-dire juste après l'instruction de lecture), vérifier que la variable N est positive ou nulle - il est facile de prouver que pour N négatif, ce programme ne se termine pas.

Notons aussi que les bornes $-\infty$ et $+\infty$ n'existent pas en machine et sont représentées par respectivement le plus petit entier et le plus grand entier ; cette convention conduira à la mise en place de tests supplémentaires afin d'éviter les débordements arithmétiques (lors d'additions, soustractions, ...).

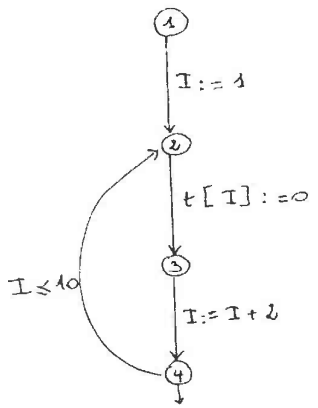
g) Extension aux tableaux

La méthode s'étend très rapidement aux tableaux d'entiers à un seul indice entier. En ce qui concerne l'indice, c'est une variable comme les autres ; les instructions où cet indice est utilisé sont plus diverses : par ex l'instruction t [ i ] : = ... où t est un tableau d'entiers et i un indice de ce tableau utilise la variable i : il faudra donc veiller à ce qu'elle soit bien initialisée avant (sinon veiller à la mise en place d'un test portant sur l'initialisation de i si celle-ci est indéterminée). L'initialisation du tableau est plus délicate à traiter, car à cause de la fonction d'élargissement, il n'est pas possible de savoir si tous les éléments du tableau ont été vus ou si seulement une partie d'entre eux a été initialisée.

ex :




il est clair que seuls, sur ce schéma, les éléments t [ 1 ] , ... t [ 10 ] seront initialisés à la valeur 0, mais prenons le même exemple modifié :



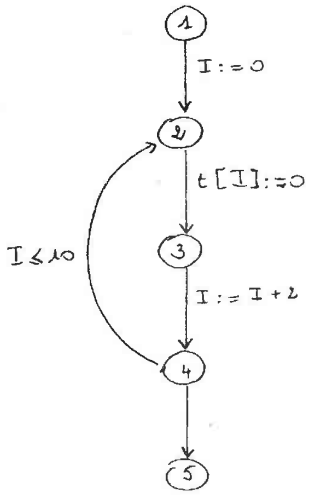
de façon générale, la perte d'information dûe au choix des intervalles de valeur (et non à des ensembles de valeurs) entraîne l'impossibilité de pouvoir distinguer entre les deux exemples.

Comme, de plus, l'élargissement est indispensable pour qu'une analyse sémantique puisse se faire un un temps raisonnable, il nous faudra perdre de l'information sur les tableaux et ne pas les considérer comme des variables entières normales.

Nous nous contenterons donc de savoir si aucun élément du tableau considéré n'est initialisé, sinon nous indiquerons l'initialisation par la marque d'indétermination.

La problème des intervalles numériques peut être résolu de façon similaire en considérant que le tableau est une seule variable dont l'initialisation ne sera que ⊥, ⊤i et T, et dont l'intervalle sera la réunion de tous les intervalles des éléments dont la marque d'initialisation n'est pas ⊤i.

par ex :




l'analyse sémantique appliquée à ce programme fournira l'information suivante en 5 : t = (T, [0,0] ). c'est-à-dire, il y a au moins un élément de t qui est initialisé et si on choisit un élément initialisé, sa valeur sera dans l'intervalle [ 0,0 ]

Comme nous étudions les tableaux, nous sommes amenés à considérer que les déclarations qui seront faites des variables ne seront plus seulement les valeurs entières, mais aussi des sous-ensembles d'entiers.

Pour les variables, il faudra faire attention lors des affectations de rester conforme aux déclarations du programmeur.

Pour les tableaux, les affectations aux éléments devront rester conformes aux déclarations ainsi que les utilisations des indices.

Voyons un exemple complet, celui de la recherche dichotomique d'une clé dans un tableau :

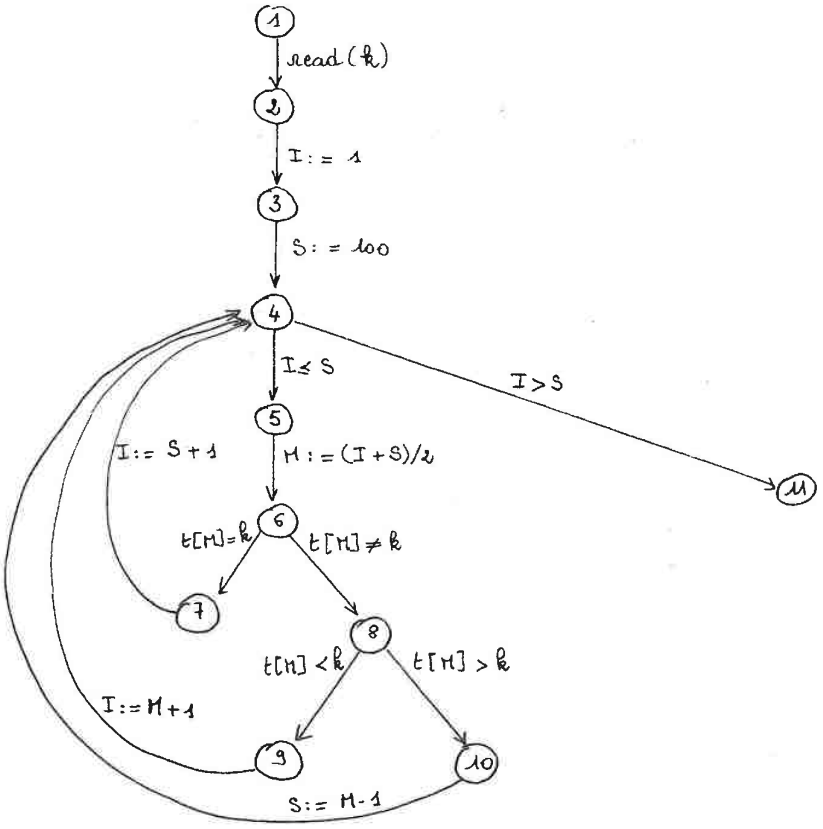
recherche dichotomique

```
program   dichotomie ;
var       I, S, M : integer ;
          t : array [ 1 .. 100 ] of integer ;
          k : integer ;          (* clé à rechercher *)
          l : integer ;          (* indice auxiliaire *)
          a : integer ;          (* valeur auxiliaire *)
begin
      for l : = 1  to  100 do
      begin
          read (a) ;
          t [l ] : = a ;
      end ;
      read (k) ;
      I : = 1 ;
      S : = 100 ;

      while  I ≤ S do
      begin
            M : = (I + S) div 2 ;
          if t [ M] = k then  I : = S + 1
          else if  t [ M ] >  k then   S : = M - 1
          else                         I : = M + 1 ;
      end ;
end.
```

La boucle for au début du programme sert uniquement à initialiser le tableau t (on suppose évidemment que les éléments du tableau sont triés)

Le graphe de dépendance simplifié est le suivant :




Le résultat de l'analyse sémantique est :

| | I | S | M | T |
|---|---|---|---|---|
| 1 | ⊤i | ⊤i | ⊤i | T, [ -∞ , +∞ ] |
| 2 | ⊤i | ⊤i | ⊤i | T, [ -∞ , +∞ ] |
| 3 | i, [1,1] | ⊤i | ⊤i | T, [ -∞ , +∞ ] |
| 4 | i, [1,101] | i, [0,100] | T, [1,100] | T, [ -∞ , +∞ ] |
| 5 | i, [1,100] | i, [1,100] | T, [1,100] | T, [ -∞ , +∞ ] |
| 6 | i, [1,100] | i, [1,100] | i, [1,100] | T, [ -∞ , +∞ ] |
| 7 | i, [1,100] | i, [1,100] | i, [1,100] | T, [ -∞ , +∞ ] |
| 8 | i, [1,100] | i, [1,100] | i, [1,100] | T, [ -∞ , +∞ ] |
| 9 | i, [1,100] | i, [1,100] | i, [1,100] | T, [ -∞ , +∞ ] |
| 10 | i, [1,100] | i, [1,100] | i, [1,100] | T, [ -∞ , +∞ ] |
| 11 | i, [1,101] | s, [0,100] | T, [1,100] | T, [ -∞ , +∞ ] |

A chaque utilisation de la variable m comme indice, l'analyse vérifie que m reste conforme aux déclarations [1,100] de l'indice de t et que M est bien initialisée.

## 7. Conclusion

Nous venons de voir les résultats obtenus par un analyse sémantique de deux exemples;dans un prochain chapitre, nous donnerons des exemples supplémentaires.

La première constatation que nous pouvons faire est que le nombre des tests qui devront être mis à l'exécution est de beaucoup plus faible que le nombre de tests mis par le compilateur, car les vérifications de débordement sont faites par l'analyse sémantique ; certains tests mis par l'analyse sémantique se révèlent fort utiles et pourtant le compilateur ne peut pas les trouver : par ex le test $n \geqslant 0$ dans l'exemple du tri par remontée de bulles. Ceci provient du fait que, par les intervalles, l'analyse sémantique garde un lien entre les différentes variables du programme. La vérification de l'initialisation n'est pas faite par les compilateurs et pourtant la non-initialisation des variables est source de bien des erreurs, très souvent difficilement détectables.

Ce qui va donc nous intéresser, c'est de pouvoir automatiser cette méthode et c'est l'objet de ce qui va suivre.

# IV Le Prototype d'Analyseur Sémantique

## 1. Introduction

Le prototype que nous avons construit est conçu pour analyser des programmes qui seront écrits dans un langage, qui sera un sous-ensemble du langage Pascal dont nous définirons la syntaxe exacte. Cette syntaxe, qui est relativement succinte, a été choisie ainsi afin de ne pas compliquer le programme du prototype, étant entendu que le principal problème qui nous préoccupait était la sémantique. L'analyseur sera décomposé en deux parties : la première phase de préparation s'occupera de reconnaissance syntaxique, construction de graphes de dépendance, d'étude de ces graphes et construction d'interpréteur abstrait ; la seconde phase sera consacrée uniquement à l'analyse sémantique.

## 2. Syntaxe

Les différentes instructions sont :

- les affectations
- les sauts conditionnels if .... then goto
- les sauts inconditionnels goto
- les boucles for, while, repeat non imbriquées
- les lectures read, readln
- les écritures write, writeln
- les conditionnelles if ... then ... (else...)

Les expressions arithmétiques comportent un opérateur unaire ou binaire (+, -, *, div, mod).

Les expressions logiques sont simples : un opérateur logique binaire ( $<$ , $\leq$ , $>$ , $\geq$ , =, $<>$ ).

Les variables seront des variables entières (ou d'intervalles entières) ou des tableaux d'entiers (ou intervalles d'entiers) à un indice à valeurs dans les entiers (ou les intervalles d'entiers).

Les déclarations des étiquettes sont, comme en Pascal, obligatoires et les déclarations des variables ne feront apparaître que les deux types suivants : entier et intervalle d'entiers.

Les erreurs de syntaxe les plus flagrantes seront détectées au moment de la reconnaissance syntaxique et produiront un arrêt de l'exécution du programme.

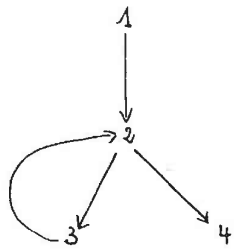
Pendant la reconnaissance de la syntaxe, nous construisons des tables d'ares et d'instructions qui représentent en mémoire les graphes de dépendance (le graphe et son inverse).

Les instructions qui se trouvent dans une boucle while, for, repeat ou une conditionnelle if .. then .. (else .. sont réduites à la lecture, l'écriture ou l'affectation, ce qui interdit momentanément l'imbrication de telles instructions et oblige de ce fait l'utilisation de l'instruction de saut ; ce choix a été fait afin que, dans un premier temps, la reconnaissance syntaxique ne soit pas trop importante.

Après la construction des graphes une étude rapide permet de détecter les erreurs grossières comme par ex : les boucles sans fin (1 : goto 1). Pour les programmes contenant ce type d'erreurs, il sera demandé au programmeur de revoir son programme.

3. Etude du graphe

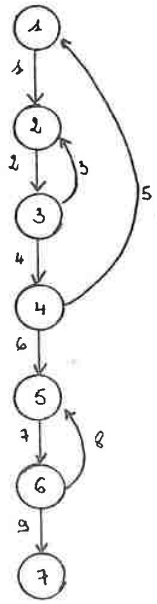
Cette étude fera l'objet d'un complet chapitre ultérieurement ; dans ce chapitre, nous allons seulement donner l'idée générale du problème. Dans le programme suivant représenté par le graphe :




Nous voyons que l'analyse symbolique aura le noeud 1 comme point de départ ; intuitivement, nous nous doutons qu'elle va exécuter la boucle 2 - 3 jusqu'à obtention d'une condition qui lui permettra d'atteindre le point 4. L'exécution calcule la condition de sortie de la boucle 2 - 3 - 2 grâce aux valeurs exactes des variables du programme. Ce calcul d'une condition de sortie ne pourra évidemment se faire de façon identique lors d'une analyse sémantique puisque les valeurs exactes ne seront pas connues.

Pour l'analyse sémantique, il faudra donc connaître, à partir du graphe seul, les chemins qu'il faudra emprunter, dans quel ordre et sous quelle condition.

Intuitivement, nous pourrons affirmer qu'il faut analyser une boucle avant de pouvoir analyser l'arc de sortie lui correspondant et que lors d'une jonction simple de chemins, il faut attendre que toutes les informations en provenance de ces chemins soient présentes. Sur le petit exemple, l'exécution symbolique se ferait donc ainsi : (1 2), (2 3), (3 2), 2 3), (3 2), ...., (3 2), (2 4).
La condition de sortie serait calculée par l'analyse sémantique proprement dite mais on peut trouver un ordre d'itération indépendamment du calcul de cette condition. Nous verrons que la numérotation des arcs va nous faciliter le travail et permettra d'écrire cette représentation schématique sous la forme d'une expression régulière. Afin de permettre une meilleure compréhension du sujet, voyons l'exemple suivant où les arcs ont été numérotés à l'avance :




L'ordre de parcours est facile à trouver sachant ce que nous venons de préciser : arcs 1, (entrée de boucle), (2,3), (2,3), ... (2,3), (sortie de boucle), 4, 5 on recommence cette séquence jusqu'à obtenir stabilité des résultats en tous les points puis on peut quitter la boucle et analyser l'arc 6, entrer dans la boucle (7,8) et l'analyser jusqu'à obtenir pour elle aussi stabilité des résultats ; à ce moment, on peut analyser l'arc 9.

L'étude du graphe devra comprendre de plus la recherche des points où devront se faire les élargissements dans l'analyse sémantique du programme considéré. L'élargissement permet à toute boucle de voir ses

résultats stabilisés plus rapidement ; il est donc indispensable que toute boucle dans le programme puisse bénéficier de cette propriété ; et par conséquent, toute boucle doit contenir parmi les points qui la constituent un point particulier appelé noeud d'élargissement ou point de coupure. Etant donné que l'élargissement va dans le sens d'une perte de précisions sur les résultats, il est préférable que l'ensemble des points de coupure de tout programme soit le plus petit possible.
Par ex : dans le graphe précédent, les noeuds 2 et 5 constituent le plus petit ensemble de points de coupure.

Ces deux thèmes seront repris dans le chapitre correspondant.

Toute anomalie dans le graphe (instructions non accessibles physiquement, entrées ou sorties multiples du programme, ...) sera détectée à ce stade de l'étude et l'arrêt de l'analyse sera automatique afin de permettre au programmeur de corriger son programme.

## 4. Compilateur abstrait

Pour l'exécution, nous avons tout d'abord écrit les équations sémantiques associées aux graphes de dépendance, puis nous avons résolu ces équations. Mais il semble évident que automatiquement, il n'est pas raisonnable de vouloir écrire ces équations et les résoudre. La méthode que nous allons employer consistera à choisir des actions qui permettront de simuler la résolution des équations.

ex $\{P1\}$ qui aura pour équations

$x := x + 1$ $\qquad P2 = P1\ (x \longleftarrow x + 1)$

$\{P2\}$

pourra en fait s'écrire plus simplement :

à partir du contexte abstrait en P1, exécuter ($x \longleftarrow X + 1$)
et ranger le résultat dans le contexte de P2.

Pour cela, nous allons décrire des instructions élémentaires qui seront utilisées pour constituer le code généré. Ces instructions ont pour origines les actions élémentaires rencontrées par l'analyse sémantique (par ex : exécuter un arc) ou les actions élémentaires que l'on trouve dans tout langage d'assemblage (par ex : saut conditionnel).

A partir de ces instructions et des résultats obtenus par l'étude du graphe de dépendance d'un programme, l'analyseur (dans une première

passe de compilation) génère une suite de ces instructions, formant ainsi un interpréteur abstrait dont l'exécution aura pour résultat les résultats attendus de l'analyse sémantique. Les instructions de ce langage auront la particularité de ne pas travailler sur les valeurs réelles des variables mais sur leurs valeurs abstraites ; les instructions pourront être considérées comme des fonctions dont les ensembles de définition varient selon l'instruction. Appelons provisoirement C le contexte abstrait d'un noeud ;

* la première de ces instructions est l'instruction d'exécution d'un arc du graphe (c'est-à-dire une affectation, une branche de test, une lecture, ...)

  arc : $N \times C \longrightarrow C$

  $(j, m) \longmapsto O$

  où j désignera l'instruction à exécuter,

  m sera le numéro du contexte de départ

  le résultat sera mis dans un contexte dit de travail et référencé par la valeur O ceci permettra de vérifier la cohérence de tout résultat intermédiaire - après une exécution d'arc-avant de ranger le résultat dans le contexte convenable.

* Nous noterons que l'analyse en avant fait toujours référence aux résultats obtenus par l'analyse en arrière (et vice-versa, l'analyse en arrière fait toujours référence aux résultats obtenus par l'analyse en avant) ; il faut donc prévoir de pouvoir faire l'intersection de ces deux analyses à tout moment, spécialement lorsque l'un des contextes vient d'être modifié (par un calcul, un élargissement, ...)

  inter : $C \times C \longrightarrow C$

  $(i, j) \longmapsto i.$

La jonction de chemins dans le graphe nous fait apparaître deux possibilités lors de l'analyse sémantique : la phase d'élargissement et celle de rétrécissement. Mais l'élargissement (ou le rétrécissement) ne se fait qu'aux noeuds qui ont été désignés par l'étude du graphe comme points de coupure, les autres noeuds de jonction étant des noeuds de jonction normaux ; pour les noeuds de coupure, ce seront les fonctions d'élargissement et de rétrécissement qui seront utilisées ; pour les autres, il suffira de connaître les fonctions de réunion et d'intersection ;

* élargissement : $C \times C \longrightarrow C$

  - elargt - $(i, j) \longmapsto j$ soit $C(j) = C(i) \nabla C(j)$

* rétrécissement : $C \times C \longrightarrow C$

  - retrect- $(i, j) \longmapsto j$ soit $C(j) = C(i) \Delta C(j)$

* réunion : $C \times C \longrightarrow C$

  - reuna - $(i, j) \longmapsto j$ soit $C(j) = C(i) \cup C(j)$

* intersection : $C \times C \longrightarrow C$

  - interc - $(i, j) \longmapsto j$ soit $C(j) = C(i) \cap C(j)$

Notons que interc fait l'intersection de deux contextes calculés par l'analyse en avant (ou deux contextes calcules par l'analyse en arrière) alors que inter fait l'intersection d'un contexte abstrait trouvé par l'analyse en avant avec un contexte abstrait calculé par l'analyse en arrière : ces deux fonctions sont différentes et il était préférable de bien les dissocier.

Nous aurons bien entendu besoin de l'instruction de chargement que nous noterons :

rangtda : $C \longrightarrow C$ qui range un contexte i dans un contexte j

$i \longmapsto j$

Les fonctions d'élargissement et de rétrécissement, bien que similaire, ont dû être séparées lors de la création de l'interpréteur ; ceci est dû au choix que nous avons fait lors de l'étude du graphe de dépendance et, plus particulièrement, l'étude des noeuds de coupure.

Nous sommes de plus amenés à conserver certaines valeurs à ces différents noeuds, et à introduire de ce fait des contextes spéciaux que nous appellerons contextes de rétrécissement. Et bien entendu, il faudra être capable de charger des valeurs dans ces contextes de rétrécissement, ainsi que de récupérer des valeurs rangées.

Ce qui nous fait définir les fonctions de rangement suivantes :

rangtdr : $C \longrightarrow C$ rétrécissement

rangtra : $C \longrightarrow C$.
rétrécissement

Ces fonctions seront complétées par une fonction permettant d'ajouter de nouvelles valeurs dans ces contextes de rétrécissement

reunr : $C \times C \longrightarrow C$
rétrécissement rétrécissement rétrécissement

* rangtdr : $i \longmapsto j$ soit $C(j) = C(i)$ retrec.

* rangtra : $i \longmapsto j$ soit $C(j) = C(i)$ retrec

* reunr : $(i,j) \longmapsto j$ soit $C(j) = C(i)$ retrec $U$ $C(j)$ retrec.

C'est à ce niveau que vont apparaître les conditions de stabilité dont nous parlions sous peu. Ces conditions vont déterminer si une boucle mérite d'être analysée une nouvelle fois ou si l'exécution peut quitter la boucle et passer à l'arc suivant. Une fonction comp va comparer deux contextes et fournira la réponse par l'intermédiaire d'une variable cond booléenne :

* cond : $C \times C \longrightarrow B$
$(i,j)$ où $= C(i)$ opc $C(j)$ et opc est un opérateur de comparaison

C (i) et C (j) étant des contextes, le seul moyen de les comparer est de voir lequel contient l'autre. Nous faisons le choix opc = C et il nous suffira de changer la place de i et j si nous voulons obtenir l'inclusion inverse.

Le saut conditionnel sera obtenu par la fonction jump qui aura pour argument la valeur booléen cond et pour paramètre une valeur booléenne :si cond est égale à cette valeur, alors le saut au numéro de ligne indiqué comme résultat devra s'effectuer, sinon continuer en séquence.

* jump : $B \longrightarrow N$ p sera en général choisi à tme
$p \longmapsto n$ n sera le numéro de ligne dans l'interpréteur abstrait

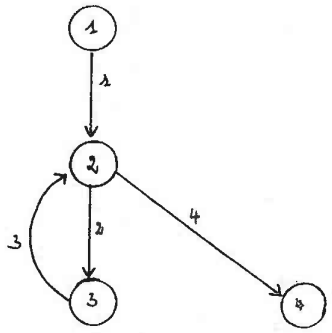
Le saut inconditionnel sera désigné par

* jumpi : $\{\ \} \longrightarrow N$
n

Grâce à ces 13 fonctions nous allons pouvoir construire le compilateur abstrait mais auparavant, nous pouvons faire quelques remarques.

Tout d'abord, de la même façon que nous devions conserver les contextes de rétrécissement, nous allons devoir garder pour chaque noeud tête de cycle (point de coupure ou non) les contextes de l'exécution précédente

mais au lieu de créer une nouvelle classe de contextes - ce qui nous obligerait à utiliser de nouvelles fonctions d'interprétation -, nous allons tout simplement utiliser les contextes correspondant aux noeuds de numérotation. De la sorte les fonctions portant sur les contextes "normaux" (ie par les contextes de rétrécissement) sont toujours utilisables. Les contextes de comparaison seront numérotés à partir de - 1 en décroissant à mesure que les noeuds têtes de cycle seront rencontrés. Un problème auquel il faudra faire très attention est celui des graphes pour lesquels il existe une boucle qui a plusieurs entrées : ce cas, s'il est souvent oublié, perturbe considérablement les calculs de l'analyse sémantique.
Pour chaque programme, il suffira d'écrire le code abstrait en avant et le code abstrait en arrière et les informations sur les graphes deviennent alors sans objet.
Donnons l'exemple du code abstrait en avant pour l'exemple suivant :




Nous omettrons volontairement le détail des techniques de compilation utilisées pour obtenir le résultat suivant :

| | | | |
|---|---|---|---|
| 1 | arc | 1 | 1 |
| 2 | inter | 0 | 2 |
| 3 | rangtda | 0 | 2 |
| 4 | comp | 2 | - 1 |
| 5 | jump | 4 | 16 |
| 6 | élargt | -1 | 2 |
| 7 | inter | 2 | 2 |
| 8 | rangtda | 2 | - 1 |
| 9 | arc | 2 | 2 |
| 10 | inter | 0 | 3 |
| 11 | rangtda | 0 | 3 |

| | | | |
|---|---|---|---|
| 12 | arc | 3 | 3 |
| 13 | inter | 0 | 2 |
| 14 | réuna | 0 | 2 |
| 15 | jumpi | 4 | |
| 16 | arc | 4 | 2 |
| 17 | inter | 0 | 4 |
| 18 | rangtda | 0 | 4 |

Ces 18 instructions élémentaires décrivent uniquement la phase d'élargissement. La construction de ce compilateur nécessite une étude rapide de l'ordre de parcours délivré comme résultat par l'étude du graphe. Ce compilateur fait gagner du temps (50 % à 75 %) à l'analyse sémantique : pour donner un ordre d'idée, pour un programme d'environ 150 lignes, le gain de temps est de 50 secondes pour un temps d'exécution (d'analyse sémantique) total de 75 sec. Mais nous devrons faire d'autres comparaisons afin de pouvoir juger efficacement du gain réalisé.

## 5. Analyse sémantique proprement dite

La phase d'analyse sémantique est constituée de deux parties : l'exécution de l'analyse en avant et l'exécution de l'analyse en arrière. Ces exécutions se font directement à partir des interpréteurs abstraits décrits ci-dessus ; il reste à décrire la mise en oeuvre des fonctions communes à ces deux analyses, à savoir l'arithmétique et les predicats sur les valeurs abstraites. La dernière phase de l'analyseur sémantique consiste à calculer les tests qui doivent être mis à l'exécution.
La description des actions propres à chaque situation ou à chaque instruction a été faite dans tout ce qui a précédé.

## 6. Conclusions

Nous avons effectué un regroupement des différentes phases afin de rendre le programme plus lisible et plus structuré d'une part, d'utiliser le minimum de place avec les variables d'autre part.
Ce programme complet a été écrit en langage Pascal et a été implanté sur ordinateur IRIS 80 ; il compte environ 6500 instructions et a été divisé en deux parties sensiblement de même taille.

La première partie regroupe la phase d'analyse syntaxique et de construction des graphes ($\sim$1800 lignes), d'étude des graphes ($\sim$500 lignes) et de construction des interpréteurs abstraits ($\sim$ 700 lignes) : en tout $\sim$ 3000 lignes.
La seconde partie ne s'occupe que de la sémantique et des tests ( $\sim$ 3500 lignes).

## 7. Annexe : utilisation de l'analyseur

Avant de commencer l'étude d'un programme, l'analyseur a besoin de deux valeurs d'indicateurs qui sont les options. La première option concerne la première phase de l'analyseur : six valeurs de 0 à 5 sont possibles et ces valeurs correspondent à :

pour 0 : aucune informatique n'est écrite
pour 1 : écriture de la table des arcs et de l'interpréteur abstrait
pour 2 : écriture de toutes les tables construites pendant la phase d'analyse syntaxique (arcs, étiquettes, instructions, noeuds, noeuds d'élargissement)
écriture des expressions régulières représentant les graphes
écriture de l'interpréteur abstrait

les options 3, 4 et 5 reprennent les options 0, 1 et 2 respectivement et font en plus une étude complémentaire du graphe afin de détecter les boucles mal construites.

La seconde option concerne la phase sémantique et permet la sortie de résultats intermédiaires. 11 valeurs sont possibles

0 : liste des tests à l'exécution
1 : liste des tests à l'exécution et liste des numéros des lignes dans lesquelles apparaissent des erreurs
2 : liste des tests à l'exécution
contexte final sans autre indication que les valeurs abstraites de toutes les variables en tous les noeuds du graphe
3 : ajoute à l'option 2 une analyse en avant préliminaire, permettant de découvrir le type d'une erreur apparue

4 : ajoute à l'option 2 tous les contextes intermédiaires

5 : même résultat que l'option 4 mais signale le code des erreurs

Les options suivantes (de 6 à 10) ne peuvent être utilisées que sur de petits exemples à cause de la quantité d'informations réunies

6 : ajoute à 5 la liste des fonctions appelées de l'interpréteur

7 : ajoute à 6 toutes les analyses en avant détaillées

8 : ajoute à 6 toutes les analyses en arrières détaillées

9 : ajoute à 6 toutes les analyses détaillées

Ces options ont surtout servi à vérifier que les fonctions sémantiques étaient bien conformes à leurs spécifications.

L'option 10 est une option 9 réduite (elle évite l'impression de contextes intermédiaires tant que ceux-ci sont invariants).

Pendant toute l'analyse, nous avons conservé avec l'indicateur d'erreur un code qui permet dans une très large mesure de trouver d'où provient l'erreur. Nous avons donc rangés les différentes erreurs possibles dans sept catégories :

- utilisation d'une variable non intialisée
- débordement arithmétique (dans le calcul d'une expression)
- débordement logique (comparaison de deux variables dont les intervalles ne sont pas compatibles pour ce test. Ce qui conduit à une branche de test inaccessible)
- incompatibilité avec les déclarations
- incompatibilité de variables (cas où l'on veut affecter une valeur d'une variable à une autre variable dont les valeurs abstraites sont incompatibles)
- incompatibilité avant-arrière (l'information obtenue par l'analyse en avant est incompatible avec l'information obtenue par l'analyse en arrière)
- valeur incompatible avec une fonction (ex mod - 1 est interdit)

Remarque

Dans la machine, il n'existe pas de $-\infty$ ou $+\infty$ ; nous avons donc utilisé les plus petite et plus grande valeurs entières - 32768 et + 32767. Bien que l'IRIS 80 soit une machine 32 bits, nous avons préféré ces deux

valeurs pour la lisibilité. Le fait que les infinis soient devenus des valeurs finies va compliquer un peu les vérifications et obliger à ajouter des tests dans les fonctions arithmétiques et les prédicats afin d'éviter les débordements ; dans toutes ces fonctions, il faudra tenir compte du fait que chaque opération doit donner un résultat qui doit rester entre les bornes numériques fixées.

## V - Exemples

1. programme : uniforme - binary - search : temps d'analyse $\frac{19}{100^e}$ mn

2. programme : bubble - sorting : temps d'analyse $\frac{23}{100^e}$ mn

3. programme : gries - justify : temps d'analyse $\frac{25}{100^e}$ mn

4. programme : gries - justify : déduit du précédent par modification de la ligne 23

5. programme : ackermann : temps d'analyse $\frac{20}{100^e}$ mn

6. programme : partition - exchange - sort : temps d'analyse $\frac{105}{100^e}$ mn

7. programme : list marge - sort : temps d'analyse $\frac{132}{100^e}$ mn

8. programme : pged (erroné : à la ligne 41)

9. programme : binary - search (erroné : k non initialisé)

```
 1    PROGRAM UNIFORM-BINARY-SEARCH;
 2
 3    LABEL 20,30,40,97,98,99;
 4
 5    VAR
 6         K1 : ARRAY[1..1000] OF INTEGER;
 7         K : INTEGER;
 8         I : INTEGER;
 9         N : INTEGER;
10         M : INTEGER;
11         P : INTEGER;
12         A : INTEGER;
13         U : INTEGER;
14         R : 0..1;
15
16    BEGIN
17       READ(N);
18       READ(K);
19       U := 1;
20       WHILE   U <= N   DO
21       BEGIN
22          READ(A);
23          K1[U] := A;
24          U := U + 1;
25       END;
26       M := N DIV 2;
27       I := N DIV 2;
28    20:
29       IF   K < K1[I]   THEN GOTO 30;
30       IF   K > K1[I]   THEN GOTO 40;
31       GOTO 97;
32    30:
33       IF   M = 0   THEN GOTO 98;
34       P := M DIV 2;
35       I := I - P;
36       M := M DIV 2;
37       GOTO 20;
38    40:
39       IF   M = 0   THEN GOTO 98;
40       P := M DIV 2;
41       I := I + P;
42       M := M DIV 2;
43       GOTO 20;
44    97:
45       R := 1;
46       GOTO 99;
47    98:
48       R := 0;
49    99:
50    END.
51
```

LISTE DES TESTS A L EXECUTION
==========

| | | | | |
|---|---|---|---|---|
| LIGNE : | 18 | N | >= | 2 |
| LIGNE : | 18 | N | <= | 1000 |
| LIGNE : | 36 | I | >= | 1 |
| LIGNE : | 36 | I | <= | 1000 |
| LIGNE : | 42 | I | >= | 1 |
| LIGNE : | 42 | I | <= | 1000 |

```
 1  PROGRAM BUBBLE-SORTING;
 2
 3  LABEL 1,2,3,4,5;
 4
 5  VAR
 6        I : INTEGER;
 7        J : INTEGER;
 8        N : INTEGER;
 9
10        S1 : ARRAY[0..1000] OF INTEGER;
11        X : INTEGER;
12        K : INTEGER;
13        L : INTEGER;
14
15
16  BEGIN
17     READ(N);
18     K := 1;
19     WHILE    K <= N    DO
20     BEGIN
21        READ(X);
22        S1[K] := X;
23        K := K + 1;
24     END;
25     I := N;
26  1:
27     IF    I = 0    THEN GOTO 5;
28     J := 0;
29  2:
30     IF    J <> I    THEN GOTO 3;
31     I := I - 1;
32     GOTO 1;
33  3:
34     L := J + 1;
35     IF    S1[J] <= S1[L]    THEN GOTO 4;
36     K := S1[L];
37     S1[L] := S1[J];
38     S1[J] := K;
39  4:
40     J := J + 1;
41     GOTO 2;
42  5:
43  END.
```

CONTEXTE GENERAL

A LA LIGNE : 17

```
I  :: N-INIT
J  :: N-INIT
N  :: N-INIT
S1 :: N-INIT
X  :: N-INIT
K  :: N-INIT
L  :: N-INIT
```

A LA LIGNE : 18

```
I  :: N-INIT
J  :: N-INIT
N  ::   INIT  ,  (    0 ,  1000 )
S1 :: N-INIT
X  :: N-INIT
K  :: N-INIT
L  :: N-INIT
```

```
A LA LIGNE  :   19
.................

        I   ::  N-INIT
        J   ::  N-INIT
        N   ::     INIT  ,     (      0 ,  1000 )
        S1  ::     INIT-?,     (-32768 , 32767 )
        X   ::     INIT-?,     (-32768 , 32767 )
        K   ::     INIT  ,     (      1 ,  1001 )
        L   ::  N-INIT


A LA LIGNE  :   21
.................

        I   ::  N-INIT
        J   ::  N-INIT
        N   ::     INIT  ,     (      1 ,  1000 )
        S1  ::     INIT-?,     (-32768 , 32767 )
        X   ::     INIT-?,     (-32768 , 32767 )
        K   ::     INIT  ,     (      1 ,  1000 )
        L   ::  N-INIT


A LA LIGNE  :   22
.................

        I   ::  N-INIT
        J   ::  N-INIT
        N   ::     INIT  ,     (      1 ,  1000 )
        S1  ::     INIT-?,     (-32768 , 32767 )
        X   ::     INIT  ,     (-32768 , 32767 )
        K   ::     INIT  ,     (      1 ,  1000 )
        L   ::  N-INIT


A LA LIGNE  :   23
.................

        I   ::  N-INIT
        J   ::  N-INIT
        N   ::     INIT  ,     (      1 ,  1000 )
        S1  ::     INIT-?,     (-32768 , 32767 )
        X   ::     INIT  ,     (-32768 , 32767 )
        K   ::     INIT  ,     (      1 ,  1000 )
        L   ::  N-INIT


A LA LIGNE  :   25
.................

        I   ::  N-INIT
        J   ::  N-INIT
        N   ::     INIT  ,     (      0 ,  1000 )
        S1  ::     INIT-?,     (-32768 , 32767 )
        X   ::     INIT-?,     (-32768 , 32767 )
        K   ::     INIT  ,     (      1 ,  1001 )
        L   ::  N-INIT
```

```
A LA LIGNE  :    27
.................

        I   ::      INIT   ,     (      0 ,   1000 )
        J   ::      INIT-?,      (      1 ,   1000 )
        N   ::      INIT   ,     (      0 ,   1000 )
        S1  ::      INIT-?,      (-32768 , 32767 )
        X   ::      INIT-?,      (-32768 , 32767 )
        K   ::      INIT   ,     (-32768 , 32767 )
        L   ::      INIT-?,      (      1 , 32767 )

A LA LIGNE  :    28
.................

        I   ::      INIT   ,     (      1 ,   1000 )
        J   ::      INIT-?,      (      1 ,   1000 )
        N   ::      INIT   ,     (      0 ,   1000 )
        S1  ::      INIT-?,      (-32768 , 32767 )
        X   ::      INIT-?,      (-32768 , 32767 )
        K   ::      INIT   ,     (-32768 , 32767 )
        L   ::      INIT-?,      (      1 , 32767 )

A LA LIGNE  :    30
.................

        I   ::      INIT   ,     (      1 ,   1000 )
        J   ::      INIT   ,     (      0 ,   1000 )
        N   ::      INIT   ,     (      0 ,   1000 )
        S1  ::      INIT-?,      (-32768 , 32767 )
        X   ::      INIT-?,      (-32768 , 32767 )
        K   ::      INIT   ,     (-32768 , 32767 )
        L   ::      INIT-?,      (      1 , 32767 )

A LA LIGNE  :    31
.................

        I   ::      INIT   ,     (      1 ,   1000 )
        J   ::      INIT   ,     (      1 ,   1000 )
        N   ::      INIT   ,     (      0 ,   1000 )
        S1  ::      INIT-?,      (-32768 , 32767 )
        X   ::      INIT-?,      (-32768 , 32767 )
        K   ::      INIT   ,     (-32768 , 32767 )
        L   ::      INIT-?,      (      1 , 32767 )

A LA LIGNE  :    34
.................

        I   ::      INIT   ,     (      1 ,   1000 )
        J   ::      INIT   ,     (      0 ,    999 )
        N   ::      INIT   ,     (      0 ,   1000 )
        S1  ::      INIT-?,      (-32768 , 32767 )
        X   ::      INIT-?,      (-32768 , 32767 )
        K   ::      INIT   ,     (-32768 , 32767 )
        L   ::      INIT-?,      (      1 , 32767 )
```

```
A LA LIGNE  :   35
.................

        I   ::      INIT   ,     (      1 ,  1000 )
        J   ::      INIT   ,     (      0 ,   999 )
        N   ::      INIT   ,     (      0 ,  1000 )
        S1  ::      INIT-?,      (-32768 , 32767 )
        X   ::      INIT-?,      (-32768 , 32767 )
        K   ::      INIT   ,     (-32768 , 32767 )
        L   ::      INIT   ,     (      1 ,  1000 )

A LA LIGNE  :   36
.................

        I   ::      INIT   ,     (      1 ,  1000 )
        J   ::      INIT   ,     (      0 ,   999 )
        N   ::      INIT   ,     (      0 ,  1000 )
        S1  ::      INIT-?,      (-32768 , 32767 )
        X   ::      INIT-?,      (-32768 , 32767 )
        K   ::      INIT   ,     (-32768 , 32767 )
        L   ::      INIT   ,     (      1 ,  1000 )

A LA LIGNE  :   37
.................

        I   ::      INIT   ,     (      1 ,  1000 )
        J   ::      INIT   ,     (      0 ,   999 )
        N   ::      INIT   ,     (      0 ,  1000 )
        S1  ::      INIT-?,      (-32768 , 32767 )
        X   ::      INIT-?,      (-32768 , 32767 )
        K   ::      INIT   ,     (-32768 , 32767 )
        L   ::      INIT   ,     (      1 ,  1000 )

A LA LIGNE  :   38
.................

        I   ::      INIT   ,     (      1 ,  1000 )
        J   ::      INIT   ,     (      0 ,   999 )
        N   ::      INIT   ,     (      0 ,  1000 )
        S1  ::      INIT-?,      (-32768 , 32767 )
        X   ::      INIT-?,      (-32768 , 32767 )
        K   ::      INIT   ,     (-32768 , 32767 )
        L   ::      INIT   ,     (      1 ,  1000 )

A LA LIGNE  :   40
.................

        I   ::      INIT   ,     (      1 ,  1000 )
        J   ::      INIT   ,     (      0 ,   999 )
        N   ::      INIT   ,     (      0 ,  1000 )
        S1  ::      INIT-?,      (-32768 , 32767 )
        X   ::      INIT-?,      (-32768 , 32767 )
        K   ::      INIT   ,     (-32768 , 32767 )
        L   ::      INIT   ,     (      1 ,  1000 )
```

A LA LIGNE : 43
.....

```
I  ::    INIT  ,    (     0 ,     0 )
J  ::    INIT-?,    (     1 ,  1000 )
N  ::    INIT  ,    (     0 ,  1000 )
S1 ::    INIT-?,    (-32768 , 32767 )
X  ::    INIT-?,    (-32768 , 32767 )
K  ::    INIT  ,    (-32768 , 32767 )
L  ::    INIT-?,    (     1 , 32767 )
```

LISTE DES TESTS A L EXECUTION
==========

```
LIGNE :  18  N  >=          0
LIGNE :  18  N  <=       1000
LIGNE :  34  J  <=        999
```

```
 1    PROGRAM GRIES-JUSTIFY;
 2
 3    LABEL 21,27,33;
 4
 5    VAR
 6         N : 0..32767;
 7         Z : 0..32767;
 8         S : 0..32767;
 9
10         B : INTEGER;
11         P : 0..32767;
12         Q : 0..32767;
13         T : 1..32767;
14         E : INTEGER;
15
16         K : 1..32767;
17         A : INTEGER;
18         X : INTEGER;
19         C : INTEGER;
20
21   BEGIN
22      READ(X);
23      N := 100;
24      READ(Z);
25      READ(S);
26      READ(B);
27      IF   N <= 1   THEN GOTO 33;
28      A := Z MOD 2;
29      IF   A = 0   THEN
30      BEGIN
31         A := N - 1;
32         Q := S DIV A;
33         C := A;
34         A := S MOD C;
35         T := 1 + A;
36         P := Q + 1;
37      END
38      ELSE
39      BEGIN
40         A := N - 1;
41         P := S DIV A;
42         C := A;
43         A := S MOD C;
44         T := N - A;
45         Q := P + 1;
46      END;
47      K := N;
48      E := S;
49   21:
50      IF   K = T   THEN GOTO 27;
51      IF   X <> X   THEN
52      BEGIN
53         B := B + E;
54      END;
55      K := K - 1;
56      E := E - Q;
57      GOTO 21;
58   27:
59      IF   E = 0   THEN GOTO 33;
60      IF   X <> X   THEN
61      BEGIN
62         B := B + E;
63      END;
64      K := K - 1;
65      E := E - P;
66   33:
67   END.
```

LISTE DES TESTS A L EXECUTION
==========

```
LIGNE :   25   Z   >=          0
LIGNE :   26   S   >=          0
LIGNE :   54   K   >=          2
LIGNE :   63   K   >=          2
```

```
PROGRAM GRIES-JUSTIFY;

LABEL 21,27,33;

VAR
      N : 0..32767;
      Z : 0..32767;
      S : 0..32767;

      B : INTEGER;
      P : 0..32767;
      Q : 0..32767;
      T : 1..32767;
      E : INTEGER;

      K : 1..32767;
      A : INTEGER;
      X : INTEGER;
      C : INTEGER;

BEGIN
   READ(X);
   READ(N);
   READ(Z);
   READ(S);
   READ(B);
   IF   N <= 1   THEN GOTO 33;
   A := Z MOD 2;
   IF   A = 0   THEN
   BEGIN
      A := N - 1;
      Q := S DIV A;
      C := A;
      A := S MOD C;
      T := 1 + A;
      P := Q + 1;
   END
   ELSE
   BEGIN
      A := N - 1;
      P := S DIV A;
      C := A;
      A := S MOD C;
      T := N - A;
      Q := P + 1;
   END;
   K := N;
   E := S;
21:
   IF   K = T   THEN GOTO 27;
   IF   X <> X   THEN
   BEGIN
      B := B + E;
   END;
   K := K - 1;
   E := E - Q;
   GOTO 21;
27:
   IF   E = 0   THEN GOTO 33;
   IF   X <> X   THEN
   BEGIN
      B := B + E;
   END;
   K := K - 1;
   E := E - P;
33:
END.
```

LISTE DES TESTS A L EXECUTION

| | | | | |
|---|---|---|---|---|
| LIGNE : | 24 | N | >= | 0 |
| LIGNE : | 25 | Z | >= | 0 |
| LIGNE : | 26 | S | >= | 0 |
| LIGNE : | 33 | Q | <= | 32766 |
| LIGNE : | 42 | P | <= | 32766 |
| LIGNE : | 45 | T | >= | 1 |
| LIGNE : | 54 | K | >= | 2 |
| LIGNE : | 63 | K | >= | 2 |

```
 1  PROGRAM ACKERMANN;
 2
 3  LABEL 1,2,3,4,5,6;
 4
 5  VAR
 6        V : ARRAY[1..1000] OF INTEGER;
 7        P : ARRAY[1..1000] OF INTEGER;
 8        M : INTEGER;
 9        I : INTEGER;
10        L : INTEGER;
11        J : INTEGER;
12        Z : INTEGER;
13        N : INTEGER;
14
15  BEGIN
16     READ(M);
17     READ(N);
18     IF    M = 0    THEN GOTO 5;
19     L := M + 1;
20     J := 1;
21     WHILE    J <= L    DO
22     BEGIN
23        V[J] := 1;
24        P[J] := -1;
25        J := J + 1;
26     END;
27     V[1] := 1;
28     P[1] := 0;
29  1:
30     V[1] := V[1] + 1;
31     P[1] := P[1] + 1;
32     I := 1;
33  2:
34     IF    P[I] = 1    THEN GOTO 3;
35     L := I + 1;
36     IF    P[I] <> V[L]    THEN GOTO 1;
37     L := I + 1;
38     V[L] := V[1];
39     P[L] := P[L] + 1;
40     IF    I = M    THEN GOTO 4;
41     I := I + 1;
42     GOTO 2;
43  3:
44     L := I + 1;
45     V[L] := V[1];
46     P[L] := 0;
47     IF    I <> M    THEN GOTO 1;
48  4:
49     L := M + 1;
50     IF    P[L] <> N    THEN GOTO 1;
51     Z := V[1];
52     GOTO 6;
53  5:
54     Z := N + 1;
55  6:
56  END.
```

LISTE DES TESTS A L EXECUTION
==========

| | | | | |
|---|---|---|---|---|
| LIGNE : | 17 | M | >= | 0 |
| LIGNE : | 17 | M | <= | 999 |
| LIGNE : | 19 | N | >= | -1 |
| LIGNE : | 41 | I | <= | 998 |
| LIGNE : | 54 | N | <= | 32766 |

```
1     PROGRAM PARTITION-EXCHANGE-SORT;
2
3     LABEL 20,30,40,41,50,60,61,70,71,80,81,82,83,90,99;
4
5     VAR
6          R : ARRAY[1..1000] OF INTEGER;
7          K : ARRAY[1..1000] OF INTEGER;
8          S : ARRAY[1..1000] OF INTEGER;
9          SC: ARRAY[1..1000] OF INTEGER;
10
11         N : INTEGER;
12         A : INTEGER;
13         P : INTEGER;
14         L : INTEGER;
15         Q : INTEGER;
16         D : INTEGER;
17         C : INTEGER;
18         I : INTEGER;
19         J : INTEGER;
20         B : INTEGER;
21         M : INTEGER;
22         R1: INTEGER;
23         K1: INTEGER;
24
25    BEGIN
26       READ(N);
27       B := 1;
28       WHILE   B <= N   DO
29       BEGIN
30          READ(A);
31          R[B] := A;
32          K[B] := A;
33          B := B + 1;
34       END;
35       Q := 0;
36       L := 1;
37       C := N;
38       READ(M);
39    20:
40       B := C - L;
41       IF   B < M   THEN GOTO 80;
42       I := L;
43       J := C;
44       K1 := K[L];
45       R1 := R[L];
46    30:
47       IF   K1 >= K[L]   THEN GOTO 40;
48       J := J - 1;
49       GOTO 30;
50    40:
51       IF   J > 1   THEN GOTO 41;
52       R[I] := R1;
53       GOTO 70;
54    41:
55       R[I] := R[J];
56       I := I + 1;
57    50:
58       IF   K[I] >= K1   THEN GOTO 60;
59       I := I + 1;
60       GOTO 50;
61    60:
62       IF   J <= I   THEN GOTO 61;
63       R[J] := R[I];
64       J := J - 1;
65       GOTO 30;
66    61:
67       R[J] := R1;
68       I := J;
```

```
 70        D := C - I;
 71        B := I - L;
 72        IF    D >= B    THEN GOTO 71;
 73        Q := Q + 1;
 74        S[Q] := I + 1;
 75        SO[Q] := R1;
 76        C := I - 1;
 77        GOTO 20;
 78     71:
 79        Q := Q + 1;
 80        S[Q] := L;
 81        SO[Q] := I - 1;
 82        L := I + 1;
 83        GOTO 20;
 84     80:
 85        D := L + 1;
 86     81:
 87        IF    J > R1    THEN GOTO 90;
 88        K1 := K[J];
 89        R1 := R[J];
 90        I := J - 1;
 91     82:
 92        IF    K[I] <= K1    THEN GOTO 83;
 93        B := I + 1;
 94        R[B] := R[I];
 95        I := I - 1;
 96        GOTO 82;
 97     83:
 98        B := I + 1;
 99        R[B] := R1;
100        J := J + 1;
101        GOTO 81;
102     90:
103        IF    Q = 0    THEN GOTO 99;
104        L := S[Q];
105        C := SO[Q];
106        Q := Q - 1;
107        GOTO 20;
108     99:
109     END.
```

LISTE DES TESTS A L EXECUTION
==========

```
LIGNE :   27   N   <=          1000
LIGNE :   40   M   >=        -32767
LIGNE :   42   L   >=             1
LIGNE :   42   L   <=          1000
LIGNE :   42   Q   <=           999
LIGNE :   42   C   >=        -32767
LIGNE :   48   J   >=        -32767
LIGNE :   51   J   <=          1000
LIGNE :   59   I   <=           999
LIGNE :   85   VARIABLE J  A INITIALISER
LIGNE :   85   J   >=        -32767
LIGNE :   85   VARIABLE R1 A INITIALISER
LIGNE :   88   J   >=             2
LIGNE :   88   J   <=          1000
LIGNE :   93   I   >=             2
LIGNE :  105   L   <=         32766
```

```
 1    PROGRAM LIST-MERGE-SORT;
 2
 3    LABEL 20,30,31,32,51,60,61,62,71,80,81,82,99;
 4
 5    VAR
 6         R1 : ARRAY[1..1000] OF INTEGER;
 7         K1 : ARRAY[1..1000] OF INTEGER;
 8         L1 : ARRAY[0..1000] OF INTEGER;
 9         N : INTEGER;
10         I : INTEGER;
11         A : INTEGER;
12         U : INTEGER;
13         T : INTEGER;
14         P : INTEGER;
15         Q : INTEGER;
16         E : INTEGER;
17         M : INTEGER;
18         S : INTEGER;
19
20    BEGIN
21       READ(N);
22       U := 1;
23       WHILE   U <= N   DO
24       BEGIN
25          READ(A);
26          R1[U] := A;
27          K1[U] := A;
28          U := U + 1;
29       END;
30       M := N + 1;
31       L1[0] := 1;
32       L1[M] := 2;
33       L1[N] := 0;
34       M := N - 1;
35       L1[M] := 0;
36       U := N - 2;
37       I := 1;
38       READ(E);
39       WHILE   I <= U   DO
40       BEGIN
41          L1[I] := 0 - I;
42          L1[I] := L1[I] - 2;
43          I := I + 1;
44       END;
45    20:
46       S := 0;
47       T := N + 1;
48       P := L1[S];
49       Q := L1[T];
50       IF   Q = 0   THEN GOTO 99;
51    30:
52       IF   K1[P] > K1[Q]   THEN GOTO 60;
53       IF   E > 0   THEN GOTO 31;
54       L1[S] := 0 - P;
55       GOTO 32;
56    31:
57       L1[S] := P;
58    32:
59       S := P;
60       P := L1[P];
61       IF   P > 0   THEN GOTO 30;
62       L1[S] := Q;
63       S := T;
64       T := Q;
65    51:
66       Q := L1[Q];
67       IF   Q > 0   THEN GOTO 51;
68       GOTO 80;
```

```
70        IF    E > 0    THEN GOTO 61;
71        L1[S! := 0 - Q;
72        GOTO 62;
73     61:
74        L1[S! := Q;
75     62:
76        S := Q;
77        Q := L1[Q!;
78        IF    Q > 0    THEN GOTO 30;
79        L1[S! := P;
80        S := T;
81        T := P;
82     71:
83        P := L1[P!;
84        IF    P > 0    THEN GOTO 71;
85     80:
86        P := 0 - P;
87        Q := 0 - Q;
88        IF    Q <> 0    THEN GOTO 30;
89        IF    E > 0    THEN GOTO 81;
90        L1[S! := P;
91        GOTO 82;
92     81:
93        L1[S! := 0-P;
94     82:
95        L1[T! := 0;
96        GOTO 20;
97     99:
98     END.
```

LISTE DES TESTS A L EXECUTION

| | | | | |
|---|---|---|---|---|
| LIGNE : | 22 | N | >= | 1 |
| LIGNE : | 22 | N | <= | 999 |
| LIGNE : | 50 | Q | >= | 0 |
| LIGNE : | 50 | Q | <= | 1000 |
| LIGNE : | 52 | P | >= | 1 |
| LIGNE : | 52 | P | <= | 1000 |
| LIGNE : | 53 | T | >= | 2 |
| LIGNE : | 53 | S | <= | 0 |
| LIGNE : | 61 | P | >= | -32767 |
| LIGNE : | 61 | P | <= | 1000 |
| LIGNE : | 67 | Q | >= | -1000 |
| LIGNE : | 78 | Q | >= | -1000 |
| LIGNE : | 78 | Q | <= | 1000 |
| LIGNE : | 84 | P | >= | -32767 |

```
1     PROGRAM P.G.C.D.-TABLEAU;
2
3     LABEL 1,2,3,4,5,6;
4
5     VAR
6          S1 : ARRAY[1..1000] OF INTEGER;
7          K : INTEGER;
8          I : INTEGER;
9          Z : INTEGER;
10         S : INTEGER;
11         FC : INTEGER;
12         J : INTEGER;
13         N : INTEGER;
14         A : INTEGER;
15
16    BEGIN
17       READ(N);
18       K := 1;
19       WHILE   K <= N   DO
20       BEGIN
21          READ(A);
22          S1[K] := A;
23          K := K + 1;
24       END;
25    1:
26       FO := S1[1];
27       S := 0;
28       I := 2;
29       J := 0;
30    2:
31       IF   I > N   THEN GOTO 5;
32       IF   S1[I] > FO   THEN GOTO 3;
33       IF   S1[I] <= S   THEN GOTO 4;
34       S := S1[I];
35       GOTO 4;
36    3:
37       S := FO;
38       FC := S1[I];
39       J := I;
40    4:
41       I := I - 1;      % %
42       GOTO 2;
43    5:
44       IF   S = 0   THEN GOTO 6;
45       S1[J] := FO - S;
46       GOTO 1;
47    6:
48       Z := FO;
49    END.
```

A LA LIGNE : 17

```
S1 :: N-INIT
K  :: N-INIT
I  :: N-INIT
Z  :: N-INIT
S  :: N-INIT
FO :: N-INIT
J  :: N-INIT
N  :: N-INIT
A  :: N-INIT
```

A LA LIGNE : 18

```
S1 :: N-INIT
K  :: N-INIT
I  :: N-INIT
Z  :: N-INIT
S  :: N-INIT
FO :: N-INIT
J  :: N-INIT
N  ::   INIT  ,   (-32768 , 32767 )
A  :: N-INIT
```

A LA LIGNE : 19

```
S1 ::   INIT-?,   (-32768 , 32767 )
K  ::   INIT  ,   (     1 ,  1001 )
I  :: N-INIT
Z  :: N-INIT
S  :: N-INIT
FO :: N-INIT
J  :: N-INIT
N  ::   INIT  ,   (-32768 , 32767 )
A  ::   INIT-?,   (-32768 , 32767 )
```

A LA LIGNE : 21

```
S1 ::   INIT-?,   (-32768 , 32767 )
K  ::   INIT  ,   (     1 ,  1001 )
I  :: N-INIT
Z  :: N-INIT
S  :: N-INIT
FO :: N-INIT
J  :: N-INIT
N  ::   INIT  ,   (     1 , 32767 )
A  ::   INIT-?,   (-32768 , 32767 )
```

```
A LA LIGNE  :   22
................

       S1 ::     INIT-?,     (-32768 , 32767 )
       K  ::     INIT  ,     (     1 ,  1000 )
       I  ::  N-INIT
       Z  ::  N-INIT
       S  ::  N-INIT
       FO ::  N-INIT
       J  ::  N-INIT
       N  ::     INIT  ,     (     1 , 32767 )
       A  ::     INIT  ,     (-32768 , 32767 )

A LA LIGNE  :   23
................

       S1 ::     INIT-?,     (-32768 , 32767 )
       K  ::     INIT  ,     (     1 ,  1000 )
       I  ::  N-INIT
       Z  ::  N-INIT
       S  ::  N-INIT
       FO ::  N-INIT
       J  ::  N-INIT
       N  ::     INIT  ,     (     1 , 32767 )
       A  ::     INIT  ,     (-32768 , 32767 )

A LA LIGNE  :   26
................

       S1 ::     INIT-?,     (-32768 , 32767 )
       K  ::     INIT  ,     (     1 , 32767 )
       I  ::     INIT-?,     (-32767 ,     2 )
       Z  ::  N-INIT
       S  ::     INIT-?,     (-32768 , 32767 )
       FO ::     INIT-?,     (-32768 , 32767 )
       J  ::     INIT-?,     (     1 ,  1000 )
       N  ::     INIT  ,     (-32768 , 32766 )
       A  ::     INIT-?,     (-32768 , 32767 )

A LA LIGNE  :   27
................

       S1 ::     INIT-?,     (-32768 , 32767 )
       K  ::     INIT  ,     (     1 , 32767 )
       I  ::     INIT-?,     (-32767 ,     2 )
       Z  ::  N-INIT
       S  ::     INIT-?,     (-32768 , 32767 )
       FO ::     INIT  ,     (-32768 , 32767 )
       J  ::     INIT-?,     (     1 ,  1000 )
       N  ::     INIT  ,     (-32768 , 32766 )
       A  ::     INIT-?,     (-32768 , 32767 )

A LA LIGNE  :   28
................

       S1 ::     INIT-?,     (-32768 , 32767 )
       K  ::     INIT  ,     (     1 , 32767 )
       I  ::     INIT-?,     (-32767 ,     2 )
       Z  ::  N-INIT
       S  ::     INIT  ,     (     0 ,     0 )
       FO ::     INIT  ,     (-32768 , 32767 )
       J  ::     INIT-?,     (     1 ,  1000 )
       N  ::     INIT  ,     (-32768 , 32766 )
```

A LA LIGNE : 29
.....

```
S1 ::     INIT-?,    (-32768 , 32767 )
K  ::     INIT  ,    (     1 , 32767 )
I  ::     INIT  ,    (     2 ,     2 )
Z  :: N-INIT
S  ::     INIT  ,    (     0 ,     0 )
FO ::     INIT  ,    (-32768 , 32767 )
J  ::     INIT-?,    (     1 ,  1000 )
N  ::     INIT  ,    (-32768 , 32766 )
A  ::     INIT-?,    (-32768 , 32767 )
```

A LA LIGNE : 31
.....

```
S1 ::     INIT-?,    (-32768 , 32767 )
K  ::     INIT  ,    (     1 , 32767 )
I  ::     INIT  ,    (     0 ,     2 )
Z  :: N-INIT
S  ::     INIT  ,    (-32768 , 32767 )
FO ::     INIT  ,    (-32768 , 32767 )
J  ::     INIT  ,    (     0 , 32767 )
N  ::     INIT  ,    (-32768 , 32766 )
A  ::     INIT-?,    (-32768 , 32767 )
```

A LA LIGNE : 32
.....

```
S1 ::     INIT-?,    (-32768 , 32767 )
K  ::     INIT  ,    (     1 , 32767 )
I  ::     INIT  ,    (     0 ,     2 )
Z  :: N-INIT
S  ::     INIT  ,    (-32768 , 32767 )
FO ::     INIT  ,    (-32768 , 32767 )
J  ::     INIT  ,    (     0 , 32767 )
N  ::     INIT  ,    (     0 , 32766 )
A  ::     INIT-?,    (-32768 , 32767 )
```

A LA LIGNE : 33
.....

```
S1 ::     INIT-?,    (-32768 , 32767 )
K  ::     INIT  ,    (     1 , 32767 )
I  ::     INIT  ,    (     1 ,     2 )
Z  :: N-INIT
S  ::     INIT  ,    (-32768 , 32767 )
FO ::     INIT  ,    (-32768 , 32767 )
J  ::     INIT  ,    (     0 , 32767 )
N  ::     INIT  ,    (-32768 , 32766 )
A  ::     INIT-?,    (-32768 , 32767 )
```

A LA LIGNE : 34
.....

```
S1 ::     INIT-?,    (-32768 , 32767 )
K  ::     INIT  ,    (     1 , 32767 )
I  ::     INIT  ,    (     1 ,     2 )
Z  :: N-INIT
S  ::     INIT  ,    (-32768 , 32766 )
FO ::     INIT  ,    (-32768 , 32767 )
J  ::     INIT  ,    (     0 , 32767 )
N  ::     INIT  ,    (-32768 , 32766 )
A  ::     INIT-?,    (-32768 , 32767 )
```

```
A LA LIGNE  :   37
.................

        S1  ::     INIT-?,     (-32768 , 32767 )
        K   ::     INIT   ,    (     1 , 32767 )
        I   ::     INIT   ,    (     1 ,     2 )
        Z   ::  N-INIT
        S   ::     INIT   ,    (-32768 , 32767 )
        FO  ::     INIT   ,    (-32768 , 32766 )
        J   ::     INIT   ,    (     0 , 32767 )
        N   ::     INIT   ,    (-32768 , 32766 )
        A   ::     INIT-?,     (-32768 , 32767 )

A LA LIGNE  :   38
.................

        S1  ::     INIT-?,     (-32768 , 32767 )
        K   ::     INIT   ,    (     1 , 32767 )
        I   ::     INIT   ,    (     1 ,     2 )
        Z   ::  N-INIT
        S   ::     INIT   ,    (-32768 , 32766 )
        FO  ::     INIT   ,    (-32768 , 32766 )
        J   ::     INIT   ,    (     0 , 32767 )
        N   ::     INIT   ,    (-32768 , 32766 )
        A   ::     INIT-?,     (-32768 , 32767 )

A LA LIGNE  :   39
.................

        S1  ::     INIT-?,     (-32768 , 32767 )
        K   ::     INIT   ,    (     1 , 32767 )
        I   ::     INIT   ,    (     1 ,     2 )
        Z   ::  N-INIT
        S   ::     INIT   ,    (-32768 , 32766 )
        FO  ::     INIT   ,    (-32768 , 32767 )
        J   ::     INIT   ,    (     0 , 32767 )
        N   ::     INIT   ,    (-32768 , 32766 )
        A   ::     INIT-?,     (-32768 , 32767 )

A LA LIGNE  :   41
.................

        S1  ::     INIT-?,     (-32768 , 32767 )
        K   ::     INIT   ,    (     1 , 32767 )
        I   ::     INIT   ,    (     1 ,     2 )
        Z   ::  N-INIT
        S   ::     INIT   ,    (-32768 , 32767 )
        FO  ::     INIT   ,    (-32768 , 32767 )
        J   ::     INIT   ,    (     0 , 32767 )
        N   ::     INIT   ,    (-32768 , 32766 )
        A   ::     INIT-?,     (-32768 , 32767 )

A LA LIGNE  :   44
.................

        S1  ::     INIT-?,     (-32768 , 32767 )
        K   ::     INIT   ,    (     1 , 32767 )
        I   ::     INIT   ,    (     0 ,     2 )
        Z   ::  N-INIT
        S   ::     INIT   ,    (-32768 , 32767 )
        FO  ::     INIT   ,    (-32768 , 32767 )
        J   ::     INIT   ,    (     0 , 32767 )
        N   ::     INIT   ,    (-32768 ,     1 )
```

```
A LA LIGNE  :   45
................

       S1  ::     INIT-?,     (-32768 , 32767 )
       K   ::     INIT  ,     (     1 , 32767 )
       I   ::     INIT  ,     (     0 ,     2 )
       Z   ::   N-INIT
       S   ::     INIT  ,     (-32768 , 32767 )
       FO  ::     INIT  ,     (-32768 , 32767 )
       J   ::     INIT  ,     (     1 ,  1000 )
       N   ::     INIT  ,     (-32768 ,     1 )
       A   ::     INIT-?,     (-32768 , 32767 )


A LA LIGNE  :   48
................

       S1  ::     INIT-?,     (-32768 , 32767 )
       K   ::     INIT  ,     (     1 , 32767 )
       I   ::     INIT  ,     (     0 ,     2 )
       Z   ::   N-INIT
       S   ::     INIT  ,     (     0 ,     0 )
       FO  ::     INIT  ,     (-32768 , 32767 )
       J   ::     INIT  ,     (     0 , 32767 )
       N   ::     INIT  ,     (-32768 ,     1 )
       A   ::     INIT-?,     (-32768 , 32767 )


A LA LIGNE  :   49
................

       S1  ::     INIT-?,     (-32768 , 32767 )
       K   ::     INIT  ,     (     1 , 32767 )
       I   ::     INIT  ,     (     0 ,     2 )
       Z   ::     INIT  ,     (-32768 , 32767 )
       S   ::     INIT  ,     (     0 ,     0 )
       FO  ::     INIT  ,     (-32768 , 32767 )
       J   ::     INIT  ,     (     0 , 32767 )
       N   ::     INIT  ,     (-32768 ,     1 )
       A   ::     INIT-?,     (-32768 , 32767 )
```

CONTEXTE GENERAL

A LA LIGNE : 17

```
S1 :: N-INIT
K  :: N-INIT
I  :: N-INIT
Z  :: N-INIT
S  :: N-INIT
FO :: N-INIT
J  :: N-INIT
N  :: N-INIT
A  :: N-INIT
```

A LA LIGNE : 18

```
S1 :: N-INIT
K  :: N-INIT
I  :: N-INIT
Z  :: N-INIT
S  :: N-INIT
FO :: N-INIT
J  :: N-INIT
N  ::    INIT  ,    (-32768 ,     1 )
A  :: N-INIT
```

A LA LIGNE : 19

```
S1 ::    INIT-?,    (-32768 , 32767 )
K  ::    INIT  ,    (     1 ,  1001 )
I  :: N-INIT
Z  :: N-INIT
S  :: N-INIT
FO :: N-INIT
J  :: N-INIT
N  ::    INIT  ,    (-32768 ,     1 )
A  ::    INIT-?,    (-32768 , 32767 )
```

A LA LIGNE : 21

```
S1 ::    INIT-?,    (-32768 , 32767 )
K  ::    INIT  ,    (     1 ,     1 )
I  :: N-INIT
Z  :: N-INIT
S  :: N-INIT
FO :: N-INIT
J  :: N-INIT
N  ::    INIT  ,    (     1 ,     1 )
A  ::    INIT-?,    (-32768 , 32767 )
```

A LA LIGNE : 22

```
S1 ::    INIT-?,    (-32768 , 32767 )
K  ::    INIT  ,    (     1 ,     1 )
I  :: N-INIT
Z  :: N-INIT
S  :: N-INIT
FO :: N-INIT
J  :: N-INIT
N  ::    INIT  ,    (     1 ,     1 )
```

```
A LA LIGNE  :   23
.................

       S1 ::     INIT-?,     (-32768 , 32767 )
       K  ::     INIT  ,     (     1 ,     1 )
       I  ::  N-INIT
       Z  ::  N-INIT
       S  ::  N-INIT
       FO ::  N-INIT
       J  ::  N-INIT
       N  ::     INIT  ,     (     1 ,     1 )
       A  ::     INIT  ,     (-32768 , 32767 )

A LA LIGNE  :   26
.................

       S1 ::     INIT-?,     (-32768 , 32767 )
       K  ::     INIT  ,     (     1 ,  1001 )
       I  ::  N-INIT
       Z  ::  N-INIT
       S  ::  N-INIT
       FO ::  N-INIT
       J  ::  N-INIT
       N  ::     INIT  ,     (-32768 ,     1 )
       A  ::     INIT-?,     (-32768 , 32767 )

A LA LIGNE  :   27
.................

       S1 ::     INIT-?,     (-32768 , 32767 )
       K  ::     INIT  ,     (     1 ,  1001 )
       I  ::  N-INIT
       Z  ::  N-INIT
       S  ::  N-INIT
       FO ::     INIT  ,     (-32768 , 32767 )
       J  ::  N-INIT
       N  ::     INIT  ,     (-32768 ,     1 )
       A  ::     INIT-?,     (-32768 , 32767 )

A LA LIGNE  :   28
.................

       S1 ::     INIT-?,     (-32768 , 32767 )
       K  ::     INIT  ,     (     1 ,  1001 )
       I  ::  N-INIT
       Z  ::  N-INIT
       S  ::     INIT  ,     (     0 ,     0 )
       FO ::     INIT  ,     (-32768 , 32767 )
       J  ::  N-INIT
       N  ::     INIT  ,     (-32768 ,     1 )
       A  ::     INIT-?,     (-32768 , 32767 )

A LA LIGNE  :   29
.................

       S1 ::     INIT-?,     (-32768 , 32767 )
       K  ::     INIT  ,     (     1 ,  1001 )
       I  ::     INIT  ,     (     2 ,     2 )
       Z  ::  N-INIT
       S  ::     INIT  ,     (     0 ,     0 )
       FO ::     INIT  ,     (-32768 , 32767 )
       J  ::  N-INIT
       N  ::     INIT  ,     (-32768 ,     1 )
       A  ::     INIT-?,     (-32768 , 32767 )
```

A LA LIGNE : 31
.....

```
S1 ::     INIT-?,     (-32768 , 32767 )
K  ::     INIT  ,     (     1 ,  1001 )
I  ::     INIT  ,     (     2 ,     2 )
Z  ::  N-INIT
S  ::     INIT  ,     (     0 ,     0 )
FO ::     INIT  ,     (-32768 , 32767 )
J  ::     INIT  ,     (     0 ,     0 )
N  ::     INIT  ,     (-32768 ,     1 )
A  ::     INIT-?,     (-32768 , 32767 )
```

A LA LIGNE : 32
.....

PCINT INACCESSIBLE A L EXECUTICN
CU
A PARTIR DE CE POINT,L EXECUTICN NE SE TERMINE PAS OU CONDUIT A UNE ERRE

ERREURS POSSIBLES

. DEBORDEMENT LOGIQUE

A LA LIGNE : 33
.....

PCINT INACCESSIBLE A L EXECUTION
CU
A PARTIR DE CE POINT,L EXECUTION NE SE TERMINE PAS OU CONDUIT A UNE ERRI

ERREURS POSSIBLES

. DEBORDEMENT LOGIQUE

A LA LIGNE : 34
.....

PCINT INACCESSIBLE A L EXECUTION
CU
A PARTIR DE CE POINT,L EXECUTION NE SE TERMINE PAS OU CONDUIT A UNE ERRI

ERREURS POSSIBLES

. DEBORDEMENT LOGIQUE

A LA LIGNE : 37
.....

PCINT INACCESSIBLE A L EXECUTION
CU
A PARTIR DE CE POINT,L EXECUTION NE SE TERMINE PAS OU CONDUIT A UNE ERRI

ERREURS POSSIBLES

. DEBORDEMENT LOGIQUE

```
LA LIGNE  :   38
.................

 POINT INACCESSIBLE A L EXECUTION
       OU
 A PARTIR DE CE POINT,L EXECUTION NE SE TERMINE PAS OU CONDUIT A UNE ERREUR

     ERREURS POSSIBLES

          . DEBORDEMENT LOGIQUE


LA LIGNE  :   39
.................

 POINT INACCESSIBLE A L EXECUTION
       OU
 A PARTIR DE CE POINT,L EXECUTION NE SE TERMINE PAS OU CONDUIT A UNE ERREUR

     ERREURS POSSIBLES

          . DEBORDEMENT LOGIQUE


LA LIGNE  :   41
.................

 POINT INACCESSIBLE A L EXECUTION
       OU
 A PARTIR DE CE POINT,L EXECUTION NE SE TERMINE PAS OU CONDUIT A UNE ERREUR

     ERREURS POSSIBLES

          . DEBORDEMENT LOGIQUE


LA LIGNE  :   44
.................

     S1 ::     INIT-?,     (-32768 , 32767 )
     K  ::     INIT  ,     (     1 ,  1001 )
     I  ::     INIT  ,     (     2 ,     2 )
     Z  ::   N-INIT
     S  ::     INIT  ,     (     0 ,     0 )
     FO ::     INIT  ,     (-32768 , 32767 )
     J  ::     INIT  ,     (     0 ,     0 )
     N  ::     INIT  ,     (-32768 ,     1 )
     A  ::     INIT-?,     (-32768 , 32767 )

LA LIGNE  :   45
.................

 POINT INACCESSIBLE A L EXECUTION
       OU
 A PARTIR DE CE POINT,L EXECUTION NE SE TERMINE PAS OU CONDUIT A UNE ERREUR

     ERREURS POSSIBLES

          . DEBORDEMENT LOGIQUE
```

```
A LA LIGNE  :   48
................

        S1 ::     INIT-?,     (-32768 , 32767 )
        K  ::     INIT  ,     (     1 ,  1001 )
        I  ::     INIT  ,     (     2 ,     2 )
        Z  ::   N-INIT
        S  ::     INIT  ,     (     0 ,     0 )
        FO ::     INIT  ,     (-32768 , 32767 )
        J  ::     INIT  ,     (     0 ,     0 )
        N  ::     INIT  ,     (-32768 ,     1 )
        A  ::     INIT-?,     (-32768 , 32767 )

A LA LIGNE  :   49
................

        S1 ::     INIT-?,     (-32768 , 32767 )
        K  ::     INIT  ,     (     1 ,  1001 )
        I  ::     INIT  ,     (     2 ,     2 )
        Z  ::     INIT  ,     (-32768 , 32767 )
        S  ::     INIT  ,     (     0 ,     0 )
        FO ::     INIT  ,     (-32768 , 32767 )
        J  ::     INIT  ,     (     0 ,     0 )
        N  ::     INIT  ,     (-32768 ,     1 )
        A  ::     INIT-?,     (-32768 , 32767 )

                LISTE DES TESTS A L EXECUTION
                =============================


                LIGNE :  18   N  <=          1
```

```
1     PROGRAM BINARY-SEARCH;
2
3     LABEL 20,40,50,97,98,99;
4
5     VAR
6          K1 : ARRAY[1..1000] OF INTEGER;
7          N : INTEGER;
8          L : INTEGER;
9          A : INTEGER;
10         U : INTEGER;
11         M : INTEGER;
12         I : INTEGER;
13         J : INTEGER;
14         K : INTEGER;
15         P : INTEGER;
16         R : 0..1;
17
18    BEGIN
19       READ(N);
20       J := 1;
21       WHILE   J <= N   DO
22       BEGIN
23          READ(A);
24          K1[J] := A;
25          J := J + 1;
26       END;
27       L := 1;
28       U := N;
29    20:
30       IF   U < L   THEN GOTO 98;
31       P := L + U;
32       I := P DIV 2;
33       IF   K < K1[I]   THEN GOTO 40;
34       IF   K > K1[I]   THEN GOTO 50;
35       GOTO 97;
36    40:
37       U := I - 1;
38       GOTO 20;
39    50:
40       L := I + 1;
41       GOTO 20;
42    97:
43       R := 1;
44       GOTO 99;
45    98:
46       R := 0;
47    99:
48    END.
```

LYSE PRELIMINAIRE
---

A LA LIGNE : 19
.....

```
K1 :: N-INIT
N  :: N-INIT
L  :: N-INIT
A  :: N-INIT
U  :: N-INIT
M  :: N-INIT
I  :: N-INIT
J  :: N-INIT
K  :: N-INIT
P  :: N-INIT
R  :: N-INIT
```

A LA LIGNE : 20
.....

```
K1 :: N-INIT
N  ::   INIT  ,    (-32768 , 32767 )
L  :: N-INIT
A  :: N-INIT
U  :: N-INIT
M  :: N-INIT
I  :: N-INIT
J  :: N-INIT
K  :: N-INIT
P  :: N-INIT
R  :: N-INIT
```

A LA LIGNE : 21
.....

```
K1 ::    INIT-?,    (-32768 , 32767 )
N  ::    INIT  ,    (-32768 , 32767 )
L  :: N-INIT
A  ::    INIT-?,    (-32768 , 32767 )
U  :: N-INIT
M  :: N-INIT
I  :: N-INIT
J  ::    INIT  ,    (     1 ,  1001 )
K  :: N-INIT
P  :: N-INIT
R  :: N-INIT
```

A LA LIGNE : 23
.....

```
K1 ::    INIT-?,    (-32768 , 32767 )
N  ::    INIT  ,    (     1 , 32767 )
L  :: N-INIT
A  ::    INIT-?,    (-32768 , 32767 )
U  :: N-INIT
M  :: N-INIT
I  :: N-INIT
J  ::    INIT  ,    (     1 ,  1001 )
K  :: N-INIT
P  :: N-INIT
R  :: N-INIT
```

```
LA LIGNE  :   24
.................

    K1 ::     INIT-?,     (-32768 , 32767 )
    N  ::     INIT  ,     (     1 , 32767 )
    L  :: N-INIT
    A  ::     INIT  ,     (-32768 , 32767 )
    U  :: N-INIT
    M  :: N-INIT
    I  :: N-INIT
    J  ::     INIT  ,     (     1 ,  1000 )
    K  :: N-INIT
    P  :: N-INIT
    R  :: N-INIT


LA LIGNE  :   25
.................

    K1 ::     INIT-?,     (-32768 , 32767 )
    N  ::     INIT  ,     (     1 , 32767 )
    L  :: N-INIT
    A  ::     INIT  ,     (-32768 , 32767 )
    U  :: N-INIT
    M  :: N-INIT
    I  :: N-INIT
    J  ::     INIT  ,     (     1 ,  1000 )
    K  :: N-INIT
    P  :: N-INIT
    R  :: N-INIT


LA LIGNE  :   27
.................

    K1 ::     INIT-?,     (-32768 , 32767 )
    N  ::     INIT  ,     (-32768 ,  1000 )
    L  :: N-INIT
    A  ::     INIT-?,     (-32768 , 32767 )
    U  :: N-INIT
    M  :: N-INIT
    I  :: N-INIT
    J  ::     INIT  ,     (     1 ,  1001 )
    K  :: N-INIT
    P  :: N-INIT
    R  :: N-INIT


LA LIGNE  :   28
.................

    K1 ::     INIT-?,     (-32768 , 32767 )
    N  ::     INIT  ,     (-32768 ,  1000 )
    L  ::     INIT  ,     (     1 ,     1 )
    A  ::     INIT-?,     (-32768 , 32767 )
    U  :: N-INIT
    M  :: N-INIT
    I  :: N-INIT
    J  ::     INIT  ,     (     1 ,  1001 )
    K  :: N-INIT
    P  :: N-INIT
    R  :: N-INIT
```

```
LIGNE  :   30
.............

  K1 ::     INIT-?,     (-32768 , 32767 )
  N  ::     INIT  ,     (-32768 , 32766 )
  L  ::     INIT  ,     (     1 ,     1 )
  A  ::     INIT-?,     (-32768 , 32767 )
  U  ::     INIT  ,     (-32768 , 32766 )
  M  ::   N-INIT
  I  ::   N-INIT
  J  ::     INIT  ,     (     1 ,  1001 )
  K  ::   N-INIT
  P  ::   N-INIT
  R  ::   N-INIT


LIGNE  :   31
.............

  K1 ::     INIT-?,     (-32768 , 32767 )
  N  ::     INIT  ,     (-32768 , 32766 )
  L  ::     INIT  ,     (     1 ,     1 )
  A  ::     INIT-?,     (-32768 , 32767 )
  U  ::     INIT  ,     (     1 , 32766 )
  M  ::   N-INIT
  I  ::   N-INIT
  J  ::     INIT  ,     (     1 ,  1001 )
  K  ::   N-INIT
  P  ::   N-INIT
  R  ::   N-INIT


LIGNE  :   32
.............

  K1 ::     INIT-?,     (-32768 , 32767 )
  N  ::     INIT  ,     (-32768 , 32766 )
  L  ::     INIT  ,     (     1 ,     1 )
  A  ::     INIT-?,     (-32768 , 32767 )
  U  ::     INIT  ,     (     1 , 32766 )
  M  ::   N-INIT
  I  ::   N-INIT
  J  ::     INIT  ,     (     1 ,  1001 )
  K  ::   N-INIT
  P  ::     INIT  ,     (     2 , 32767 )
  R  ::   N-INIT


LIGNE  :   33
.............

  K1 ::     INIT-?,     (-32768 , 32767 )
  N  ::     INIT  ,     (-32768 , 32766 )
  L  ::     INIT  ,     (     1 ,     1 )
  A  ::     INIT-?,     (-32768 , 32767 )
  U  ::     INIT  ,     (     1 , 32766 )
  M  ::   N-INIT
  I  ::     INIT  ,     (     1 ,  1000 )
  J  ::     INIT  ,     (     1 ,  1001 )
  K  ::   N-INIT
  P  ::     INIT  ,     (     2 , 32767 )
  R  ::   N-INIT
```

A LIGNE : 34
.....

POINT INACCESSIBLE A L EXECUTION
OU
A PARTIR DE CE POINT,L EXECUTION NE SE TERMINE PAS OU CONDUIT A UNE ERREUR

ERREURS POSSIBLES

. UTILISATION D UNE VARIABLE NON INITIALISEE

A LIGNE : 37
.....

POINT INACCESSIBLE A L EXECUTION
OU
A PARTIR DE CE POINT,L EXECUTION NE SE TERMINE PAS OU CONDUIT A UNE ERREUR

ERREURS POSSIBLES

. UTILISATION D UNE VARIABLE NON INITIALISEE

A LIGNE : 40
.....

POINT INACCESSIBLE A L EXECUTION
OU
A PARTIR DE CE POINT,L EXECUTION NE SE TERMINE PAS OU CONDUIT A UNE ERREUR

ERREURS POSSIBLES

. UTILISATION D UNE VARIABLE NON INITIALISEE

A LIGNE : 43
.....

POINT INACCESSIBLE A L EXECUTION
OU
A PARTIR DE CE POINT,L EXECUTION NE SE TERMINE PAS OU CONDUIT A UNE ERREUR

ERREURS POSSIBLES

. UTILISATION D UNE VARIABLE NON INITIALISEE

A LA LIGNE : 46
.....

```
K1 ::     INIT-?,     (-32768 , 32767 )
N  ::     INIT  ,     (-32768 , 32766 )
L  ::     INIT  ,     (     1 ,     1 )
A  ::     INIT-?,     (-32768 , 32767 )
U  ::     INIT  ,     (-32768 ,     0 )
M  ::  N-INIT
I  ::  N-INIT
J  ::     INIT  ,     (     1 ,  1001 )
K  ::  N-INIT
P  ::  N-INIT
R  ::  N-INIT
```

A LA LIGNE : 48
.....

```
K1 ::     INIT-?,     (-32768 , 32767 )
N  ::     INIT  ,     (-32768 , 32766 )
L  ::     INIT  ,     (     1 ,     1 )
A  ::     INIT-?,     (-32768 , 32767 )
U  ::     INIT  ,     (-32768 ,     0 )
M  ::  N-INIT
I  ::  N-INIT
J  ::     INIT  ,     (     1 ,  1001 )
K  ::  N-INIT
P  ::  N-INIT
R  ::     INIT  ,     (     0 ,     0 )
```

# VI Etude du graphe

## 1. Introduction

Dans les chapitres, nous avons supposé que cette étude devait nous donner des informations susceptibles de servir l'analyse sémantique. Les deux informations que nous aimerions avoir sont :

- comment choisir les points de coupure ?
- comment parcourir un graphe ? ie. quels arcs suivre ?

Nous avons vu que les points de coupure sont les noeuds du graphe auxquels sera appliquée la fonction d'élargissement (et par voie de conséquence, la fonction de rétrécissement) ; remarquant que l'élargissement conduit à une perte de précision sur le résultat, il est nécessaire d'obtenir un ensemble de points de coupure le plus petit possible.

La parcours d'un graphe doit se faire indépendamment des instructions et devra tenir compte d'une série de contraintes concernant le graphe ainsi que du fait que le graphe n'est pas forcément "bien construit".

Ces deux problèmes ont en fait un point commun : l'algorithme de parcours en profondeur d'un graphe. Cette idée nous a amenés à penser qu'un seul programme pouvait résoudre ces problèmes simultanément.

La première partie va nous rappeler ce qu'est le parcours en profondeur d'un graphe, suivie d'un paragraphe consacré à la recherche des points de coupure dans un graphe réductible. Ce résultat sera étendu dans un troisième paragraphe aux graphes irréductibles.

Le parcours en profondeur va également nous permettre de construire des composantes fortement connexes dans un graphe et la décomposition d'un graphe en composantes fera l'objet d'un quatrième et cinquième paragraphe.

Notre but sera de regrouper tous ces résultats en un unique algorithme, de préférence itératif et nous en parlerons à la fin de ce chapitre.

## 2. Parcours avec priorité à la profondeur

Ce paragraphe résume les travaux de R. Tarjan [72]

a) définitions

. un graphe G est défini par A x N où A est un ensemble d'arcs et N un ensemble de noeuds ; nous ne considèrerons que les graphes orientés ;

. un chemin $m \xrightarrow{*} n$ est une suite de noeuds et d'arcs qui part de m pour arriver à n ;

. un chemin est simple si tous les noeuds sont distincts ;

. un chemin $n \xrightarrow{*} n$ est dit fermé ;

. un chemin c : $n \xrightarrow{*} n$ est un cycle si tous les arcs sont distincts et si le seul noeud à apparaître deux fois (exactement deux fois) dans C est n.

b) parcours en profondeur

Soit G un graphe que nous aimerions explorer ; à l'état initial, aucun noeud n'est exploré ; nous commençons le parcours en un noeud quelconque et choisissons un arc que nous allons suivre ; en traversant cet arc, cela nous conduit à un nouveau noeud ; nous continuons ainsi et à chaque étape, nous choisissons un arc partant d'un noeud déjà atteint et nous suivons cet arc qui conduit à noeud qui est soit nouveau, soit déjà atteint.

Quand nous suivons un arc partant d'un noeud, nous choisirons un noeud non encore atteint (s'il existe) et commencerons une nouvelle exploration à partir de ce point ; éventuellement, nous traverserons tous les arcs de G exactement une fois chacun ; un tel procédé est appelé "parcours" de G.

Considérons maintenant la règle suivante : quand il s'agit de suivre un arc, nous choisirons toujours un arc partant du dernier noeud atteint qui a encore des arcs inexplorés ; une exploration qui utilise cette technique est appelée "parcours avec priorité à la profondeur" de G.

Ci-dessous l'algorithme de parcours en profondeur ; le résultat de ce programme est un tableau 'nombre' qui fait correspondre à chaque noeud un numéro d'ordre ; notons que les numéros d'ordre dépendent de la façon dont on choisit entre plusieurs arcs inexplorés issus d'un même noeud celui qui sera traversé en premier.

c) algorithme de parcours en profondeur

```
procédure  dfs  (n) ;
début
     i : = i + 1
     nombre [ n ] : = i
     pour tout m successeur de n faire
             si m n'a pas encore été visité alors  dfs (m)  fsi
     fpour
fin
début   /* programme principal */
     i : = 0
     dfs (no)  /* no est le noeud initial */
fin
```

3. Recherche des ensembles minimaux de coupure dans un graphe réductible

a) introduction

Les graphes finis orientés qui ne contiennent pas de cycles, ne peuvent être décrits que par des chemins finis, chacun d'eux contenant un nombre fini d'arcs ; et, de ce fait, l'analyse des chemins de ce graphe se fait habituellement dans le sens d'orientation des arcs.

L'analyse devient différente lors de la présence des cycles, car le nombre et la longueur de chemins ne sont plus finis; dans la plupart des cas, l'analyse des chemins de graphes quelconques peut être ramenée à celle de graphes sans cycle en choisissant un sous-ensemble approprié de noeuds - appelés noeuds ou points de coupure - de telle sorte que tout cycle du graphe contienne au moins un de ces noeuds ; les points de coupure décomposent le graphe d'une façon naturelle en composantes ne contenant aucun cycle et pouvant être analysées séparément.

Nous allons résumer ici l'étude faite par A. Shamir [ 79 ]

b) définitions

. un graphe qui ne contient aucun cycle est appelé "dag" (directed acyclic graph) ;

. un parcours en profondeur d'un graphe est une façon d'explorer ce graphe en utilisant une pile ;

. le parcours en profondeur définit deux ordres possibles sur les noeuds :
  - préordre : ordre dans lequel les noeuds ont été empilés pendant le parcours ;
  - postordre : ordre dans lequel les noeuds ont été dépilés pendant le parcours ;
. il définit aussi une partition sur les arcs.
  - les arcs en arrière $u \longrightarrow v$ tq v est déjà dans la pile quand u y est empilé ;
  - les arcs du dag ;

  la classification des arcs dépend du graphe et du parcours en profondeur ;
. pour un graphe G = (N, A, r), (où r est la racine), et un parcours en profondeur $\alpha$ , on définit l'arbre de G par $G_d^\alpha = (N, A_d^\alpha, r)$ ou A est l'ensemble des arcs du dag généré par le parcours en profondeur ; $G_d$ est toujours un arbre avec racine et si un arc de $A - A_d^\alpha$ lui est ajouté, un cycle est aussitôt généré.

c) ensembles de coupure

def . un noeud u coupe un chemin P ssi il est point terminal d'un arc dans P ;
. un ensemble S de noeuds dans un graphe G est un ensemble de coupure ssi tout cycle de G est coupé par au moins un noeud de S ;
. un ensemble S est minimal si pour tout autre ensemble S' : $|S| \leq |S'|$ ;
. notons $C_G^S$ l'ensemble des cycles de G qui ne sont pas coupés par un ensemble S.

rq

S est un ensemble de coupure ssi $C_G^S = \emptyset$ .

Le problème de coupure de cycle est monotone dans le sens suivant :

Lemme

soit G un graphe et S, $S_2$ deux ensembles de noeuds tels que $C_G^{S'} \subseteq C_G^{S_2}$ ; alors le nombre minimal de noeuds qu'il faut ajouter à $S_2$ pour obtenir un ensemble de coupure est supérieur ou égal au nombre minimal de noeuds qu'il faut ajouter à $S_1$ pour obtenir un ensemble de coupure.

Le principal problème est de savoir choisir un nouveau noeud qui pourrait être ajouté à un ensemble S sans violer l'hypothèse d'induction. Le théorème suivant montre que, sous certaines conditions, ceci peut être réalisé :

th

soit S un ensemble d'un ensemble minimum de coupure donné dans un graphe G et soit $v_1 \longrightarrow v_2 \longrightarrow \cdots \longrightarrow v_n \longrightarrow v_1$ un cycle non coupé de $C_G^S$ supposons que $v_1$ à la propriété que chaque cycle de $C_G^S$ coupé par vi ($i \in [2,n]$) est aussi coupé par $v_1$ ;
alors il existe un ensemble minimal de coupure de G qui contient $S \cup \{v_1\}$

d) ensembles de coupures minimaux dans les graphes réductibles

Partie détaillée par Adi Shamir [79]

Nous ne reverrons pas ici la notion du graphe réductible (voir Hecht 77 )

def. . si $u \longrightarrow v$ est un arc en arrière dans un graphe réductible, alors u est appelé la tête et v la queue ;

. soit G un graphe réductible qui est partiellement coupé par un ensemble S de noeuds ; alors une tête u est active s'il existe un chemin dans l'arbre, chemin de u à la queue correspondante qui n'est pas coupé par les noeuds de S ;

. une tête active est maximale si aucun de ses descendants dans l'arbre n'est une tête active.

Notons que si une tête u est active, alors il existe au moins un cycle dans $C_G^S$ qui contienne u mais la réciproque est fausse.

De plus, à moins que S ne soit un ensemble de coupure, le graphe G contient au moins une tête, et, de ce fait, aussi au moins une tête active maximale. Le théorème principal justifiant l'algorithme de coupure peut s'énoncer comme suit :

th

soit G un graphe réductible, S un sous-ensemble d'un ensemble minimum de coupure ; si u est une tête active maximale de G, alors $S \cup \{u\}$ est aussi un sous-ensemble de l'ensemble minimal

de coupure.

L'algorithme de base est à présent une conséquence directe du théorème :

algorithme 1

1. commencer avec S = ∅
2. choisir une tête active maximale u dans G qui respecte l'ensemble S ; s'il n'y en a plus, arrêter ; sinon ajouter u à S et répéter le pas 2.

Pour implanter cet algorithme, il convient d'énumérer les têtes de G par un parcours en profondeur du graphe et de les considérer en post-ordre. D'après les propriétés du parcours, tous les descendants d'un noeud u dans l'arbre (ceci est en particulier vrai pour les têtes actives) apparaissent avant u dans le post ordre. Toute tête qui est active à un moment intermédiaire de l'algorithme est aussitôt ajoutée à S et, de ce fait, cesse d'être active, en respect avec le nouvel ensemble S. De plus, l'ensemble S ne peut que grossir et une tête qui n'est pas active ne peut pas le devenir à un stade ultérieur.

Par conséquent, si des têtes sont considérées en post ordre, alors toute tête qui est encore active quand vient son tour est une tête active maximale.

L'algorithme 1 peut être implanté de la façon suivante :

algorithme 2

```
mettre S=∅ , empiler la racine sur la pile (vide) ;
répéter
        tant que  il existe un arc non marqué sommet ———→ u faire
                  marquer cet arc ;
                  si u n'a pas encore été visité alors l'empiler ;
                  sinon si l'arc sommet——→u est un arc en arrière
                          alors  marquer u en tant que tête ;
                  fsi
        ftant ;
        si sommet est marqué en tant que tête et est active en res-
           pectant l'ensemble S alors l'ajouter à S ;
        fsi ;
        dépiler le sommet de la pile ;
jusqu'à pile vide ;
```

algorithme linéaire

Le moyen le plus simple de vérifier qu'une tête donnée u est active est de rechercher des chemins non coupés dans l'arbre entre u et les queues correspondantes. Ceci peut-être réalisé de façon linéaire en propageant des étiquettes à partir de u en traversant les arcs de l'arbre. La structure spéciale des graphes réductibles nous permet d'utiliser des étiquettes très économiques : à chaque noeud u est rattaché un nombre $l_s$ (u) qui sera modifié chaque fois qu'un nouveau point de coupure sera rajouté à l'ensemble courant S.
Notons n (u) (nombre compris entre 1 et |N|) la position de u dans la séquence de noeuds de b pris en préordre.

def

ls (u) = max { n (v) / il existe un arc en arrière $w \longrightarrow v$ et un chemin dans l'arbre $u \xrightarrow{*} w$ qui n'est pas coupé par S }

si aucune tête v de la sorte n'existe, alors $l_s$ (u) est défini par 0.
Ces étiquettes ont la propriété suivante :

th

soit G un graphe réductible S un ensemble arbitraire de noeuds de G et u un noeud qui ne fait pas partie des noeuds dont les descendants propres dans l'arbre sont des têtes actives, alors $l_s$ (u) $\leqslant$ n (u).

Supposons maintenant que ces étiquettes sont initialisées et modifiées (quand S change) par un processus externe de telle sorte que l'algorithme 2 puisse bénéficier de cette information supplémentaire. Le théorème suivant montre que, quand des têtes actives successives sont considérées et désactivées en postordre, le pas 6 de l'algorithme peut être implanté comme une simple vérification qui ne requiert qu'un temps constant.

th

soit G un graphe réductible et S un ensemble arbitraire de noeuds de G. Si un noeud u est tel qu'aucun de ses descendants directs dans l'arbre n'est une tête active, alors u est lui-même une tête active ssi $l_s$ (u) = n (u) .

Ce qu'il reste à faire, c'est développer une procédure efficace pour générer les étiquettes $l_s$ (u) ; pour cela, nous allons mélanger les deux opérations de calculs d'étiquettes et de sélection de point de coupure. A une étape intermédiaire dans le processus, seuls quelques noeuds de G auront une étiquette (soit L ce sous-ensemble) et S est un sous-ensemble de noeuds étiquetés. Afin de rendre ce procédé efficace, nous devons ajouter de nouveaux noeuds à L et à S, en accord avec les règles suivantes :

i) les noeuds sont ajoutés à L en postordre (i.e. un noeud est étiqueté seulement une fois que les étiquettes, de tous ses descendants sont connues)

ii) un noeud qui ne peut être ajouté à S qu'immédiatement après avoir été ajouté à L.

A présent, nous allons développer deux procédures, l'une pour ajouter un noeud à L (tout en gardant S fixe), l'autre pour ajouter un noeud à S (tout en gardant L fixe)

Ces procédures préservent la correction des étiquettes dans l'ensemble L avec respect des points de coupure dans l'ensemble S, et la correction en tout point du processus d'étiquetage se déduit alors par une induction sur $|L| + |S|$).

La procédure pour étendre L est motivée par le théorème suivant :

th

soit G un graphe réductible et S un ensemble arbitraire de noeuds, alors pour tout noeud u de G, l'équation suivante est vérifiée :

$$ls(u) = \begin{cases} 0 \text{ si } u \in S \text{ ou si } u \text{ n'a pas de descendant direct} \\ \underline{\max}\,(l_s\, v_1) \ldots l_s\,(vi),\ n\,(vi+1),\ \ldots\ n\,(v_p)) \text{ sinon} \end{cases}$$

où $u \longrightarrow v_1, \ldots, u \longrightarrow vi$ sont tous les arcs de l'arbre partant de u

et $u \longrightarrow v_1 + 1, \ldots u \longrightarrow vp$ sont tous les arcs en arrière partant de u.

Considérons le problème de modification d'étiquettes des noeuds de L quand un nouveau noeud est ajouté à S ; comme le prouve le théorème suivant, au plus une étiquette peut être affectée si les règles sont observées :

th

soit G un graphe, L un ensemble de noeuds pour lesquels des étiquettes ont été calculées en postordre , et S un sous ensemble de L.

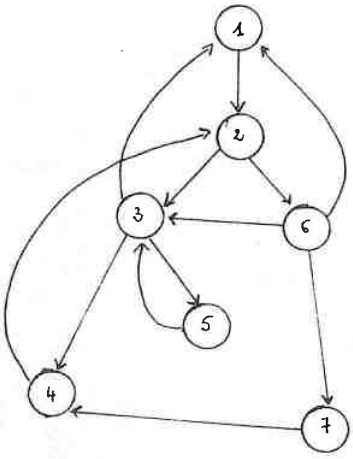
si u est le noeud suivant ajouté à L, alors ajouter u à S laisse toutes les étiquettes de S correctes avec respect du nouvel ensemble SU $\{u$ et change $l_{SU}\{u\}$

L'algorithme final utilise un parcours en profondeur pour numéroter les noeuds du graphe G en préordre, pour les étiqueter en postordre, pour considérer les têtes successives en postordre et pour ajouter de nouveaux noeuds à S (en changeant leur étiquette à O quand l'étiquette de postordre et le numéro de préordre coïncident).
Il a la même forme que l'algorithme 2 ; et son temps d'exécution et son occupation de mémoire sont linéaires en |N| + |A|.

Algorithme 3

mettre S à Ø ; mettre le compteur c à 1, empiler r sur la pile (vide) ;
répéter
mettre n (sommet) à c ; incrémenter c ; mettre l (sommet) à O ;
tant que (( il existe arc non marqué sommet ⟶ u et u déjà visité) ou (∄ arc)) faire
si (il existe arc non marqué et u déjà visité) alors
si (sommet ⟶ u) est un arc en arrière alors l
l(sommet) : = max (l (sommet), n (u)) fsi
sinon
si (l (sommet) = n (sommet)) alors ajouter sommet à S
faire l (sommet) : = O ;
fsi
conserver le sommet courant de la pile dans v ;
dépiler ;
si ( ¬ pile vide) alors l (sommet) : = max (l(sommet
l (v)) fsi
fsi
ftant
si (il existe un arc non marqué sommet ⟶ u et u non visité)
et ( ¬ pile vide) alors marquer l'arc ;
l'empiler ;
fsi
jusqu'à pile vide ;

exemple :




un ensemble minimal de coupure S est : $\{2,3\}$
il n'est pas unique puisque $\{2,5\}$ et $\{3,6\}$ sont également des ensembles minimaux de coupure

L'objectif est à présent d'étendre ces résultats aux graphes qui ne sont plus réductibles.

## 4. Ensembles de coupure dans les graphes quelconques

Cette partie résume un article de Barry Rosen [80]

### a) Introduction

Soit G = (N, A, r) un graphe orienté où tous les noeuds pourront être atteints par un chemin partant de la racine r ;
dans les applications, il ne sera pas nécessaire de découvrir un ensemble de coupure minimal mais il serait intéressant de découvrir un ensemble de coupure bien plus petit que l'ensemble des noeuds du graphe ;
l'algorithme 4 qui va être développé ici s'exécute en temps linéaire et trouve deux ensembles $S_1$ et $S_2$ tels que $S = S_1 \cup S_2$ est un ensemble de coupure vérifiant :

$$S_1 \leqslant |\text{ ensemble minimal de coupure}| \leqslant |S_1| + |S_2| = |S| \text{ ;}$$

et lorsque le graphe sera réductible, l'ensemble $S_2$ sera vide et $S = S_1$ sera l'ensemble minimal de coupure.

### b) Modification

L'algorithme 4 sera l'algorithme 3 sensiblement modifié ;
les structures de données de cet algorithme sont connues, mais il y a certaines modifications dans la terminologie ;
un arc à part d'un noeud u (origine ou queue de a) pour aller à un noeud v (extrémité ou tête de a) ; le nombre d'arcs partant d'un noeud

u est le degré ;
nous ne définirons pas une pile ni les fonctions classiques que l'on peut appliquer à ce type abstrait ; une affectation au sommet de la pile T change T sans en changer la longueur ;
l'algorithme 4 exige une plus grande dépendance vis à vis de l'ordre de parcours; nous supposons que les graphes sont représentés de telle sorte que les fonctions premier - arc et arc - suivant soient utilisables pour énumérer les arcs qui partent de chaque noeud : si des arcs partent d'un noeud u, premier - arc (u) est l'un d'eux, sinon premier - arc (u) prend la valeur spéciale nil , soit un noeud u et un arc a qui part de u , arc-suivant (u, a) est soit un autre arc partant de u, soit la valeur nil ;
si le degré de u et non nul, la suite (a, ...ad) des arcs partant de u est définie par : $a_1$ = premier - arc (u), $a_{i+1}$ = arc - suivant (u, ai) pour i $\in$ [1, degré (u) - 1].

c) algorithme 4

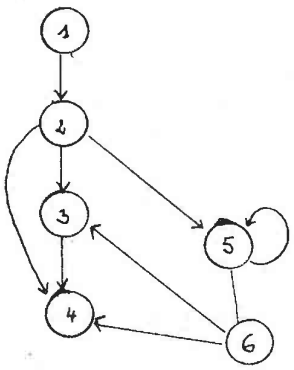
```
variables :: c : entier (compteur)
             nombre, étiq  : tableaux d'entiers indexés par les noeuds
             et initialisés à 0
             u, v : noeuds de G
             a : arc de G
             T : pile
             visite - sommet, essai - arcs : étiquettes

début
      T : = pile vide,
      empiler (r, premier - arc (r) ;
      c : = 1 ; S1 : = ∅ ; S2 : = ∅
visite-sommet :
      (u, a) : = sommet (T) ;
      nombre [u] : = C ; C : = C + 1 ;
essai - arcs :
      v : = extrémité (a) ;
      a : = arc - suivant (u, a) ;
      sommet (T) : = (u, a) ;
```

```
        si  nombre [ v ] = 0 alors  empiler (v, premier - arc (v)) ;
            aller à visite - sommet ;
        sinon si v est dans une paire empilée alors etiq [ u ]: = max
                    (etiq [u ] , nombre [ v])
        sinon       etiq [u]:= max (étiq [u], etiq [v])
        fsi
        si    etiq [u] = nombre [ u ] alors ajouter u à S1 ; étiq [u] : = 0
        sinon si étiq [u]> nombre [ u ] alors ajouter u à S2 ; étiq [u] : = 0
        fsi
        depiler T,
        si  T ≠ pile vide  alors
                           (v, a) : = sommet (T)
                           étiq [ v ] : = max (etiq [ v ] , etiq [u] ) ;
                           u : = v ;
                           aller à   essai-arcs ;
        fsi
    fin.
```

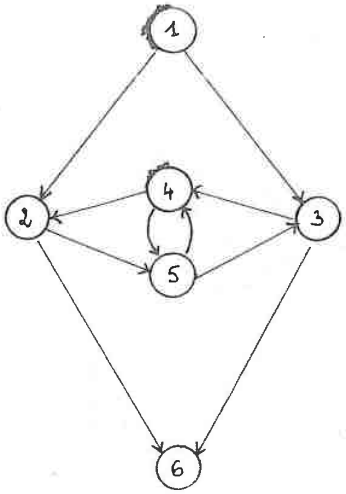
d) exemples

ex 1




$S_1 = \{ 2,5 \}$ et $S_2 = \emptyset$
il est pourtant évident que le seul point qui nous intéresse en tant que point de coupure est le noeud 5 il faudra donc améliorer ce résultat.

ex 2




deux résultats possibles :
i) $S_1 = \{ 4 \}$
ii) $S_1 = \{ 5 \}$
dans les deux cas $S_2 = \emptyset$

Nous nous contentons pour l'instant des résultats obtenus par l'algorithme 4 car nous verrons ultérieurement qu'une combinaison astucieuse avec l'algorithme que nous allons développer maintenant nous donnera de bien meilleurs résultats.

## 5. Composantes fortement connexes

Cette étude a été traitée par Robert Tarjan [ 72 ]

### a) Introduction

A présent, nous allons tenter de trouver une méthode de parcours d'un graphe ; pour ce faire nous voulons décomposer ce graphe en sous-graphes qui auront des propriétés très particulières.

### b) connection forte

def soit G un graphe orienté ; supposons que pour tout couple (u, v) de noeuds de $G_1$ il existe deux chemins $p_1 : u \xrightarrow{*} v$ et

$$p_2 : v \xrightarrow{*} u$$

alors G est dit fortement connexe.

prop

on définit une relation d'équivalence sur l'ensemble des noeuds

$u \sim v$ ssi $\exists$ p chemin fermé : $u \xrightarrow{*} u$ qui contienne v ;

soit Ni les classes d'équivalence définies par cette relation et $G_i$ les sous graphes (Ni, Ai) où $Ai = \{ (u, v) \in A / u, v \in Ni \}$

alors

i) tout Gi est fortement connexe

ii) aucun Gi n'est un sous graphe propre d'un sous graphe fortement connexe

def les sous graphes Gi ainsi définis sont appelés composantes fortement connexes de G.

considérons les arcs du graphe et voyons l'incidence d'un parcours en profondeur appliqué à celui-ci

. l'ensemble des arcs qui conduisent à un nouveau noeud forme un un arbre

Les autres arcs se subdivisent en trois catégories ;

. certains arcs vont d'un ancêtre à ses descendants dans l'arbre : ils seront ignorés car ils n'affectent en rien les composantes fortement connexes ;

. d'autres arcs vont des descendants vers leurs ancêtres dans l'arbre : ils seront appelés arcs frontaux et seront notés $u \longmapsto v$ ;

. d'autres enfin vont d'un sous arbre à une autre dans l'arbre : ils seront appelés arcs transversaux et seront notés également $u \longmapsto v$ ;

Le problème de trouver les composantes fortement connexes d'un graphe se ramène aux problème de trouver les racines des composantes. Nous construisons un test simple qui permet de déterminer si un noeud est racine d'une composante fortement connexe :

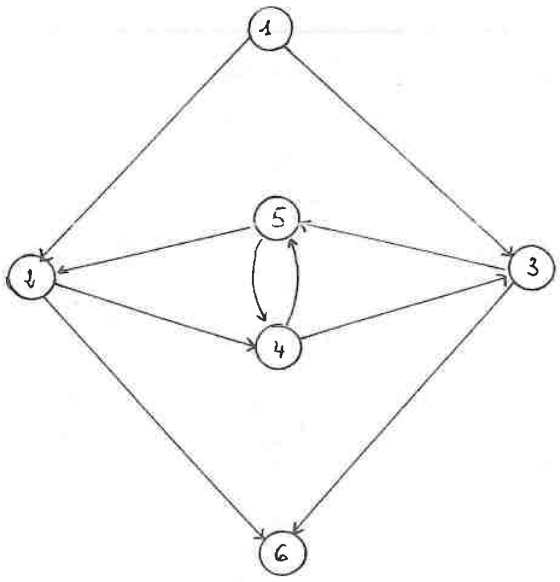
$$\text{lowlink}(u) = \underline{\min}\ (\{u\} \cup \{ v / u \xrightarrow{*} \longmapsto v$$
$$\underline{\text{et}}\ \exists w\ (w \xrightarrow{*} u\ \underline{\text{et}}\ w \xrightarrow{*} v\ \underline{\text{et}}\ w \text{ et } v \text{ dans la même composante}\})\text{;}$$

c'est-à-dire que lowlink (u) est le plus petit noeud qui est dans la même composante que u et que l'on peut atteindre en traversant zéro ou plus d'arcs dans l'arbre suivi plus un arc frontal ou un arc transversal.

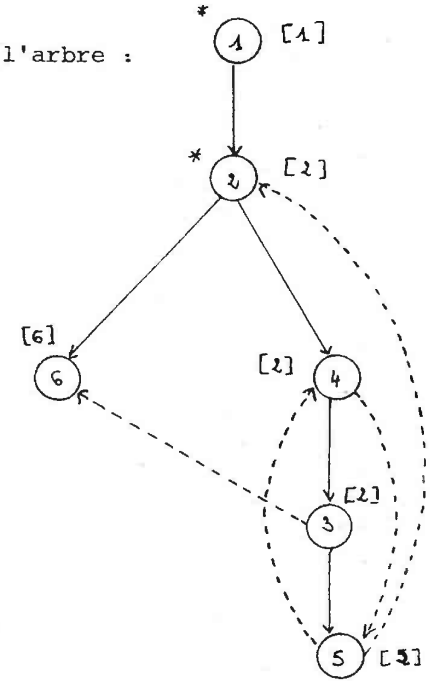
prop v est racine d'une composante fortement connexe ssi lowlink (v) = v

ex

pour le graphe suivant



on a l'arbre :




les valeurs de lowlink sont entre crochets
Ces racines sont marquées par *

c) algorithme

```
procédure connection (u).
début
     C : = C + 1 ;
     lowlink (u) : = C ;
     nombre [ u] : = C ;
     empiler u ;
     pour v successeur de u faire
          si v n'a pas été numéroté alors
                    connection (v)
                    lowlink (u) : = min (lowlink (u), lowlink (v))
        sinon si nombre[ v] < nombre [u] alors
                 si v  est dans la pile alors lowlink (u): =
                    min (lowlink (u), nombre [ v])

          fsi
     fpour
```

```
        si  lowlink (u) = nombre [ u ] alors
                commence une nouvelle composante fortement connexe
                tant que v au sommet de pile vérifie nombre [ v ] ≥ nombre [ u
                        enlever v de la pile et l'ajouter à la composant
                ftant
        fsi
    fin ;
    programme principale
    début
        C : = 0 ;
        pile : = ∅
        pour w non encore numéroté faire connection (w) ;
    fin.
```

th l'algorithme qui découvre les composantes fortement connexes s'exécute correctement et requiert un temps en O (|N|, |A|).

## 6. Décomposition d'un graphe en composantes fortement connexes

Ce paragraphe résume l'étude de Michael Karr [74]

### a) Introduction

Nous allons dans ce paragraphe poursuivre l'étude des composantes fortement connexes ; notre problème sera de décomposer un graphe en composantes fortement connexes, puis de décomposer chaque sous-graphe en composantes fortement connexes.

### b) Problème

Remarquons que si un arc fait partie d'un sous graphe, les noeuds qui le forment font également partie de ce sous graphe, mais la réciproque est fausse.

def Si H est un sous graphe de G, on dit que H est une composante fortement connexe de G si

i) H est fortement connexe

ii) il n'existe aucun sous-graphe de ce contenant H et strictement plus grand que H, qui soit lui-même fortement connexe.

En d'autres termes, H est un sous graphe fortement connexe maximal de G ; pour abréger, on dit que H est une composante de G.
Il n'est pas très difficile de prouver qu'il existe une unique décomposition de tout graphe orienté en composantes fortement connexes.
La question qui se pose maintenant est de savoir comment choisir entre les sous graphes fortement connexes contenus dans H.

def soit H un sous-graphe de G ; pour tout noeud n de G, on dit que n est noeud d'entrée de H si les arcs dont n est la tête ne sont pas dans $A_H$.

A présent nous pouvons définir les sous graphes fortement connexes de H.

def soit H un sous graphe de G ; pour tout sous-graphe K de G, on dit que K est une composante fortement connexe relativement à H si

i) K est sous graphe de H,

et

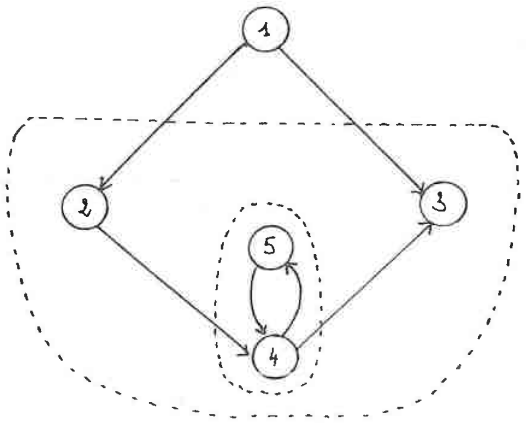
ii) K consiste en un noeud isolé et aucun arc ;

ou

deux noeuds distincts de K sont sur un cycle orienté contenu dans K et qui ne contient aucun noeud d'entrée de H.

Pour abréger, nous dirons que K est une composante de H.

ex



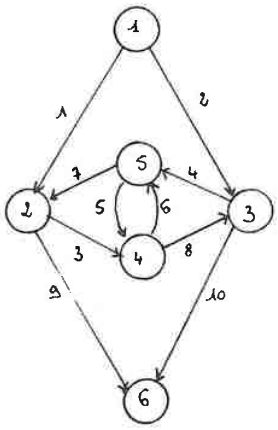

rq Pour spécifier un sous graphe fortement connexe, il suffit de spécifier ses arcs ; pour réaliser ceci, on associe une fonction $o : A \longrightarrow N^+$ injective telle que si toute composante est objectivement définie par un couple (p, q) on ai :

$$a \in H \iff p_H \leq o(a) \leq q_H$$

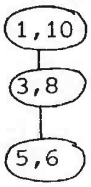
avec cette convention nous pouvons représenter une décomposition en composantes fortement connexes par un arbre (les noeuds de l'arbre contiennent un couple d'entiers représentant un sous-graphe non trivial)

ex

le graphe




se décompose en composantes fortement connexes et cette décomposition sera représentée par l'arbre




Pour plus de commodité, nous préfèrerons écrire sous la forme

1 2 [3 4 [5 6] 7 8] 9 10

c) Algorithme

L'algorithme présenté est une adaptation de l'algorithme connection vu dans la paragraphe précédent ; nous allons faire un parcours en profondeur dans le graphe, mais en prenant tous les arcs dans le sens contraire (à cause de la numérotation des arcs qui doit se faire à partir du début du graphe.

Afin de faciliter la suite de l'étude, nous faisons les hypothèses suivantes :

- tout noeud de G peut être atteint par un chemin en arrière à partir d'un noeud de sortie ;
- soit X l'ensemble des noeuds de sortie.

Pour l'implantation de l'algorithme de recherche des composantes fortement connexes, nous allons utiliser les données suivantes :

. une pile T
. O : fonction de numérotation des arcs
. A : arbre de décomposition
. c : compteur des noeuds

. i : compteur des arcs

. entrée : tableau de booléens indicé par les noeuds
indique qu'un noeud est noeud d'entrée potentiel
d'une boucle

. bloqué : tableau de booléens indicé par les noeuds
précise si une tentative de chemin en arrière
à travers les noeuds en question est possible ou
non

```
programme principal ;
début
     C : = i : = 0 ;
     T : = pile vide ;
     X : = { noeuds de sortie du graphe }
     pour tout arc a de A faire  o (a) = 0  fpour
     pour tout noeud n de N faire
           bloque [ n]: = entrée [n] : = faux ;
           lowlink [ n]: = nombre [n] : = 0;
     fpour
     t : = décomposition(X) ;
fin

procédure  décomposition (Z ) ;
     p : entier; y : ensemble ;
début
     t : = arbre vide ;
     p : = i + 1,
     pour tout noeud n de Z faire bloque [n] : = vrai  f pour
     pour tout noeud n de Z faire connection (n) f pour
     si    p ≤ i   alors  t : = [ p, i] ∧ t, fsi
fin

procédure numérotation 1 (n) ; (* numérote tous les arcs qui entrent
          par un noeud qui n'est noeud d'entrée d'une composante *)
début
     pour tout arc a dont la tête est n faire
          i : = i + 1 ;
          o (a) : = i ;
     fpour
          bloque [ n ] : = vrai;
fin
```

```
procédure  numérotation 2 (n) ; (* numérote les arcs d'entrée d'une
           composante, arcs qui n'ont pas encore été numérotés *)

début
      pour  tout avec  a  dont la tête est n  et tq o (a) ≠ o faire
            i : = i + 1 ;
            o (a) : = i ;
     fpour
fin

procédure  numérotation 3 (n), (* numérote le reste des arcs d'en-
           trée d'une composante au moment où celle-ci va être
           étudiée *)
début
      pour tout arc a dont la tête est n et tq o (a) ≠ 0 et issu
           d'un noeud bloqué faire
           i : = i + 1 ;
           o (a) : = i ;
     fpour
fin

procédure  connection (n)
début
     C : = C + 1 ; nombre  n  : =  C ; lowlink  n  : = c ;
           empiler n ;
     pour tout noeud p prédécesseur de n faire
          si bloque [ p] = vrai alors entrée [n] : =  vrai
        sinon si nombre [p] = 0 alors
                 connection (p) ;
                 si bloque [p] = vrai alors entrée [n] : = vrai
               sinon                          lowlink [n] : = min
                                  (lowlink [n] ,  lowlink [p] )
                 fsi
        sinon si  nombre  [p] ≤ nombre [n] et p est dans T alors
                 lowlink [n] : = min (lowlink [n] , nombre [p] )
        fsi
     fpour
```

```
si lowlink [ n ] =  nombre  [ n ] alors
      si   nombre [ n ]  =  c  alors
             si   bloque [ n ] = faux alors numérotation 1 (n)
           sinon                          numérotation 2 (n)
            fsi
            dépiler n ;
     sinon
           y = ∅
           repeter
                  m : = sommet (T)                                  B
                  dépiler m ;
                  si  entrée [ m ]  : = vrai  alors
                        numérotation 3  (m)
                        y : = y U { m }
                  fsi
                  lowlink [m] : = nombre [m] : = oj
           jusque  m = n ;
     fsi
fsi

fin
```

7. Remarques

Cet algorithme, s'il trouve les composantes fortement connexes de façon correcte, perd beaucoup de temps à l'exécution en considérant les noeuds isolés et en testant par un appel de la procédure décomposition si ce noeud est un cycle : la réponse est évidemment négative et cet appel a été de trop. Nous devons donc améliorer les résultats sur ce plan.

De même, la numérotation des arcs est superflue puisque les arcs sont déjà numérotés (au moment de la construction du graphe) ; bien sûr, les numéros ne sont pas les mêmes mais peut être que si l'arbre de décomposition prenait en compte les numéros effectifs plutôt que les valeurs données par la fonction O, pourrait-on espérer un gain de place mémoire.

Les deux algorithmes (recherche des points de coupure et recherche des composantes fortement connexes d'un graphe) sont basés sur la même tech-

nique de parcours à savoir parcours avec priorité à la profondeur, si ces deux informations pouvaient être découvertes simultanément, cela économiserait quelques parcours du graphe. Ceci fait l'objet des paragraphes qui suivent.

## 8. Graphe - inverse du graphe

La première différence qui nous apparaît clairement est que l'algorithme de recherche des points de coupure exécute un parcours en profondeur du graphe étudié ( cad en traversant les arcs dans leur sens d'orientation) tandis que l'algorithme de recherche des composantes fortement connexes exécute un parcours en profondeur du graphe déduit du graphe initial en inversant le sens de tous les arcs (appelons le graphe inverse) et notons $G^{-1}$ le graphe inverse de G.

Nous allons ici faire des restrictions sur les graphes que nous étudierons (l'analyseur veillera à ce que ces conditions puissent être vérifiées) ; ces restrictions ne seront pas trop contraignantes si l'on se souvient que les graphes qui seront étudiés ne sont pas quelconques et sont en fait des graphes de dépendance de programme Pascal.

Les restrictions sont les suivantes :

- un seul point d'entrée et un seul point de sortie
  (tout programme comme par l'instruction begin et se termine par l'instruction end)
- tout noeud du graphe peut être atteint par un chemin en arrière partant du noeud de sortie
- tout noeud du graphe peut être atteint par un chemin en avant partant du noeud d'entrée.

lemme

Il existe une bijection entre un graphe G et le graphe inverse $G^{-1}$ qui a tout arc associe l'arc dont le sens d'orientation a été inversé.

On en déduit immédiatement :

lemme

L'image du noeud d'entrée de G par cette bijection est le noeud de sortie de $G^{-1}$, inversement, l'image du noeud de sortie de G est le

d'entrée de $G^{-1}$.

Nous nous souvenons en outre que l'analyse sémantique utilise deux techniques qui sont l'analyse en avant et l'analyse en arrière. L'analyse en avant a besoin d'informations sur le graphe G tandis que l'analyse en arrière a besoin des mêmes informations sur le graphe inverse $G^{1}$.
Il nous est naturellement venu à l'idée de regrouper dans un même algorithme la recherche des points de coupure d'un graphe G et la recherche des composantes fortement connexes du graphe inverse $G^{-1}$ ; l'application de cet algorithme au graphe inverse nous donnera les points de coupure de $G^{1}$ et les composantes fortement connexes de $(G^{-1})^{-1} = G$. Ainsi l'information sera complète pour le graphe et son inverse. Si nous arrivons à construire cet algorithme , nous effectuerons un parcours avec priorité à la profondeur d'un graphe, ce qui nous permettra de connaître un ensemble (si possible minimal) de points de coupure ; ce parcours sera exactement celui que nous ferons dans le graphe inverse où tous les arcs auront été inversés, ce qui nous permettra de connaître les composantes fortement connexes de ce graphe inverse.

## 9. Algorithme récursif

### a) Introduction

Dans ce paragraphe, nous allons regrouper en un algorithme récursif l'algorithme de recherche des points de coupure et l'algorithme de recherche des composantes fortement connexes. Une remarque : le premier a été écrit de façon itérative mais il est clair que sa structure est celle d'un algorithme récursif qui utilise une pile.

```
début  /* P*/
  C : = C + 1 ; nombre [ n ] : = C ; empiler (n)
  pour tout suivant  s  de  n  faire
       si nombre [ s ] = 0  alors
                P(s) ;
                étiq [ n ] : = max (étiq [ n ] ,  etiq [ d ] )
       sinon si  s  est dans la pile alors etiq [ n ] : = max (etiq [n] ,
                                                            nombre [ s ] )
       sinon                                  etiq [ n ] : = max (etiq [ n ] ,
                                                            etiq [s] )
       fsi
fpour
```

```
        si  etiq[ n] ≥ nombre [ n ]  alors
                S : =S U { n }
                etiq [n] : = 0
        fsi
        dépiler (n)
fin
```

A

b) Méthode

L'instruction "pour tout suivant s de n faire" va devenir du fait de la bijection $G \longrightarrow G^{-1}$ : "pour tout prédécesseur $p$ de n faire "

Entre l'algorithme de connection (VI.6.) et l'algorithme P ci-dessus les similitudes provenant du parcours en profondeur sont évidentes. Les différentes proviennent des partes A et B où la recherche s'effectue d'une part si les points de coupure, d'autre part sur les têtes de cycles de composantes fortement connexes.

On peut noter ainsi que les piles utilisées ne sont pas identiques car dans l'algorithme P ci-dessus, un noeud est dépilé dès qu'on a vérifié s'il est point de coupure tandis que dans l'algorithme connection, on ne dépile des noeuds que si l'on a trouvé une composante entièrement.

De plus si une composante fortement connexe est découverte, cette composante est alors aussitôt étudiée en tant que sous graphe contenant peut-être lui aussi des composantes fortement connexes ; mais dans ce sous graphe, les points de coupure ont déjà été découverts et n'ont pas à être recalculés ou modifiés ; pour pallier à ce problème, nous introduisons une variable auxiliaire qui nous indiquera à tout moment si nous sommes ou non dans le premier appel de la procédure décomposition (si ce n'est pas le premier appel, il ne faut rien changer aux points de coupure).

Comme les piles ne sont pas utilisées de la même façon dans les deux algorithmes, la solution consiste à garder deux piles différentes. Nous allons pouvoir écrire un algorithme récursif très simplement en modifiant la procédure connection mais sans changer le programme principal ou les différentes procédures de numérotation des arcs.

Les différents cas qui apparaissent dans l'algorithme connection sont :

- bloque [ p ] = vrai
- nombre [ p ] = 0
- nombre [ p ] $\leq$ nombre [ n ] et p est dans la pile

Etudions les cas :

. Le cas "nombre [ p ] = 0 " est le même que pour l'algorithme P puisque la numérotation par un parcours en profondeur est la même.

. Un noeud P est bloqué si tous les arcs qui lui sont incidents sont déjà numérotés (on bloque un noeud afin d'éviter tout retour en arrière à travers lui) ; dans ces conditions, les étiquettes rangées dans le tableau étiq et qui vont définir les points de coupure sont déjà déterminées et ne sont pas modifiées dans ce cas.

. Remarquons que la pile de l'algorithme connection contient toujours la pile de l'algorithme P et l'on peut même écrire que :

nombre [ p ] $\leqslant$ nombre [ n ] si p est dans la pile de l'algorithme P

ceci est immédiat si l'on considère qu'un noeud est empilé en même temps qu'il est numéroté.

Cette étude par cas nous amène à considérer les possibilités suivantes :

- bloque [ p ] = vrai
- nombre [ p ] = 0
- nombre [ p ] $\leqslant$ nombre [ n ] et p est dans la pile (de l'algorithme connection)
- p est dans la pile

c) algorithme

Les parties appelées A et B ne sont pas modifiées ; elles garderont chacune leur pile S et T (resp) ; la variable niveau = vrai s'il s'agit de l'appel principal de la procédure décomposition. La procédure connection modifiée s'écrira :

```
procédure connection (n) ;
début
    C := C + 1, nombre [n] := lowlink [n] := C ; empiler S (n);
         empiler T (n);
    pour tout prédécesseur p de n faire
          si bloque [p] = vrai alors entrée [n] := vrai
         sinon si nombre [p] = 0 alors
                    connection (p) ;
                    si bloque [p] = vrai alors entrée [n] := vrai
                sinon       lowlink [n] := min (lowlink [n],
                                                lowlink[p] )
```

```
                    si niveau = vrai alors étiq [ n ] : = max (etiq
                                [ n ] , étiq [ p ] ) fsi
                fsi
        sinon si   nombre [ p] ⩽ nombre [n] et  p est dans T alors
                lowlink [ n ] : = min  (lowlink [ n ] , nombre [ p ] )
                si  niveau = vrai alors étiq [ n ] : = max  (etiq [ n] ,
                                nombre  [ p ] )  fsi
        sinon si  p  est dans T alors
                si niveau = vrai alors etiq [ n ] : = max  (etiq [ n ] ,
                                etiq [p] )  fsi
        fsi
    fpour
    A ;
    B ;
fin.
```

10. Algorithme itératif

a) introduction

Avant d'implanter l'algorithme précédent, nous avons voulu le transformer en un algorithme non récursif : ceci nous permettra un gain de place du fait de l'empilement souvent excessif de données par le compilateur. Mais la transformation d'un programme récursif en un programme non récursif voit apparaître une pile, ce qui porte le nombre de piles à 3 et qui ne facilitè guère la compréhension et la lisibilité du programme. Nous introduisons un nouvel ensemble.

b) ensemble

On remarquera que la pile de l'algorithme de recherche des points de coupure est toujours incluse dans la pile de l'algorithme de recherche des composantes. La troisième pile qui apparaît de par la transformation en itératif de la procédure récursive connection, va contenir pour un noeud donné, la liste de ses prédécesseurs. A ce point précis, nous pouvons faire une amélioration en considérant la possibilité de rencontrer le cas de figure suivant :

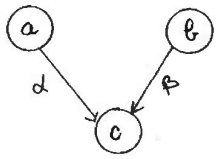
(si n n'a qu'un seul prédécesseur p, il y a pourtant deux arcs partant de p et arrivant en n)

Nous choisissons de remplacer cet empilement de noeuds prédécesseurs d'un noeud donné par l'empilement des arcs incidents à ce noeud. Et afin de conserver de quel noeud les arcs sont les arcs incidents, nous allons considérer des éléments du type (noeud, arc).

L'empilement d'un noeud pourra se faire en donnant à "arc" une valeur spéciale.

Donc il est simple d'empiler soit des arcs, soit des noeuds sous la forme d'éléments définis ci-dessus.

ex considérons le sous graphe suivant :




lorsque le contrôle sera au noeud C, l'algorithme va empiler le noeud C ainsi que tous ses arcs incidents, ce qui nous fournira l'état de l'ensemble :

(c, nil ) où nil est la marque spéciale
(c, α )
(c, β)

Comme tous les noeuds vont être empilés, les piles des algorithmes de base (à savoir l'algorithme P et l'algorithme connection) seront incluses dans cet ensemble ; et afin de pouvoir retrouver tout élément de l'une ou de l'autre, nous introduisons deux pointeurs sur l'ensemble : pt-boucle désignera le sommet de la pile de l'algorithme de recherche de points de coupure, pt-sommet désignera le sommet de la pile de l'algorithme de recherche des composantes fortement connexes. Du fait de l'inclusion de ces piles, on aura toujours pt-sommet $\geqslant$ pt-boucle. Les noms ne sont pas choisis au hasard car pt-sommet désignera (après étude des arcs) toujours l'élément au sommet du nouvel ensemble tandis que pt-boucle sera décrémenté jusqu'à la rencontre d'un noeud qui est une sortie de boucle.

Le dépilement ne pourra pas se faire de façon naturelle comme pour une pile donc seuls les éléments qui seront au delà de pt-sommet pourront être réutilisés pour les autres éléments; nous allons leur ajouter un indicateur précisant si un élément donné a été libéré, c'est-à-dire si sa présence dans l'ensemble est devenue inutile ; dans ce cas, aucun des deux pointeurs ne se soucieront de lui et la place qu'il occupe sera réuilisée dès que le pointeur pt-sommet sera revenu en deça de la position de cet élément.

La transformation en algorithme itératif de la procédure connection peut se faire de la sorte ; mais la procédure décomposition (VI. 6 C) est également récursive car appelée par connection ; ceci nécessite un apport d'information dans cet ensemble.

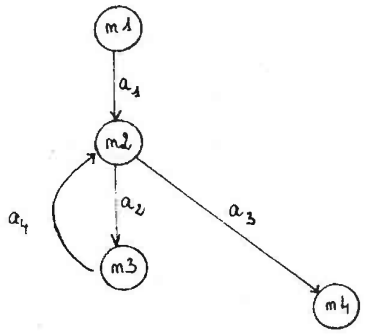
La première modification consiste à conserver tous les noeuds d'entrée à une boucle ; - ces noeuds étaient rangés dans la variable locale y de décomposition - nous "empilerons" dans notre ensemble ces noeuds munis d'une nouvelle marque spéciale * et chacun de ces éléments sera dépilé au moment de son étude.

Le second problème sera de conserver l'entrée des boucles étudiées, ceci étant fait automatiquement par le compilateur dans le cas de la procédure récursive ; comme nous en avons maintenant l'habitude, nous introduisons un élément nouveau : (- niveau, premier-arc). La variable niveau comptera le niveau d'imbrication des boucles et sera changée de signe afin qu'aucune confusion ne soit possible avec les noeuds du graphe ; la variable premier-arc désignera le numéro susceptible d'être donné au premier arc numéroté dans le sous graphe.

Cette petite astuce nous permet de savoir rapidement si nous étudions une composante triviale (ie aucun arc n'a pu être numéroté) ou non ; car dans le cas d'une composante triviale, aucun sous arbre ne doit être créé dans l'arbre de décomposition.

c) exemple

Nous donnons ici un exemple sans nous soucier de la façon dont les arcs sont numérotés, ni comment les valeurs de lowlink et etiq permettent de découvrir les composantes fortement connexes et les points de coupure




Sur le schéma, nous allons barrer les éléments qui seront libérés

étape 1 ∅

étape 2 (n4, nil)
(n4, a3)

étape 3 /* entrée de la boucle */
(n4, nil) ~~(m4, a3)~~
(n2, *)

étape 4 (n4, nil)
(n2, 0)
(n2, a1) /* premier noeud de la boucle */
(n2, a4)

étape 5 (n4, nil)
(n2, 0)
(n2, a1) ~~(m2, a4)~~
(n3,0) /* deuxième noeud de la boucle */
(n3 a2)

étape 6 (n4, nil)
(n2, 0)
(n2, a1) ~~(m2, a4)~~
(n3, 0) /* étude du deuxième noeud * /

étape 7 (n4, nil) ~~(m3, a2)~~
(n2, 0)
(n2, a1)
(n3, 0) ~~(m2, a4)~~
(n1, 0) /* étude de l'arc de sortie */
(n1, a1)

etc ...

d) algorithme final

Nous sommes maintenant prêts à faire la transformation de l'algorithme récursif décrit au paragraphe 9 en un algorithme itératif. Mais dans un premier temps, cette transformation fait apparaître des instructions de saut (conditionnel ou inconditionnel) ; ces instructions peuvent être remplacées aisément à condition de prendre quelques précautions. Nous introduisons pour cela un ensemble de situations qui vont décrire les différents cas qui apparaissent dans le programme. Trois groupes de situations sont à envisager :

le premier groupe considère les différentes possibilités concernant les noeuds prédécesseurs d'un noeud donné

le second groupe examine le noeud en question (s'il est tête de boucle)

le troisième groupe correspond au classique dépilement de la pile qui est créée lors de la transformation récursif → itératif

Voyons en détail ces trois groupes de situations :

| * situation 1 | conditions la définissant |
|---|---|
| bloquée | bloque [p] = vrai |
| non-visité | nombre [p] = 0 |
| inférieur | nombre [p] nombre n et p dans l'ensemble |
| interne | P est dans l'ensemble |

| * situation 2 | conditions la définissant |
|---|---|
| autoboucle | lowlink [n] = nombre [n] = C et ¬ block |
| cul-de-sac | lowlink [n] = nombre [n] = C et block |
| boucle-trouvée | lowlink [n] = nombre [n] ≠ C |
| shunt | lowlink [n] ≠ nombre [n] |

rq le cas"cul-de-sac" a été ajouté afin de faciliter l'exécution : en effet si tous les noeuds prédécesseurs d'un noeud n sont bloqués, alors le noeud n n'a pas d'autre solution qu'être un noeud normal c'est-à-dire qu'il est bien tête de boucle mais d'une boucle ne contenant aucun arc. Ce cas évite au programme de chercher les boucles triviales.

rq Le cas auto boucle a été différencié afin de considérer le cas où un arc part du noeud n et revient au même noeud sans passer par un autre noeud, ce cas n'était pas repéré par l'algorithme récursif d'où une amélioration sensible.

La fonction block à résultat booléen indique si tous les noeuds prédécesseurs d'un noeud sont bloqués.

Lors d'un dépilement d'un élément, le troisième groupe de situation examine le premier élément non libéré au sommet :

| * situation 3 | conditions la définissant |
|---|---|
| sortie | noeud = 0 |
| fin boucle | noeud < 0 (cad niveau < 0) |
| nouvelle entrée | arc = * |
| test | arc = nil |
| normale | autres |

La procédure principale peut s'écrire :

```
procédure   ordgraphe ;
début
      initial ;
      répéter
            initialiser situations ;
            tant que  ( arc≠nil ) et ( situation 1 ≠ non visité )
                        faire etudenoeud  ftant
            si  (situation 1 ≠ non-visité) alors
                étude arc ;
                si  ((situation 2 ≠ boucle trouvée) ou (¬ (indboucle)))
                                                              alors
                    répéter
                          continuer - étude ;
                    jusqu'à (situation 3 = sortie ou nouvelle entrée
                             ou normale) ;
                fsi
            fsi
      jusqu'à (situation 3 = sortie) ;
fin ;
```

Le détail de cette procédure risque de paraître fastidieux pour le lecteur et pour cette raison, nous ne donnerons que les points importants :

. "initial" initialise les différents tableaux ainsi que l'ensemble

que nous avons défini

- "étude-noeud" calcule tout d'abord la situation 1 et calcule les différentes valeurs de lowlink, nombre et étiq comme nous l'avons vu dans les algorithmes précédents.
- "étude-arc" calcule la situation 2 et teste si un noeud est point de coupure ; puis en fonction de valeurs de la situation 2, elle numérote les arcs grâce aux procédures de numérotation déjà détaillées ou bien commence l'étude d'une nouvelle boucle. La variable indboucle indique s'il s'agit d'une véritable boucle (ie non triviale)
- "continuer-étude" marque les fins de boucle et indique si une autre boucle commence ou si un autre chemin partant du noeud considéré est à étudier.

e) Conclusion

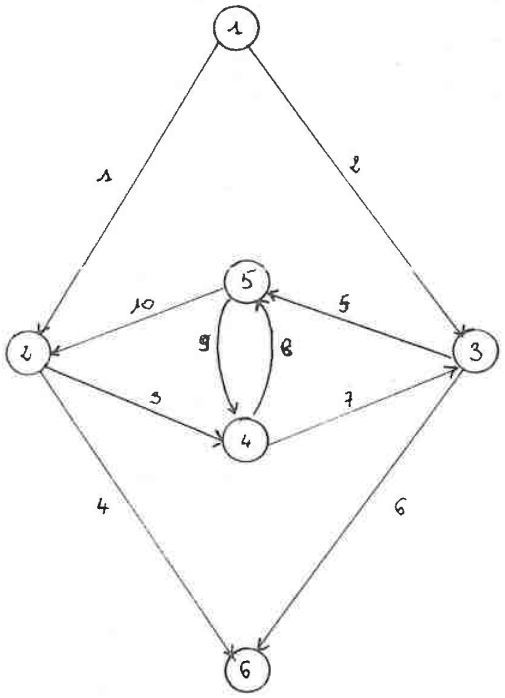
D'autres modifications permettent d'améliorer la lisibilité des résultats, comme par ex. ne plus numéroter les arcs du graphe mais écrire dans l'expression régulière du sens de parcours les numéros des arcs tels que la construction du graphe les a définis.

Cet algorithme a été implanté dans le cadre général de l'analyseur sémantique. Pour donner un ordre d'idée, il compte quelques 500 instructions en langage Pascal et son temps d'exécution varie entre 3 et 5 secondes pour des programmes comprenant entre 120 et 150 lignes.

Dans l'annexe qui suit, nous donnons trois exemples illustrant les résultats obtenus.

f) exemples

* ex 1



pour le graphe :

noeuds de coupure : 4

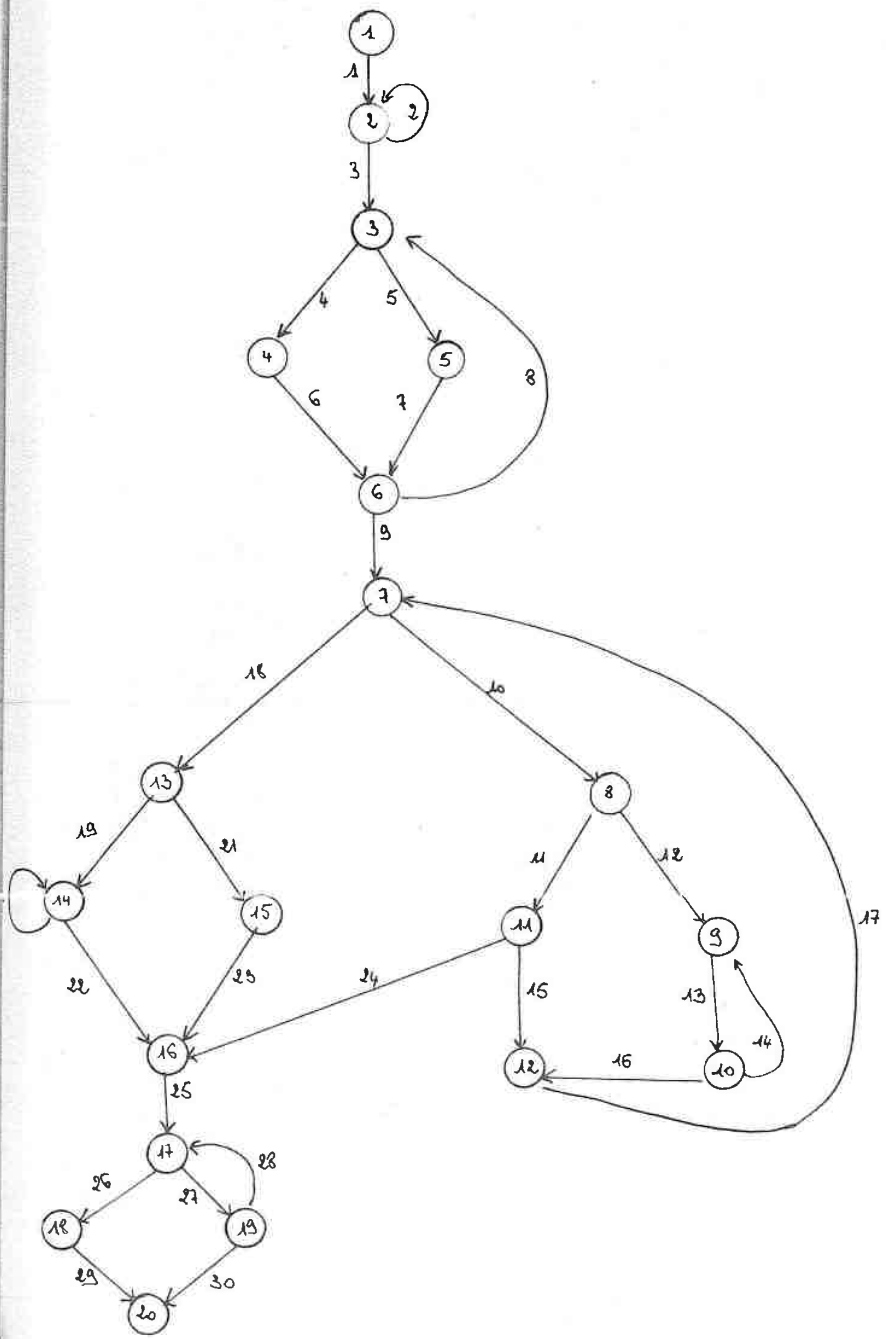
expression régulière : 1 2 [ 3 5 [ 8 9 ] 7 10 ] 4 6

pour le graphe inverse :

noeuds de coupure : 5

expression régulière : 4 6 [ 10 7 [ 8 9 ] 5 3 ] 1 2

* ex 2



pour le graphe :

noeuds de coupure : 2, 3, 7, 9, 14, 17

expression régulière :

1 [2] 3 [4 5 6 7 8] 9 [10 11 12 [13 14] 15 16 17]

18 19 [20] 21 22 23 24 25 [27 28] 26 29 30

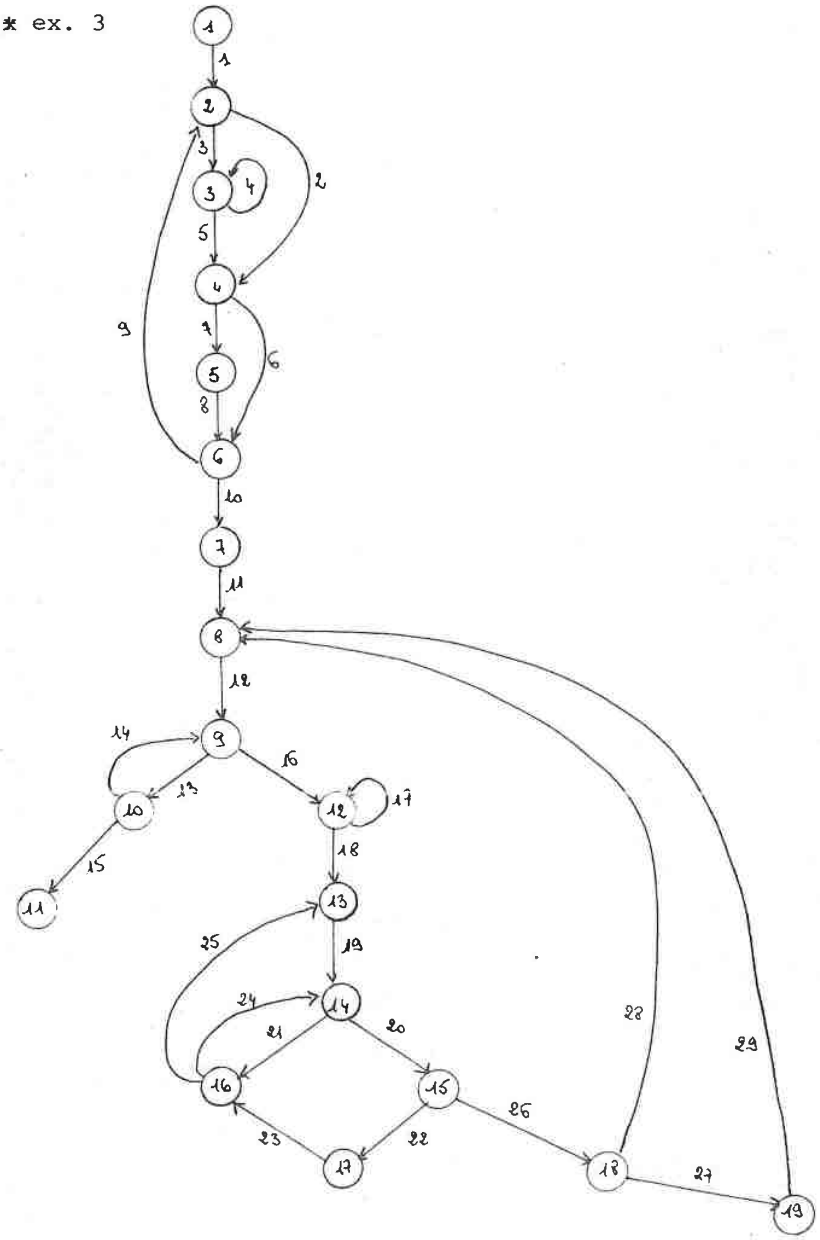
pour le graphe inverse :

noeuds de coupure : 2, 6, 11, 10, 14, 17

expression régulière :

29 30 26 [28 27] 25 23 22 [20] 19 21 18 24 [17 16 [13 14] 12 15 11 10] 9 [6 7 5 4 8] 3 [2] 1

* ex. 3



pour le graphe :

noeuds de coupure : 2, 3, 9, 12, 14

expression régulière :

1 [3 [4] 2 5 6 7 8 9] 10 11 [12 [13 14]16 [17] 18
[19 [20 21 22 23 24] 25] 26 27 28 29] 15

pour le graphe inverse :

noeuds de coupure : 3, 6, 10, 12, 14

expression régulière

15 [13 [12 29 28 27 26 [20 [19 24 25 21] 23 22]
18 [17] 16] 14] 11 10 [8 6 7 5 [4] 2 3 9] 1

# VII - Pointeurs

## 1. Introduction

Ce chapitre sera consacré à l'étude des variables de type pointeur sur le tas, mais les résultats que nous pourrons obtenir ne sont pas implantés dans l'analyseur sémantique à ce jour. Dans un langage comme Pascal, l'utilisation d'une variable de type pointeur pour accéder à un bloc de mémoire pose le problème de vérifier que le pointeur utilisé n'est pas nil ; les compilateurs reportent cette vérification au moment de l'exécution des programmes, mais cette solution à l'inconvénient de découvrir très tardivement les fautes de programmation.

Dans un premier temps, nous considèrerons un pointeur comme une variable entière puis nous étendrons le cas où le pointeur n'est pas nul pour différencier les différents éléments qu'il peut désigner.

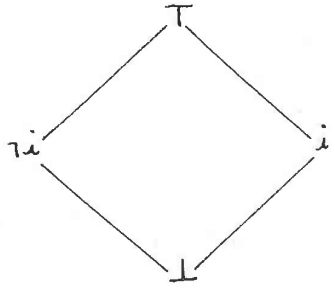
Pour résoudre le difficile problème d'interaction mutuelle des pointeurs dans un programme, nous reprendrons un analyse des partitions des variables pointeurs telles que deux variables dans des groupes distincts ne puissent pas repérer indirectement le même objet.

Et nous remarquerons que ces deux méthodes peuvent être faites simultanément et apportent alors des résultats plus fins.

## 2. Variables de type pointeur

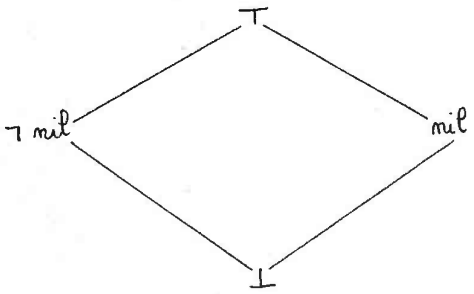
### a) Initialisation

Comme pour une variable entière, une variable de type pointeur doit être initialisée par une affectation et toute utilisation d'un pointeur non initialisé entrainera une erreur à l'exécution. Le treillis des valeurs est donc le même que les variables entières du programme :

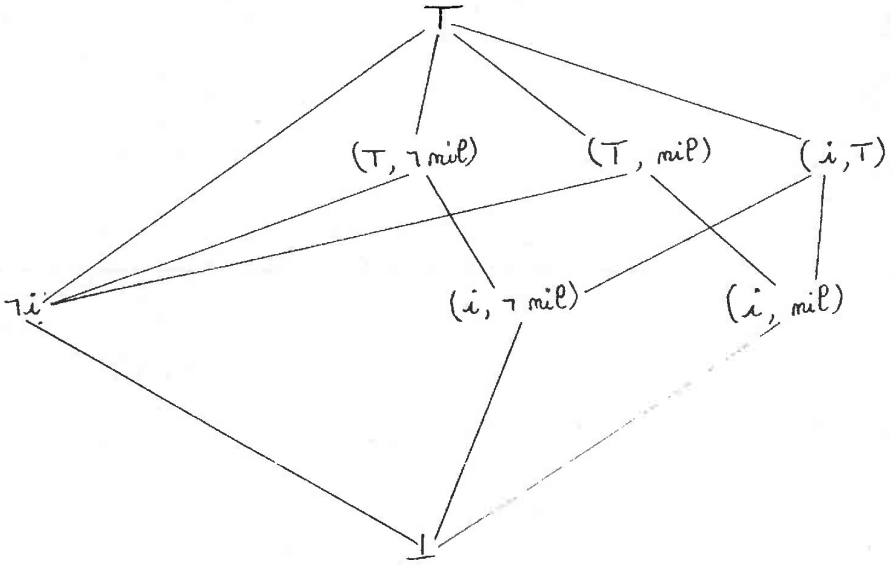


b) Valeurs

Si une variable de type pointeur a été initialisée, une valeur lui a donc été affectée : cette valeur est soit le numéro (dans la mémoire) d'un objet, soit la valeur spéciale nil. Le treillis des valeurs est le suivant :



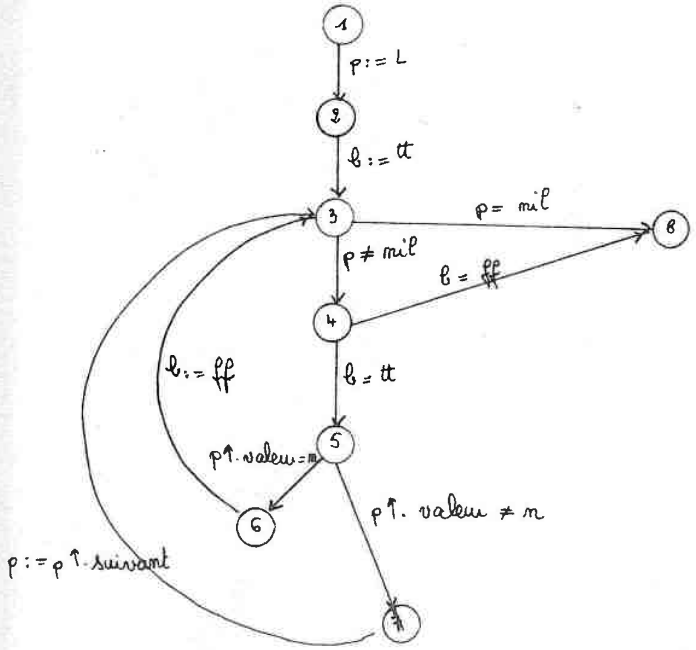

Nous pourrions terminer ici l'analyse en combinant les deux méthodes, ce qui, nous le savons, donne de meilleurs résultats que les méthodes appliquées séparément. Le treillis des valeurs possibles sera la plus petite famille de Moore contenant les éléments des treillis précédents ainsi que toutes les réunions.




ex. recherche d'une valeur n dans une liste L (où tout objet sera de la forme [valeur, suivant] )

```
p : = L ;
b : = vrai ;
tant que  ((p <> nil) et  (b)) faire
début
      si  (p↑. valeur = n) alors b : = faux
     sinon p : = p↑.suivant ;
fin ;
```

Le graphe associé est le suivant :




l'analyse des valeurs nous donne :

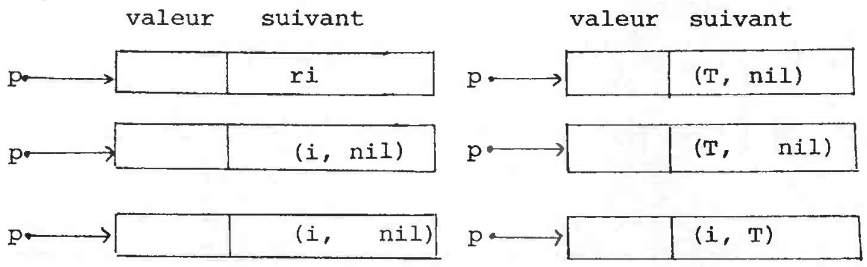
| | P | b |
|---|---|---|
| 1 | ⊥i | ⊥i |
| 2 | (i, T) | ⊥i |
| 3 | (i, T) | (i, T) |
| 4 | (i, ⊥ nil) | (i, T) |
| 5 | (i, ⊥ nil) | (i, tt) |
| 6 | (i, ⊥ nil) | (i, tt) |
| 7 | (i, ⊥ nil) | (i, tt) |
| 8 | (i, T) | (i, T) |

On constate que l'utilisation sur les arcs $5 \longrightarrow 6$ et $5 \longrightarrow 7$ du pointeur p est correcte car p est initialisé à une valeur non-nil. Mais aucune autre information n'est découverte ici.
C'est la raison pour laquelle nous allons essayer d'affiner les résultats en considérant, outre la variable pointeur elle-même, mais l'objet pointé. Cet objet pourra être un élément isolé ou bien une liste d'éléments que nous représenterons. Afin de simplifier l'exposé, nous nous limiterons aux éléments de la forme :

. un champ "valeur" et un champ "suivant" ; la généralisation se fait aisément dans le cas de plusieurs champs "valeur" ou plusieurs champs "suivant"

c) Description

Supposons que le pointeur est dans le cas (i, ≠ nil) alors, il pointe vers un élément qui a un champ "valeur" et u n champ "suivant" ; plusieurs cas surviennent, à savoir que ce champ est lui aussi une variable de type pointeur et ses valeurs possibles sont décrites dans le treillis schématisé plus haut. Voyons ces différents cas :

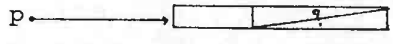
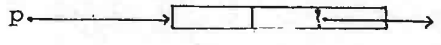
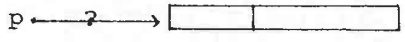
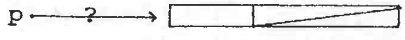
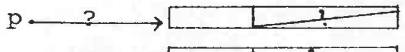



Mais dans le cas où p a pour valeur (T, nil), cela signifie que l'initialisation de p est indéterminée mais si p a été initialisée alors il pointe sur un élément (ou sur une liste d'éléments) c'est-à-dire que p peut se schématiser grâce aux éléments cités ci-dessus.

Appelons étape 1 l'étape de calcul décrite ici et revoyons les différentes valeurs possibles pour un pointeur $p_o$ et appelons $p_1$ le pointeur (s'il existe) à l'étape 1 le pointeur $p_1$ sera le contenu du champ "suivant" du premier élément de la liste pointée par $p_o$

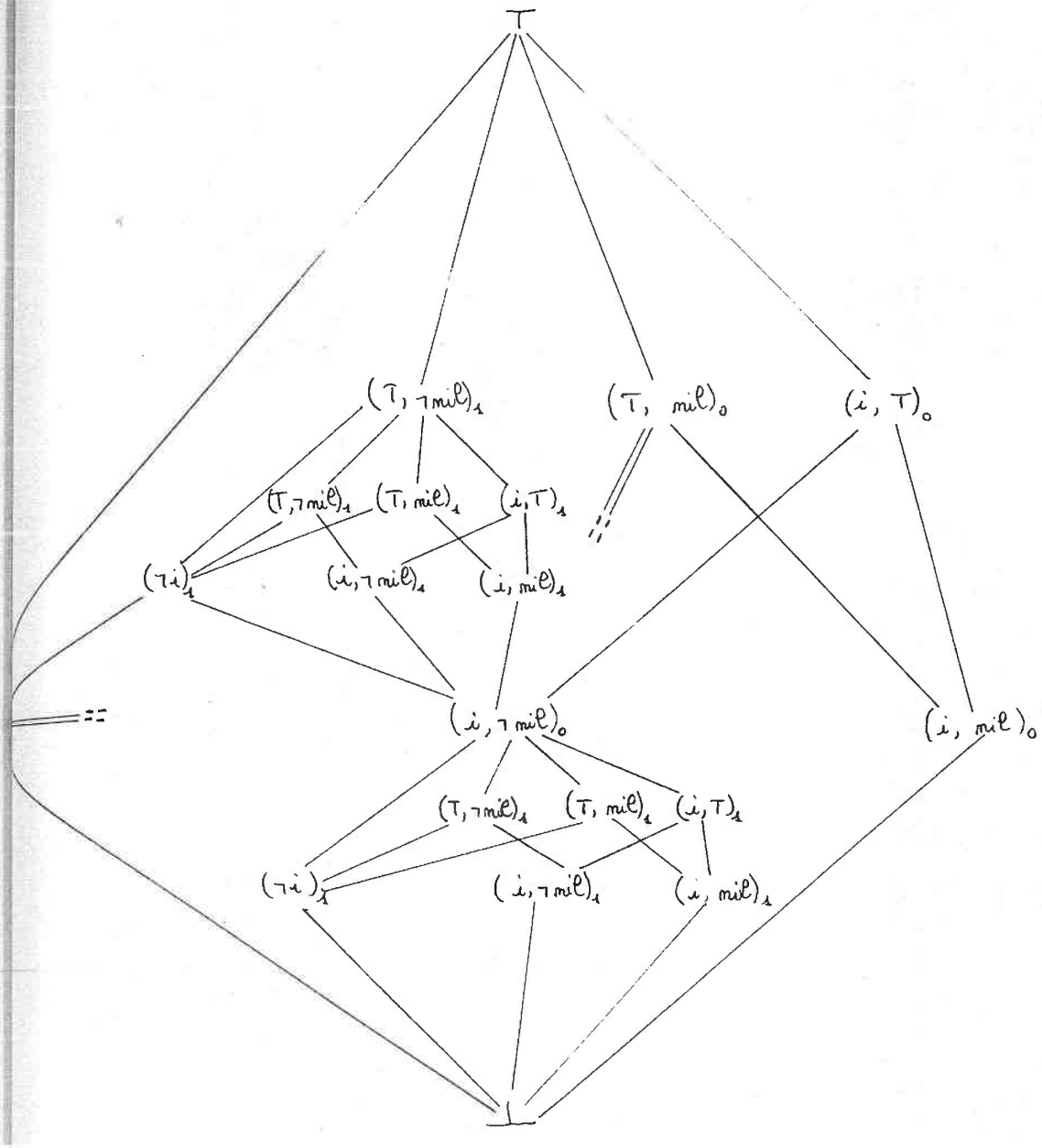
Pour l'instant nous ne soucions pas trop du champ "valeur" que nous pourrons étudier comme pour les éléments des tableaux c'est-à-dire considérer un seul élément pour tous les objets désignés.

Les différentes possibilités de ces pointeurs seront :

| étape 0 | étape 1 | schéma de liste |
|---|---|---|
| po : i | | |
| po : (i, nil) | | |
| po : (i, ⌐ nil) | p1 : ⌐ i |  |
| | p1 : (i, nil) |  |
| | p1 : (i, ⌐ nil) |  |
| | p1 : (i, T) |  |
| | p1 : (T, nil) |  |
| | p1 : (T, ⌐ nil) |  |
| po : (i, T) | | |
| po : (T, nil) | | |
| po : (T, ⌐ nil) | p1 : ⌐ i |  |
| | P1 : (i, nil) |  |
| | p1 : (i, ⌐ nil) |  |
| | p1 : (i, T) |  |
| | p1 : (T, nil) |  |
| | p1 : (T, ⌐ nil) |  |

Il apparaît clairement que dans les cas où $p_1$ est (ou peut être) ⌐ nil, on pourra faire une étude au niveau 2 et regarder le deuxième élément pointé : cet élément aura un champ "suivant" que l'on appellera $p_2$ et qui pourra avoir six valeurs différentes ; comme le cas où $p_1$ est ⌐ nil apparaît quatre fois, le nopbre de listes supplémentaires sera de 24. Et ainsi de suite, on peut construire $p_n$ si l'on veut obtenir une précision à l'étape n. Dans la pratique, nous allons vraisemblablement devoir nous limiter à n petit (!)

A l'étape 1, le treillis des valeurs pourra se construire facilement en remarquant que les six valeurs de $p_1$ forment un treillis donc le supremum est la valeur nil pour $p_o$ ; les valeurs auront un indice pour indiquer s'il s'agit de $p_o$ ou $p_1$.

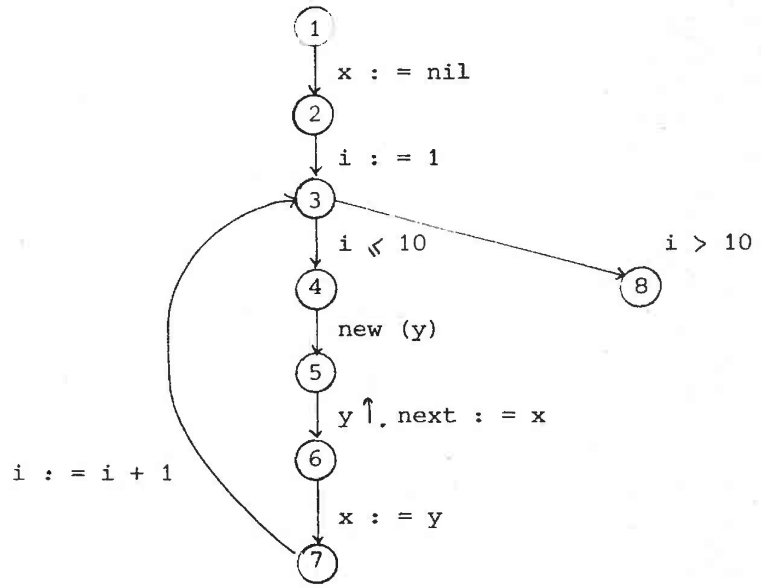



Le treillis au niveau 2 se déduit des treillis au niveau 1 en ajoutant les valeurs possibles pour $p_2$ dans le cas où $p_1$ vaut nil c'est-à-dire pour $(T, \neg \text{nil})_1$ (deux fois) et $(i, \neg \text{nil})_1$ (deux fois)

rq dans le schéma on omet le niveau 0 et $(T, \neg \text{nil})_1$ tient lieu de $(T, \neg \text{nil})_0 \wedge (T, \neg \text{nil})_1$ et plus en aval sur le schéma du treillis $(i, \neg \text{nil})_0$ $(T, \neg \text{nil})_1$

d) Exemple à l'étape 1

Etudions le programme dont le graphe de dépendance est le suivant :




L'analyse sémantique en avant nous fournit les résultats suivants :

| | x | y |
|---|---|---|
| 1 | $(\neg i)_0$ | $(\neg i)_0$ |
| 2 | $(i, nil)_0$ | $(\neg i)_0$ |
| 3 | $(i, T)_0$ | $(T, \neg nil)_0$ et $(i, T)_1$ |
| 4 | $(i, T)_0$ | $(T, \neg nil)_0$ et $(i, T)_1$ |
| 5 | $(i, T)_0$ | $(i, \neg nil)_0$ et $(\neg i)_1$ |
| 6 | $(i, T)_0$ | $(i, \neg nil)_0$ et $(i, T)_1$ |
| 7 | $(i, \neg nil)_0$ et $(i, T)_1$ | $(i, \neg nil)_0$ et $(i, T)_1$ |
| 8 | $(i, T)$. | $(T, \neg nil)_0$ et $(i, T)_1$ |

Conclusion : au noeud de sortie, le pointeur est initialisé, mais aucune valeur n'est précisée ; quant au pointeur y , on ne sait pas s'il a été initialisé mais s'il l'a été, il est $\neg$ nil et pointe sur un élément dont le champ "suivant" est un pointeur initialisé (sans qu'aucune valeur ne soit précisée là non plus).

## 3. Collections de pointeurs

Ce paragraphe résumé l'étude par Patrick et Radhia Cousot [ 76 ]

### a) Introduction

Considérons les déclarations suivantes :

```
type  liste = ↑ élément ;
      élément =  record
                    valeur : .. ;
                    suivant : liste ;
                 end ;
var  p, q : liste ;
```

Le compilateur ne peut pas dire si, à un moment donné de l'exécution, les variables de type pointeur p et q désignent des objets distincts. Il serait bien toutefois que l'analyse sémantique puisse déterminer ce genre d'information afin de repérer les effets de bord inattendus qui conduisent souvent à des erreurs à l'exécution et interdisent l'optimisation des programmes.

### b) notions de collection

Une collection sera représentée par l'ensemble des variables de type pointeur qui désigne un objet de cette collection. La construction de telles collections devra être telle que les collections soient disjointes.

ex soit p, q deux pointeurs vers une collection $C_1$
r, s, t pointant vers une collection $C_2$
nous noterons $C_1 = \{p, q\}$ et $C_2 = \{r, s, t\}$
au point de contrôle considéré (les informations obtenues sont locales), nous saurons que la partition est ( $\{p, q\}$ , $\{r, s, t\}$ )

Il reste à voir comment ces informations se propagent lors de l'analyse sémantique en fonction du graphe et des instructions du programme

### c) description

. jonction de chemins

l'union de deux partitions pose le problème de savoir si deux collections quelconques (l'une dans la première partition, l'autre dans la seconde partition) ont des pointeurs en commun, c'est-à-dire qu'il existe un objet pointé par certains éléments de la première collection et pointe par certains éléments de la seconde collection.

ex

$$(\{p_1 \ldots . p_n\},\ \{q, \ldots q_m\},\ \{r, \ldots r_i\}) \bar{u} (\{s_i \ldots . s_j, p_k\},\ \{t_i \ldots . t_1\})$$

avec $p_k \in \{p_1, \ldots p_n\}$

alors il faut supposer que les $p_z$ et les $s_p$, peuvent pointer vers les mêmes objets

la partition finale sera :

$$(\{p_1 \ldots . p_n, s, \ldots sj\}, \{q, \ldots q_m\}, \{r, \ldots r_j\}, \{t, \ldots t_1\}$$

. parmi les instructions du programme, nous remarquons que, si p, q sont des variables de type pointeur,

| | |
|---|---|
| les affectation | p : = nil, p : = q (où q vaut nil) |
| les conditionnelles | p = nil |
| les créations | new (p) |

ont la particularité de prendre la variable p indépendante de toute autre collection il faudra donc extraire p de la collection à laquelle elle appartenait avant l'instruction considérée et créer une collection réduite à p.

. les autres instructions : affectations p : = q, p : = q↑. suivant, p↑. suivant : = q, ...
retirent p de la collection à laquelle il pouvait appartenir avant l'instruction pour le mettre dans la collection à laquelle appartient q

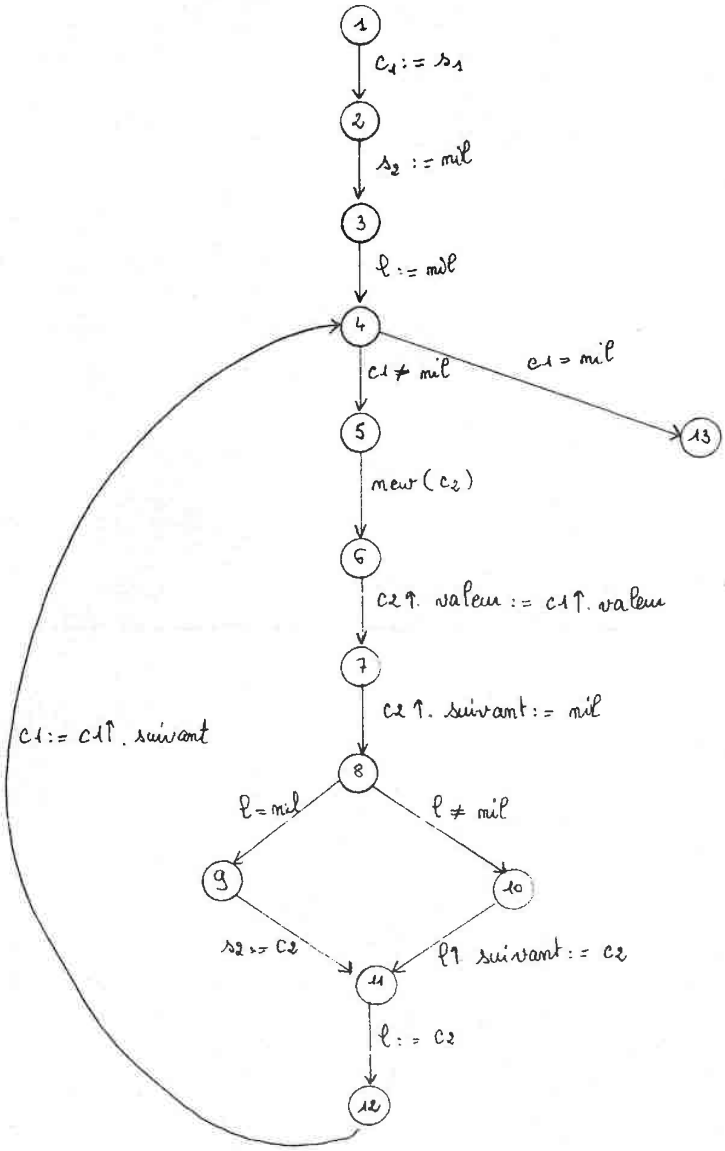
. les conditionnelles p <> nil n'apportent aucune information ni aucune modification

d) exemple

procédure copy (var $s_1$, $s_2$ : liste) ;
var $c_1$, $c_2$, l : liste ;

```
begin
      C 1 : = s 1 ; s 2 ; = nil ; l ; = nil ;
      while C1    ≠  nil do
      begin
           new (C2); C2↑ . valeur : = C1↑ . valeur ; C2↑. suivant : = nil ;
           if l = nil then  s2 : = C2
           else             l↑ . suivant = C2
           l : = C2 ; C1 : = C1↑ . suivant ;
      end ;
end.
```

Le graphe de dépendance associé à ce programme est :



Le système d'équations sémantiques associé est le suivant :

P 1 = ∅

P 2 = extraire (C1, P1) $\bar{U}$ ( { C1, s1 } )

P 3 = extraire (s2, P2)

P 4 = extraire ( 1, P3) U P 12

P 5 = P 4

P 6 = extraire (C2, P5)

P 7 = P 6

P 8 = P 7

P 9 = extraire (1, P8)

P10 = P 8

P11 = extraire (s2, P9) $\bar{U}$ ( { s2, C2 } ) U P 8 $\bar{U}$ ( { 1, C2 } )

P12 = extraire (1, $P_{11}$) $\bar{U}$ ( { 1, C2 } )

P13 = extraire (C1, P4)

La spécification d'entrée ∅ sera choisie la plus optimiste, à savoir s, et se sont dans la même collection ; la résolution de ce système nous donne le résultat final :

P 1 = ( { s1, s2 } , {C1 } , {C2 } , { 1 } )

P 2 = ( { s1, s2, c1 } , { C2 }, { 1 } )

P 3 = ( { s1, C1 } , { s2 } , {C2 } , { 1 } )

P 4 = ( { s1, C1 } , {s2, C2, 1 } )

P 5 = ( { s1, C1 } , {s2, C2, 1 } )

P 6 = ( { s1, C1 } , { s2, 1 } , { C2 )

P 7 = ( { s1, C1 } , { s2, 1 } , { C2 )

P 8 = ( { s1, C1 } , { s2, 1 } , { C2 )

P 9 = ( { s1, C1 } , { s2, 1 } , { C2 )

P10 = ( { s1, C1 } , { s2, 1 } , { C2 )

P11 = ( { s1, C1 } , { s2, 1 , C2 } )

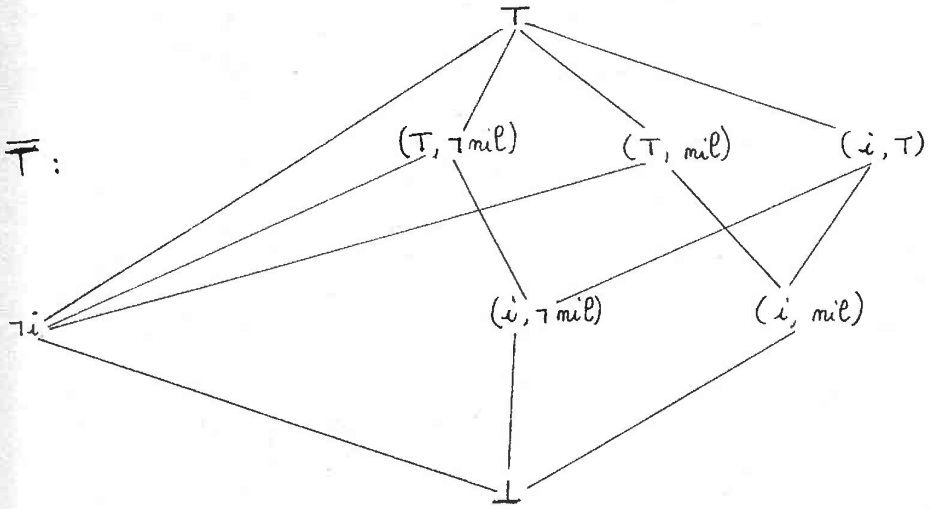
P12 = ( { s1, C1 } , { s2, 1 , C2 } )

P13 = ( { s1 }, {C1 } , { s2, 1 , C2 } )

4. Description

Le but de ce paragraphe est de regrouper les deux analyses que nous venons de développer. D'une part, l'étude des particuliers aux différents points de programme ; d'autre part, l'étude à un niveau déterminé de la liste

pointée par les divers variables de type pointeur qui se trouvent dans le programme. Le niveau donnera le nombre d'éléments de précision que nous voulons obtenir sur la liste : à l'étape 3 par ex, la variable p de type pointeur pourra pointer sur une liste quelconque mais on ne saura rien au delà du 3è élément. Le pointeur P aura des valeurs qui seront extraites du treillis suivant :




Tout élément de ce treillis se trouve sous la forme d'un couple (j, t) où $j \in \{i, \neg i, T\}$ et $t \in \{nil, \neg nil, T\}$ sauf les éléments $\perp$, $\neg i$ (un élément non initialisé n'a aucune raison d'avoir une valeur quelconque) et T = (T, T)

Dans le cas où la valeur t vaut ¬ nil, cela signifie que p pointe vers un élément qui a un champ "suivant" de type pointeur ; donc ce champ aura des valeurs qui seront extraites du même treillis ; et si ce champ à une valeur ¬ nil, alors il pointera lui aussi sur un élément de liste. Donc au niveau de précision k, il y aura pour chaque pointeur k valeurs appartenant à $\overline{T}$ avec toutefois la condition que la $i^{ème}$ valeur n'existe que si la $(i-1)^{ème}$ est ¬nil. Ceci reprend ce que nous avions dit précédemment.

A un point du programme (ou à un noeud du graphe de dépendance) seront associées d'une part une collection des variables de type pointeur du programme, d'autre part les valeurs (sous la forme d'un couple (j, t) où $j \in \{i, \neg i, T\}$ et $t \in \{nil, \neg nil, T\}$ ) de chaque pointeur, (s'il existe) de chaque champ "suivant", ainsi de suite selon le niveau de précision demandé.

Nous allons détaillé cette étude au niveau 1 pour les différentes instructions, soit une instruction I, $P_{i-1}$, et $P_i$ les points de programme encadrant cette instruction et faisons l'étude par cas :

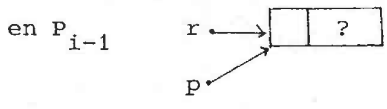
* I :: = (p : = nil)
  la partition en Pi sera définie par <u>extraire</u> ($P_{i-1}$, p)
  la valeur abstraite de p sera (i, nil) et bien sûr, p n pointe vers aucune liste dont la valeur suivante pourra être $\perp$ .

* I :: = (p : = q)
  si la valeur abstraite de q au niveau O est (i, nil) dans la partition en $P_1$ sera donnée par
  <u>extraire</u> ($P_{i-1}$, p)
  sinon la partitionen Pi sera définie par <u>extraire</u> (Pi-1, p) $\bar{U}$ ({ p, q }
  la valeur abstraite de P en $P_i$ sera la valeur abstraite de Q en $P_{i-1}$.
  Ces résultats ne peuvent bien entendu être vrais que si q est initialisé au niveau O sinon cela conduit à une erreur de l'exécution.

* I ::= (p:= Q ↑. suivant)
  Cette instruction ne peut être traitée que si q est initialisé à une valeur autre que nil au niveau O en $P_{i-1}$
  La partition en Pi sera donnée par <u>extraire</u> (Pi-1, p) $\bar{U}$ ({p, q} )
  La valeur abstraite au niveau O de p en Pi sera la valeur au niveau 1 de q en Pi-1. La valeur abstraite au niveau 1 de p en Pi ne peut être déterminée.

* I:: (p ↑. suivant : = nil)
  Cette instruction conduit à une erreur si p n'est pas (i, ¬ nil) au niveau O de $P_{i-1}$ (ou bien (T, ¬ nil), (i, T) , (T,T))
  La partition en $P_{i-1}$ n'est pas modifiée en $P_i$ car cette instruction nous met dans l'impossibilité d'isoler le pointeur p

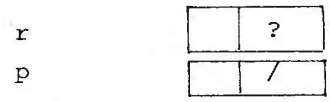
  prenons l'exemple suivant :

  en $P_{i-1}$ 

  la partition est {r,p}

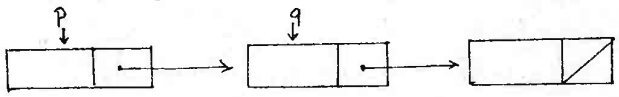
  { p ↑. suivant : = nil }

en $P_i$ la partition reste{r, p}

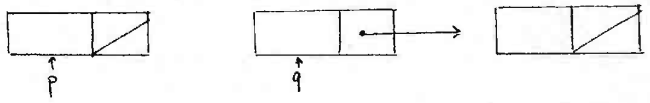



Le problème qui se pose est le suivant : la modification de la liste pointée par p change peut-être la liste pointée par r.
Il faudra donc modifier les valeurs abstraites de toutes les variables type pointeur qui pointent (peut-être) sur la même liste d'objets, c'est-à-dire tous les pointeurs qui sont dans la même collection.
On peut avoir les configurations suivantes

1.)

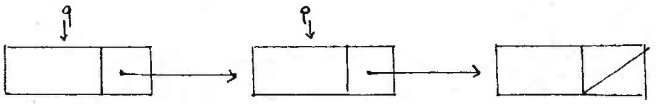



{ p ↑. suivant : = nil }

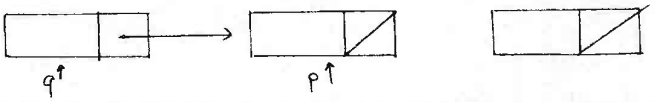



la valeur abstraite n'est pas modifiée

2.)

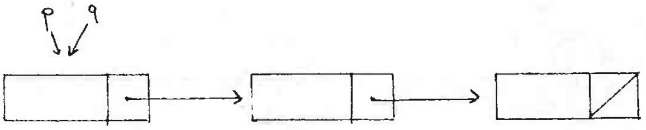



{ p ↑. suivant : = nil }

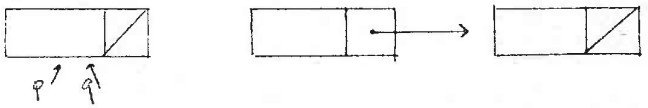



la valeur abstraite de q est modifiée au niveau 2 seulement

3.)




{ p ↑ . suivant : = nil }




la valeur abstraite de q est modifiée au niveau 1

4.)


{ p ↑ . suivant : = nil }


la valeur abstraite de q n'est pas modifiée.

Conclusions : les valeurs abstraites au niveau 1 de toutes les variables de type pointeur peuvent être modifiées
la partition n'est pas modifiée
et si v (q) est la valeur abstraite au niveau 1 avant l'instruction I, la valeur abstraite après exécution de l'instruction sera v (q) U $(i, nil)_1$ le niveau n'est évidemment pas modifié.

* I :: = (p ↑. suivant : = q)

Comme précédemment, si la variable de type pointeur p n'est pas initialisée ou est initialisée à la valeur nil au niveau O ; cela conduit à une erreur ; si q n'est pas initialisée, le résultat est le même.

Au niveau O, le pointeur p aura la même valeur en Pi qu'en $Pi_{-1}$ et sa valeur au niveau 1 sera celle de q au niveau O.

Les pointeurs p et q vont à présent pointer vers les mêmes objets donc la composante en Pi contiendra p et q et sera la réunion des deux composantes en $Pi_{-1}$ qui contenaient p et q.

Pour les variables qui étaient dans la composante de p en $Pi_{-1}$, la valeur va être modifiée puisque la structure de la liste l'a été. Pour ces pointeurs, la valeur abstraite au niveau O n'est pas modifiée ; au niveau 1, il faudra faire la réunion de l'ancienne valeur abstraite de p au niveau 1 et de la valeur abstraite de q au niveau O. C'est un cas plus général que la cas I :: = (p ↑. suivant : = nil).

* I :: = (new (p))
La partition en Pi sera modifiée et donnée par : extraire (p, Pi1) ; la fonction new crée un nouvel objet et p va pointer sur cet objet donc au niveau O, p vaudra (i, ¬ nil) mais aucune information n'est donnée sur l'objet en lui-même donc au niveau 1, p vaut ( ¬ i).

Les tests se traitent de façon analogue

* I :: = (p = nil)
La partition en Pi est définie par extraire (p, Pi-1)
Ce test conduira à une erreur si au niveau O, la valeur de P ne peut pas être nil ; donc les seules possibles sont (i, nil), (T, nil), (i, T) ou (T,T) il est bien évident que dans le cas présent, il n'existe plus rien au niveau 1, soit ⊥.

La valeur de p en Pi sera donc ( ?, nil).

* I = (p <> nil)

La partition en Pi est la même qu'en Pi-1.

Une erreur apparaîtra si, au niveau O, le pointeur p n'est pas initialisé ou est initialisé à nil exclusivement ; sinon la valeur de p en Pi sera inchangée par rapport à Pi-1.

* I = (p = q)

Ce test n'est valide que si en Pi-1 les variables p et q sont dans la même composante de la partition. Et dans ce cas, la partition en Pi est la même qu'en Pi-1.

Les valeurs de p et q seront égales en Pi et au niveau O, cette valeur sera le résultat de l'intersection des valeurs au niveau O de p et de q ; et si la définition le permet (c'est-à-dire, s'il peut exister un niveau 1 de p et q)

* I = (p <> q)

Ce test n'aura aucun effet sur les résultats en Pi sauf si des cas d'erreur apparaissent (p ou q non initialisés). Toutefois si au niveau O p = (i, nil et q = (i, nil) alors une impossibilité va être détectée.

Hormis ces cas particuliers, le contexte en Pi sera le contexte en Pi-1.

Nous nous limitons à ces quelques instructions, l'analyse à un niveau supérieur ou égal à 2 sera une généralisation de ce que nous venons de développer.

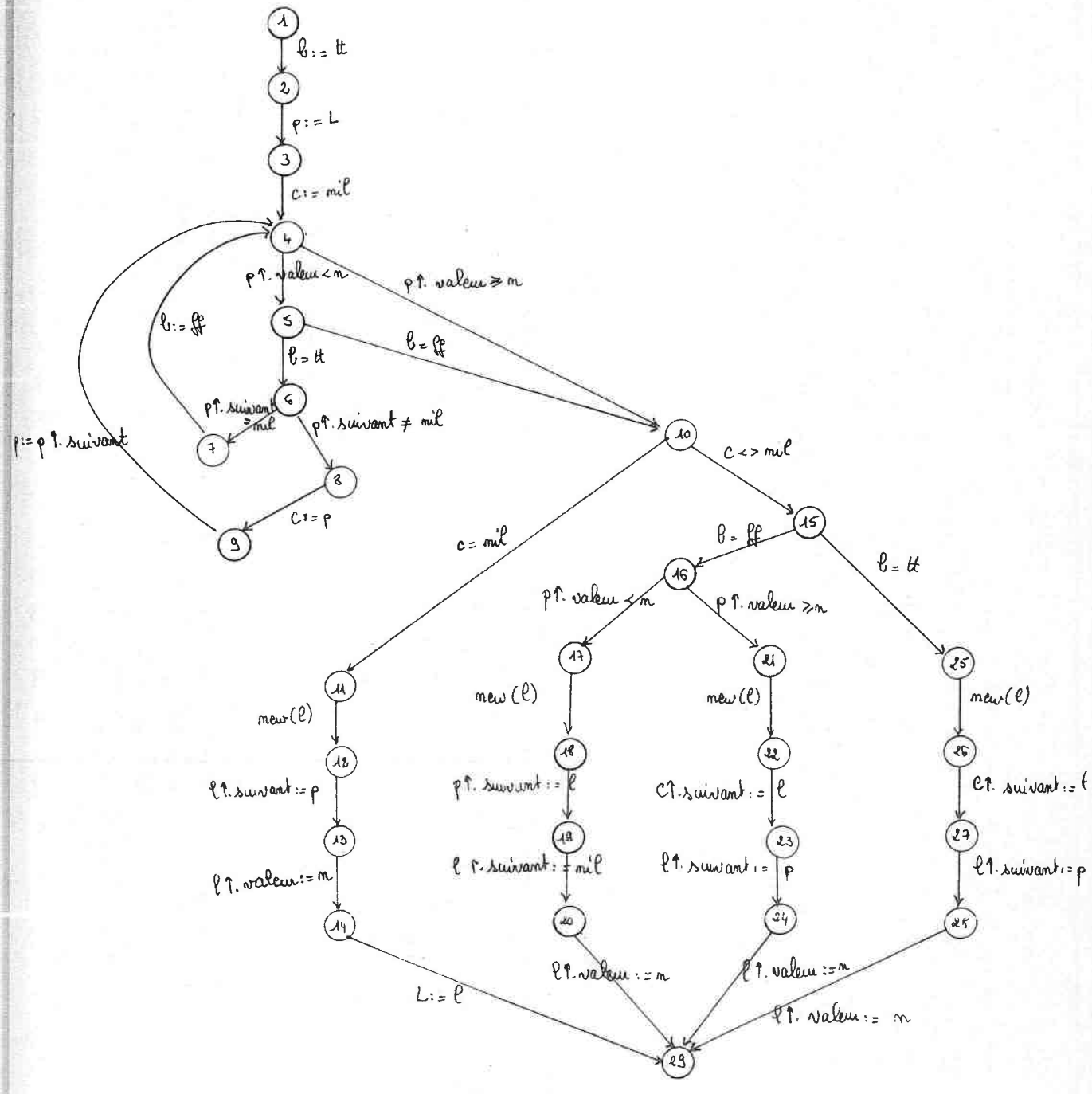
A la jonction de chemins, la partition sera la réunion des partitions sur les chemins concourants et la valeur au niveau O sera la réunion des valeurs O, (si la définition le permet) la valeur au niveau 1 sera la réunion des valeurs au niveau 1.

NB La réunion est bien entendu la réunion dans le treillis défini plus haut. Comme ce treillis est un treillis fini, il n'est pas nécessaire de définir un élargissement puisque la solution approchée sera connue en un nombre fini de pas.

5. Application à l'exemple

Nous allons prendre l'exemple de l'insertion d'un élément dans une liste d'éléments triés.

Le graphe est le suivant :




Nous supposerons qu'au départ la liste L contient au moin un élément ie L : (i, ⌝nil), (i, T) (pour les niveaux 0 et 1 dans cet ordre).

| partition | o P | 1 | o L | 1 | o C | 1 |
|---|---|---|---|---|---|---|
| {ρ}, {ℓ}, {c}, {⊥} | ¬i | ⊥ | ¬i | ⊥ | ¬i | ⊥ |
| {ρ}, {ℓ}, {c}, {⊥} | ¬i | ⊥ | ¬i | ⊥ | ¬i | ⊥ |
| {ρ,⊥}, {ℓ}, {c} | (i, ¬mil) | (i, T) | ¬i | ⊥ | ¬i | ⊥ |
| {ρ, ⊥, c} , {ℓ} | (i, ¬mil) | (i, T) | ¬i | ⊥ | (i, T) | (i, ¬mil) |
| {ρ, ⊥, c} , {ℓ} | (i, ¬mil) | (i, T) | ¬i | ⊥ | (i, T) | (i, ¬mil) |
| {ρ, ⊥, c} , {ℓ} | (i, ¬mil) | (i, T) | ¬i | ⊥ | (i, T) | (i, ¬mil) |
| {ρ, ⊥, c}, {ℓ} | (i, ¬mil) | (i, mil) | ¬i | ⊥ | (i, ¬mil) | (i, mil) |
| {ρ, ⊥, c}, {ℓ} | (i, ¬mil) | (i, ¬mil) | ¬i | ⊥ | (i, ¬mil) | (i, ¬mil) |
| {ρ, ⊥, c}, {ℓ} | (i, ¬mil) | (i, ¬mil) | ¬i | ⊥ | (i, ¬mil) | (i, ¬mil) |
| {ρ, ⊥, c}, {ℓ} | (i, ¬mil) | (i, T) | ¬i | ⊥ | (i, T) | (i, ¬mil) |
| {ρ, ⊥, c}, {ℓ} | (i, ¬mil) | (i, T) | ¬i | ⊥ | (i, T) | (i, ¬mil) |
| {ρ, ⊥}, {c}, {ℓ} | (i, ¬mil) | (i, T) | ¬i | ⊥ | (i, mil) | ⊥ |
| {ρ, ⊥}, {c}, {ℓ} | (i, ¬mil) | (i, T) | (i, ¬mil) | ¬i | (i, mil) | ⊥ |
| {ρ, ⊥, ℓ}, {c} | (i, ¬mil) | (i, T) | (i, ¬mil) | (i, ¬mil) | (i, mil) | ⊥ |
| {ρ, ⊥, ℓ}, {c} | (i, ¬mil) | (i, T) | (i, ¬mil) | (i, ¬mil) | (i, mil) | ⊥ |
| {ρ, ⊥, c}, {ℓ} | (i, ¬mil) | (i, T) | ¬i | ⊥ | (i, T) | (i, ¬mil) |
| {ρ, ⊥, c}, {ℓ} | (i, ¬mil) | (i, T) | ¬i | ⊥ | (i, ¬mil) | (i, ¬mil) |
| {ρ, ⊥, c}, {ℓ} | (i, ¬mil) | (i, T) | ¬i | ⊥ | (i, ¬mil) | (i, ¬mil) |
| {ρ, ⊥, c}, {ℓ} | (i, ¬mil) | (i, T) | ¬i | ⊥ | (i, ¬mil) | (i, ¬mil) |
| {ρ, ⊥, c}, {ℓ} | (i, ¬mil) | (i, T) | (i, ¬mil) | ¬i | (i, ¬mil) | (i, ¬mil) |
| {ρ, ⊥, c, ℓ} | (i, ¬mil) | (i, ¬mil) | (i, ¬mil) | ¬i | (T, ¬mil) | (T, ¬mil) |
| {ρ, ⊥, c, ℓ} | (i, T) | (i, T) | (i, ¬mil) | (i, mil) | (T, T) | (T, T) |
| {ρ, ⊥, c}, {ℓ} | (i, ¬mil) | (i, T) | ¬i | ⊥ | (i, ¬mil) | (i, ¬mil) |
| {ρ, ⊥, c}, {ℓ} | (i, ¬mil) | (i, T) | ¬i | ⊥ | (i, ¬mil) | (i, ¬mil) |
| {ρ, ⊥, c}, {ℓ} | (i, ¬mil) | (i, T) | (i, ¬mil) | ¬i | (i, ¬mil) | (i, ¬mil) |
| {ρ, ⊥, c, ℓ} | (i, ¬mil) | (i, T) | (i, ¬mil) | ¬i | (i, ¬mil) | (i, ¬mil) |
| {ρ, ⊥, c, ℓ} | (i, ¬mil) | (i, T) | (i, ¬mil) | (i, ¬mil) | (i, T) | (i, T) |
| {ρ, ⊥, c}, {ℓ} | (i, ¬mil) | (i, T) | ¬i | ⊥ | (i, ¬mil) | (i, ¬mil) |
| {ρ, ⊥, c}, {ℓ} | (i, ¬mil) | (i, T) | ¬i | ⊥ | (i, ¬mil) | (i, ¬mil) |
| {ρ, ⊥, c}, {ℓ} | (i, ¬mil) | (i, T) | (i, ¬mil) | ¬i | (i, ¬mil) | (i, ¬mil) |
| {ρ, ⊥, c, ℓ} | (T, ¬mil) | (T, T) | (i, ¬mil) | ¬i | (i, ¬mil) | (i, ¬mil) |
| {ρ, ⊥, c, ℓ} | (T, ¬mil) | (T, T) | (i, ¬mil) | (T, ¬mil) | (T, T) | (T, T) |
| {ρ, ⊥, c, ℓ} | (T, T) | (T, T) | (i, ¬mil) | (T, T) | (T, T) | (T, T) |

## 6. Généralisation

### a) notations et définitions

Nous allons généraliser ici la méthode d'analyse des variables de type pointeur à un niveau quelconque.

Soit n le niveau de précision demandé ($n \geq 0$), p le pointeur analysé.

La valeur abstraite v (p) obtenue par l'analyse sera définie par :

$v(p) = \bigwedge_i v(p)_i$ où $v(p)_i$ prend ses valeurs dans le treillis base $\bar{T}$ (VII. 4.)

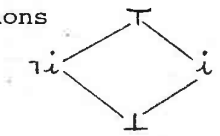
La valeur abstraite au niveau i ($0 \leq i \leq n$) de p est égale à :

- ($\perp$) i
- ($\neg$i) i
- ($\bar{i}$ (p), $\bar{v}$ (p)) i = ($\bar{i}$ (p) i, $\bar{v}$ (p) i)

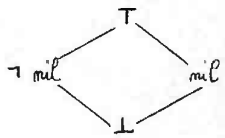
  dans ce cas seulement, une valeur pourra être donnée au niveau i + 1

Nous aurons besoin des opérations de réunion et d'intersection sur le treillis

des initialisations




et sur le treillis des valeurs




Ces fonctions seront notées $\bigcap$ (i), U (i) et $\bigcap$(v), U (v) respectivement.

Faisons l'étude des instructions I d'un programme entre les points k et k + 1

La partition en k sera notée P (k) et P (k +1) sera la partition après l'exécution de l'instruction et la collection contenant le pointeur sera notée P (l) (p) au point l.

### b) affectations

* I = (p : = nil)

- $P_{k+1}$ = extraire ($P_k$, p)
- $v(p)_0$ = (i, nil) et $\forall j > 0$ $v(p)_j = \perp$

* I :: = (p : = q)

- $P_{k+1}$ = extraire ($P_k$, p) $\bar{U}$ {p,q}
- si $v(q)_0 = \neg i$ alors $v(p)_0 = \perp$

  sinon $\forall j$ v (p) j = v (q) j.

* I :: = (p : = q ↑ . suivant)

. $P_{k+1}$ = extraire ($P_k$ , q) $\bar{U}$ { p,q }

. si v $(q)_o = \neg i$ ou $v(q)_o = (\bar{i}\ (q)_o, nil)$ alors v $(p)_o = \perp$

sinon $\forall\ 0 \leqslant j \leqslant n-1$ v (p) j = v (q) j + 1

si v $(q)_n = \perp$ ou $\neg i$ alors v (p) n = $\perp$

sinon si v (q) n = $(\bar{i}\ (q)_n, nil)$ alors v (p) n = $\perp$

sinon v (p) n = $\neg i$

fsi.

* I :: = (p ↑. suivant : = nil)

. $P_{k+i} = P_k$

. si v $(p)_o = \neg i$ ou v $(p)_o = (\bar{i}\ (p)_o, nil)$ alors v $(p)_o = \perp$

sinon . v $(p)_o$ inchangé

v $(p)_1$ = (i, nil)

v (p)j = $\perp$ $\forall\ 2 \leqslant j \leqslant n$

. $\forall$ pointeur $\alpha \in P\ (k)\ (p)$ (en l'absence de toute autre information, nous sommes obligés de considérer la solution la plus pessimiste)

. v $(\alpha)_o$ reste inchangé ainsi que tout v $(\alpha)$ j = $\perp$ $1 \leqslant j \leqslant n$

. $\forall\ 1 \leqslant j \leqslant n$ v(r)j = $(\bar{i}, \bar{v})$ où $\bar{i} = \bar{i}\ (r)\ j\ U\ (i)\ \{ i \}$

$\bar{v} = \bar{v}\ (r)\ j\ U\ (i)\ \{ nil \}$

fsi.

* I :: = (p ↑. suivant : = q)

. $P_{k+1} = P_k\ \bar{U}\ \{ p, q \}$

. si v $(p)_o = \neg i$ ou v $(p)_o = (\bar{i}\ (p), nil)$ ou v $(q)_o = \neg i$

alors v $(p)_o = \perp$

sinon . v $(p)_o$ reste inchangé

. v (p)j = v $(q)_{j-1}$ $1 \leqslant j \leqslant n$

. $\forall r \in P\ (k)_{(p)}$, v ( )$_o$ r-ste inchangé ainsi que tout v (r) j = $\perp$ 1 j $\leqslant$ n

$\forall\ 1 \leqslant j \leqslant n$ v (r) j = $(\bar{i}, \bar{v})$

où $\bar{i} = \bar{i}\ (r)\ j\ U_{(i)}\ (U(i)\ \bar{i}\ (q)_e)$ $l \geqslant 1$

$\bar{v} = \bar{v}\ (r)\ j\ U_{(v)}\left(U_{l \geqslant 1}^{(v)}\ \bar{v}\ (q)\ l\right)$

fsi

* I :: = (new (p))

. $P_{k+1}$ = extraire ($P_k$, p)

. v $(p)_0$ = (i, ¬ nil), v $(p)_1$ = ¬ i et v $(p)_j$ = ⊥ $2 \leqslant j \leqslant n$

Notons que - et ceci restera vrai pour l'étude des conditionnelles - lorsqu'une valeur abstraite où un niveau donné devient ⊥, alors les valeurs abstraites des niveaux supérieurs seront égales à .

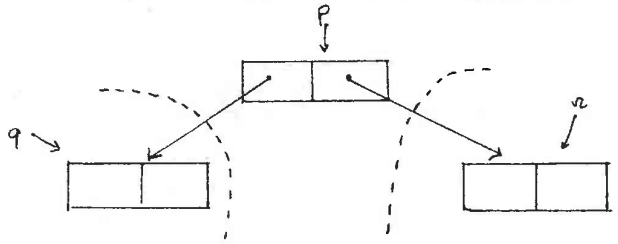
La généralisation que les affectations du type p : = q ↑.suivant↑..suivant se fera aisément en introduisant des variables intermédiaires.
Des informations locales concernant les positions relatives de deux pointeurs dans une liste éviteraient des pertes d'informations au niveau des valeurs .

ex si r : = p alors l'affectation p↑. suivant : = nil modifie la valeur abstraite au niveau 1 de p comme de r mais ne modifie pas la valeur abstraite au niveau 0 de r ., comme nous avons été obligés de le supposer.

Remarques :

des affinements des informations sont possibles

a. considérer des partitions à différents niveaux
par exemple, soit p une variable de type pointeur vers deux sous arbres il est possible de considérer la partition comme nous l'avons fait jusqu'à présent :




la partition sera
{ p, q, r }

mais il est aussi possible de considérer les pointeurs q et comme totalement indépendants et pour cela, nous introduisons des partitions à différents niveaux : en 1 : { p, q, r, .... }
en 2 : { q, .. } , { r, .. }
etc

de la sorte, il est possible de comparer de façon plus précise deux pointeurs ou de connaître les variables modifiées

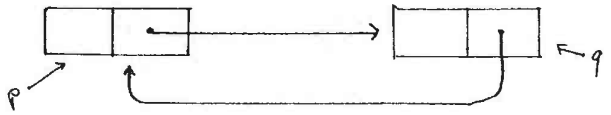
ex q ↑. champ 1 : = nil ne modifie en rien le pointeur r

b. l'instruction new (p) donne comme seule information : la valeur abstraite de p est égale à $(i, \neg nil)_o$
mais rien ne permet de distinguer cette valeur abstraite d'une autre valeur abstraite $(i, \neg nil)_o$ (- provenant d'une instruction new (q) -) il serait pourtant intéressant de pouvoir affirmer que ces deux pointeurs ne pointent pas sur le même objet.
Notons que dans certains cas, la collection de pointeurs ne peuvent pas donner d'informations.

Considérons la séquence suivante :

new (p) ;
new (q) ;
q ↑. suivant : = p ;
p ↑. suivant : = q ;

Faisons une étude au niveau 1 ;
si on ne fait aucune différence sur les résultats des divers appels de la fonction new, on obtient p : $(i, \neg nil)_o$, $(i, \neg nil)_1$
q : $(i, \neg nil)_o$, $(i, \neg nil)_1$
Mais ces pointeurs ne sont visiblement pas égaux :




la condition p = q ne peut donner aucune information ;
supposons à présent que nous gardions une trace des différents appels de la fonction new en numérotant les objets créés on obtiendrait alors

| | |
|---|---|
| new (p) ; | p : $(i, (\neg nil)_1)_o$ |
| new (q) ; | q : $(i, (\neg nil)_2)_o$ |
| q ↑. suivant : = p ; | q : $(i, (\neg nil)_2)_o$, $(i, (\neg nil)_1)_1$ |
| p ↑. suivant : = q ; | p : $(i, (\neg nil)_1)_o$, $(i, (\neg nil)_2)_1$ |

comme $(\neg nil)_1 \neq (\neg nil)_2$ , la condition p = q nous retourne alors une réponse négative.

c. <u>conditionnelles</u>

* I :: = ( p = nil)
  . $P_{k+1}$ = <u>extraire</u> $(P_k, p)$
  . <u>si</u> v $(p)_o = \neg i$ ou v $(p)_o = (\bar{i}(p), \neg nil)$ <u>alors</u> v $(p)_o = \perp$
  <u>sinon</u> v $(p)_o = (\bar{i}\ (p)_o, nil)$ et v $(p)_j = \perp$ $\quad 1 \leqslant j \leqslant n$

* I ::= (p <> nil)

. $P_{k+1} = P_k$

. si v $(p)_o = \neg i$ ou v (p) = ($\bar{i}$ (p), nil) alors v $(p)_o = \perp$
sinon v (p) est inchangée.

* I :: = (p = q)

. $P_{k+1} = P_k$

. si $P_k(p) \neq P_k$ (q) ou v $(p)_o = \neg i$ ou v $(q)_o = \neg i$
alors v $(p)_o$ = v $(q)_o = \perp$
sinon si $\exists j$ : $\bar{i}$ (p) j $\cap$(i) $\bar{i}$ (q) j = $\perp$ avec $\bar{i}$ (p) j et $\bar{i}$ $(q)_j \neq$ -
ou $\bar{v}$ (p) j (v) $\bar{v}$ (q) j = $\perp$ avec $\bar{v}$ (p) j et $\bar{v}$ $(q)_j \neq \perp$
alors v $(p)_o$ = v $(q)_o = \perp$
sinon $\forall\, 0 \leqslant j \leqslant n$ v (p)j = v (q)j = ($\bar{i}$ (p)j $\cap$(i) $\bar{i}$ (q)j,
$\bar{v}$ (p)j $\cap$(v) $\bar{v}$ (q)j).

* I :: = (p <> q)

. $P_{k+1} = P_k$

. si v $(p)_o = \neg i$ ou v $(q)_o = \neg i$ alors v $(p)_o$ = v $(q)_o = \perp$
sinon les valeurs abstraites de p et q restent inchangées.

d. jonction de chemins

Soit deux chemins k et l, m le chemin partant de la réunion en a :

. $P_m = P_k \bar{U} P_l$

. $\forall$ pointeur p, $\forall$ niveau j $(0 \leqslant j \leqslant n)$

v $(p)_j$ = ($\bar{i}/_k$ (p)j U (i) $\bar{i}/_l$ (p)j, $\bar{v}/_k$ (p)j U (v) $\bar{v}/_l$ (p) j )

ce qui se généralise en :

v (p) j = ( $\bar{U}$ (i) ($\bar{i}/$ (p)j ) , $\bar{U}$ (v) ($\bar{v}/$ (p) j)).
chemins chemin chemins chemin

## 7. Analyse en arrière

Le complément d'informations qu'apporterait l'analyse en arrière concerne les valeurs des variables de type pointeur en leur imposant à chaque niveau l'initialisation (dans le cas où elle vaut T) ou la valeur ¬ nil (dans le cas où elle vaut T) si l'analyse le juge nécessaire. Comme pour l'analyse en arrière avec l'initialisation 'III.5) l'analyse en arrière pour les variables de type pointeur n'est pas très difficile à mettre en oeuvre et les tests qu'il faudra placer à l'exécution se calculeront de la même façon que pour les variables entières.

## 8. Conclusion

Certaines informations complémentaires pourraient nous être utiles comme par exemple le fait que deux pointeurs dans une même collection qui pointent le même élément de la liste (et plus seulement la même classe d'objets) ceci nous éviterait une perte d'informations lors des affectations du type p ↑. suivant : = ...
Mais ce n'est qu'à l'usage que nous pourrons nous rendre compte de l'utilité de certains résultats intermédiaires ou de l'utilité d'employer des analyses plus fines (à des niveaux plus élevés).

## VIII - Conclusions

La mise en oeuvre de l'analyse sémantique a permis de prouver l'efficacité de la méthode ; l'analyseur fournit les renseignements que nous avons décrits dans cette thèse mais l'étude des variables de type pointeur n'est pas encore implantée. Pour donner un ordre d'idée, l'analyse d'un programme d'une centaine de lignes prend un temps compris entre 20 secondes et 2 minutes (respectivement avec 5 variables et 17 variables) analyse syntaxique et étude du graphe comprises. On peut donc obtenir à peu de frais des renseignements importants sur un programme.

La prochaine étude de cette recherche sera bien sûr l'implantation de l'étude des variables de type pointeur qui pourra être accompagnée de celle des variables booléennes considérées uniquement comme variables simples. Des problèmes techniques mais non théoriques sont posées sur les booléens construits à partir de relations entre variables du programme, par les variables de type enregistrement ou par les ensembles. Le problème théorique qu'il nous faudra résoudre sera l'étude des procédures non récursives (l'étude des procédures récursives sera résolu en combinant les résultats des recherches sur les pointeurs et sur les procédures non récursives).

Actuellement, les essais de l'analyseur portent sur une bonne centaine d'exemples choisis dans Manna [ 74 ] ainsi que dans le volume de Knuth : Sorting and Searching, vol. 3 Une rapide comparaison nous permet de constater l'efficacité de l'analyse. Si l'on considère comme instructions, les instructions significatives (i. e. les affectations, les tests, les écritures et non les instructions de saut, les déclarations, ...) et si l'on choisit un échantillon significatif d'exemples, nous avons un ensemble de 997 instructions.

L'échantillon pourra être trouvé au chapitre V

Un compilateur classique aurait placé 1976 tests à l'exécution, soit en moyenne 1,981 tests par instruction.

L'analyse sémantique fait apparaître 247 tests soit en moyenne 0,249 tests par instruction.

Le rapport du nombre de tests placés par le compilateur et du nombre de tests placés par l'analyseur est en moyenne de 7,9 (il varie entre 6 et 17) Certains tests placés par l'analyseur ne peuvent pas être découverts par un compilateur car ils concernent l'initialisation ou bien ils proviennent d'un bien existant entre des variables des programmes ; on en dénombre 70 (soit 28 %). 69 tests ont été placés dans les programmes afin d'éviter les débordements arithmétiques dans les différentes opérations. Les autres tests placés par l'analyseur proviennent du dimensionnement des tableaux (qui est statique en Pascal) ou bien de l'impossibilité pour l'analyseur de les supprimer par manque d'information (les liens qui existent entre les variables n'apparaissent que sous forme d'intervalles numériques).

Des essais plus importants vont être faits afin d'augmenter l'échantillonnage et de confirmer les premiers résultats.

Bibliographie


. Aho-Ullman [79] : A. V. Aho & J.D. Ullman
* Principles of compiler design

. Backhouse : R. Backhouse
* Syntax of Computer programs

. Cousot [75] : P. Cousot & R. Cousot
* Vérification statique de la cohérence dynamique des programmes 23 septembre 1975

. Cousot [76] : P. Cousot & R. Cousot
* Static determination of dynamic properties of generalized type unions A.C.M. Symposium on Language Design for Reliable Software Raleigh, North-Carolina, March 28-30, 1977

. Cousot [77] : P. Cousot
* Asynchronous iterative methods for solving a fixed point system of monotone equations in a complete lattice
R.R. 88 Laboratoire d'Informatique, U.S.M.G. B.P. 53 - 38041 Grenoble Cedex, septembre 1977

. Cousot [78] : P. Cousot
* Methodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis. Analyse sémantique des programmes
Thèse d'Etat Université Scientifique et Médicale de Grenoble - Institut National Polytechnique de Grenoble 21 mars 1978

. Cousot [79] : P. Cousot & R. Cousot
* Systematic design of program analysis frameworks
Laboratoire d'Informatique, U.S.M.G. , B.P. 53 X - 38041 Grenoble Cedex

. Cousot [81] : P. Cousot & R. Cousot
* Induction principles for proving invariance properties of programs Tool and Notions for Program Construction (ed. D. Neel) Cambridge University Press

. Hecht [77] : M.S. Hecht
* Flow analysis of Computer programs
Department of Computer Science, University of Maryland 1977

. Jensen [78] : K. Jensen & N. Wirth
* Pascal. User Manuel and Report
Springer - Verlag Newy-York Inc, 1978

. Karr [74] : M. Karr
* Decomposition of a graph into nested strongly connected components
Massachusetts Computer Associates, Inc.
Lakeside Office Park, Wakefield, Mass. 01880
Novembre 5, 1974 CAID - 7411 - 0511

. Koh [78] : H. Koh & H. Y. H. Chuang
* Finding a minimal set of base paths of a program
International Journal of Computer and Information Science Vol. 8, Nb 6 1978

. Leblanc [80] : R. J. Leblanc & C. N. Fischer
* An analysis of run-time errors in Pascal programs
Computer Sciences Technical Report ≠ 384
University of Wisconsin - Madison April 1980

. Manna [74] : Z. Manna
* Mathematical Theory of Computation
Mc Graw-Hill Computer Science Series 1974

. Manna [80] : Z. Manna & R. Waldinger
* Problematic factures of programming languages : a situational - calculus appoach
part 1 : assignment statements sept 1980

. Rosen [80] : B. K. Rosen
* Robust linear algouthmes for cutsets
RC 8617 (≠ 37 640) 12 - 22 - 1980
Computer Sciences Departement
I B M Thomas J. Watson Research Center
Yoktown Heights, New-York 10 598

. Schamir [79] : A. Shamir
* A linear time algorithm for finding minimum cutsets in reductible graphs
SIAM J. COMPUT. Vol. 8, N° 4, Novembre 1979

. Sharir [81] : M. Sharir
* A strong - connectivity algorithm and its applications in data flow analysis
Comp. & Maths. with Vol. 7 pp 67-72
Pergamon Press Ltd 1981

. Tarjan [72] : R. E. Tarjan
* Depth - first and linear graph algorithms
SIAM J. COMPUT. Vol. 1, N° 2, June 1972

. Tarjan [74] : R. E. Tarjan
* Finding dominators in directed graphs
SIAM J. COMPUT. Vol. 3, N° 1, March 1974

. Tarjan [81] : R. E. Tarjan
* A unified approach to path problems
Journal of the Association for Computing Machinery
Vol. 28, N° 3, July 1981 pp 577 - 593

. Tarjan [81] : R. E. Tarjan
* Fast algorithms for solving path problems
Journal of the Association for computing Machinery
Vol. 28, N° 3, July 1981 pp 594 - 614

. Walter [80] : W. M. Walter & L. R. Carter
* An analysis/synthesis interface for Pascal Compilers
University of Colorado
Boulder, Colorado 80 309

. Welsh [77] : J. Welsh
* Economic range checking in Pascal
Dept. of Computer Science, Queen's University
Belfast, Northern Ireland, Oct. 1977

. Wortman [79] : D. B. Wortman
* On legality assertions in Euclid
IEEE Transactions on Software Engineering,
July 1979

institut
national
polytechnique
de lorraine



Le Président,

N/Réf. : Scol.

AUTORISATION DE SOUTENANCE DE THESE DE DOCTORAT 3ème CYCLE

-:-:-:-:-:-:-:-:-:-:-:-:-

VU LE RAPPORT ETABLI PAR :

Monsieur le Professeur COUSOT Patrick

le Président de l'Institut National Polytechnique de Lorraine autorise :

Monsieur JUNG Jean-Pierre

à soutenir, devant l'I.N.P.L., une thèse de Doctorat intitulée :

"UN PROTOTYPE D'ANALYSEUR SEMANTIQUE POUR UN SOUS-ENSEMBLE DE PASCAL"

en vue de l'obtention du titre de DOCTEUR 3ème CYCLE

Spécialité "INFORMATIQUE"

Fait à NANCY, le 28 Mai 1983

Le Président de l'I.N.P.L.

INSTITUT NATIONAL POLYTECHNIQUE DE LORRAINE NANCY

M. LUCIUS

Vu collationné et certifié conforme à l'original qui nous a été présenté
METZ, le 5 JUIL. 1983
Pour le Maire
l'Agent communal délégué

MAIRIE DE METZ 57 (Moselle)

D. JEAN