De l'optimisation des programmes

Matija Exel

THESE

présentée à

UNIVERSITE SCIENTIFIQUE ET MEDICALE DE GRENOBLE

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

POUR OBTENIR LE GRADE DE
DOCTEUR DE 3ème CYCLE

Matija EXEL

DE L'OPTIMISATION DES PROGRAMMES

Thèse soutenue le 6 octobre 1975 devant la Commission d'Examen.

Président : Monsieur L. BOLLIET

Rapporteur : Monsieur M. GRIFFITHS

Examinateurs : Monsieur G. VEILLON
Monsieur P. JORRAND

UNIVERSITE SCIENTIFIQUE ET MEDICALE DE GRENOBLE

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

M. Michel SOUTIF	Présidents	M. Louis NEEL
M. Gabriel CAU	Vice-Présidents	MM. Lucien BONNETAIN
		Jean BENOIT

MEMBRES DU CORPS ENSEIGNANT DE L'U.S.M.G.

PROFESSEURS TITULAIRES

MM.	ANGLES D'AURIAC Paul	Mécanique des fluides
	ARNAUD Paul	Chimie
	AUBERT Guy	Physique
	AYANT Yves	Physique approfondie
Mme	BARBIER Marie-Jeanne	Electrochimie
MM.	BARBIER Jean-Claude	Physique expérimentale
	BARBIER Reynold	Géologie appliquée
	BARJON Robert	Physique nucléaire
	BARNOUD Fernand	Biosynthèse de la cellulose
	BARRA Jean-René	Statistiques
	BARRIE Joseph	Clinique chirurgicale
	BEAUDOING André	Clinique de Pédiatrie et Puériculture
	BERNARD Alain	Mathématiques Pures
Mme	BERTRANDIAS Françoise	Mathématiques Pures
MM.	BEZES Henri	Pathologie chirurgicale
	BLAMBERT Maurice	Mathématiques Pures
	BOLLIET Louis	Informatique (IUT B)
	BONNET Georges	Electrotechnique
	BONNET Jean-Louis	Clinique ophtalmologique
	BONNET-EYMARD Joseph	Pathologie médicale
	BOUCHERLE André	Chimie et toxicologie
	BOUCHEZ Robert	Physique nucléaire
	BOUSSARD Jean-Claude	Mathématiques appliquées
	BRAVARD Yves	Géographie
	CABANEL Guy	Clinique rhumatologique et hydrologie
	CALAS François	Anatomie
	CARLIER Georges	Biologie végétale
	CARRAZ Gilbert	Biologie animale et pharmacodynamie
	CAU Gabriel	Médecine légale et toxicologie
	CAUQUIS Georges	Chimie organique
	CHABAUTY Claude	Mathématiques Pures
	CHARACHON Robert	Clinique Oto-Rhino-Laryngologique
	CHATEAU Robert	Thérapeutique (Neurologie)
	CHIBON Pierre	Biologie animale
	COEUR André	Pharmacie chimique et chimie analytique
	CONTAMIN Robert	Clinique gynécologique
	COUDERC Pierre	Anatomie pathologique
	CRAYA Antoine	Mécanique
Mme	DEBELMAS Anne-Marie	Matière médicale
MM.	DEBERMAS Jacques	Géologie générale
	DEGRANGE Charles	Zoologie
	DELORMAS Pierre	Pneumo-Phtisiologie
	DEPORTES Charles	Chimie minérale
	DESRE Pierre	Métallurgie
	DESSAUX Georges	Physiologie animale
	DODU Jacques	Mécanique appliquée

MM.	DOLIQUE Jean-Michel	Physique des plasmas
	DREYFUS Bernard	Thermodynamique
	DRUCROS Pierre	Cristallographie
	DUGOIS Pierre	Clinique de dermatologie et syphiligr
	FAU René	Clinique neuro-psychiatrique
	GAGNAIRE Didier	Chimie physique
	GALLISSOT François	Mathématiques pures
	GALVANI Octave	Mathématiques pures
	GASTINEL Noël	Mathématiques appliquées
	GAVEND Michel	Pharmacologie
	GEINDRE Michel	Electroradiologie
	GERBER Robert	Mathématiques pures
	GERMAIN Jean-Pierre	Mécanique
	GIRAUD Pierre	Géologie
	JANIN Bernard	Géographie
	KAHANE André	Physique Générale
	KLEIN Joseph	Mathématiques pures
	KOSZUL Jean-Louis	Mathématiques pures
	KRAVTCHENKO Julien	Mécanique
	KUNTZMANN Jean	Mathématiques appliquées
	LACAZE Albert	Thermodynamique
	LACHARME Jean	Biologie végétale
	LAJZEROWICZ Joseph	Physique
	LATREILLE René	Chirurgie générale
	LATURAZE Jean	Biochimie pharmaceutique
	LAURENT Pierre-Jean	Mathématiques appliquées
	LEDRU Jean	Clinique médicale B
	LLIBOUTRY Louis	Géophysique
	LONGEQUEUE Jean-Pierre	Physique nucléaire
	LOUP Jean	Géographie
Mle	LUTZ Elisabeth	Mathématiques pures
	MALGRANGE Bernard	Mathématiques pures
	MALINAS Yves	Clinique obstétricale
	MARTIN-NOEL Pierre	Seméiologie médicale
	MAZARE Yves	Clinique médicale A
	MICHEL Robert	Minéralogie et pétrographie
	MICOUD Max	Clinique maladies infectieuses
	MOURIQUAND Claude	Histologie
	MOUSSA André	Chimie nucléaire
	MULLER Jean-Michel	Thérapeutique (néphrologie)
	NEEL Louis	Physique du solide
	OZENDA Paul	Botanique
	PAYAN Jean-Jacques	Mathématiques pures
	PEBAY-PEYROULA Jean-Claude	Physique
	RASSAT André	Chimie systématique
	RENARD Michel	Thermodynamique
	RINALDI Renaud	Physique
	DE ROUGEMONT Jacques	Neuro-chirurgie
	SEIGNEURIN Raymond	Microbiologie et hygiène
	SENGEL Philippe	Zoologie
	SIBILLE Robert	Construction mécanique
	SOUTIF Michel	Physique générale
	TANCHE Maurice	Physiologie
	TRAYNARD Philippe	Chimie générale
	VAILLANT François	Zoologie
	VALENTIN Jacques	Physique nucléaire
	VAUQUOIS Bernard	Calcul électronique
Mme	VERAIN Alice	Pharmacie galénique
MM.	VERAIN André	Physique
	VEYRET Paul	Géographie
	VIGNAIS Pierre	Biochimie médicale
	YOCCOZ Jean	Physique nucléaire théorique

PROFESSEURS ASSOCIES

MM.	CHEEKE John	Thermodynamique
	COPPENS Philip	Physique
	CORCOS Gilles	Mécanique
	CRABBE Pierre	CERMO
	GILLESPIE John	I.S.N.
	ROCKAFELLAR Ralph	Mathématiques appliquées

PROFESSEURS SANS CHAIRE

Mlle	AGNIUS-DELORD Claudine	Physique pharmaceutique
	ALARY Josette	Chimie analytique
MM.	AMBROISE-THOMAS Pierre	Parasitologie
	BELORIZKY Elie	Physique
	BENZAKEN Claude	Mathématiques appliquées
	BERTRANDIAS Jean-Paul	Mathématiques pures
	BIAREZ Jean-Pierre	Mécanique
	BILLET Jean	Géographie
Mme	BONNIER Jane	Chimie générale
MM.	BOUCHET Yves	Anatomie
	BRUGEL Lucien	Energétique
	CONTE René	Physique
	DEPASSEL Roger	Mécanique des fluides
	GAUTHIER Yves	Sciences biologiques
	GAUTRON René	Chimie
	GIDON Paul	Géologie et Minéralogie
	GLENAT René	Chimie organique
	GROULADE Joseph	Biochimie médicale
	HACQUES Gérard	Calcul numérique
	HOLLARD Daniel	Hématologie
	HUGONOT Robert	Hygiène et Méd. Préventive
	IDELMAN Simon	Physiologie animale
	JOLY Jean-René	Mathématiques pures
	JULLIEN Pierre	Mathématiques appliquées
Mme	KAHANE Josette	Physique
MM.	KUHN Gérard	Physique
	LOISEAUX Jean	Physique nucléaire
	LUU-DUC-Cuong	Chimie organique
	MAYNARD Roger	Physique du solide
	PELMONT Jean	Biochimie
	PERRIAUX Jean-Jacques	Géologie et minéralogie
	PFISTER Jean-Claude	Physique du solide
Mlle	PIERY Yvette	Physiologie animale
MM.	RAYNAUD Hervé	Mathématiques appliquées
	REBECQ Jacques	Biologie (CUS)
	REVOL Michel	Urologie
	REYMOND Jean-Charles	Chirurgie générale
	RICHARD Lucien	Biologie végétale
Mme	RINAUDO Marguerite	Chimie macromoléculaire
MM.	ROBERT André	Chimie papetière
	SARRAZIN Roger	Anatomie et chirurgie
	SARROT-REYNAULD Jean	Géologie
	SIROT Louis	Chirurgie générale
Mme	SOUTIF Jeanne	Physique générale
MM.	VIALON Pierre	Géologie
	VAN CUTSEM Bernard	Mathématiques appliquées

MAITRES DE CONFERENCES ET MAITRES DE CONFERENCES AGREGES

MM.	AMBLARD Pierre	Dermatologie
	ARMAND Gilbert	Géographie
	ARMAND Yves	Chimie
	BARGE Michel	Neurochirurgie
	BEGUIN Claude	Chimie organique
Mme	BERIEL Hélène	Pharmacodynamique
M.	BOUCHARLAT Jacques	Psychiatrie adultes
Mme	BOUCHE Liane	Mathématiques (CUS)
MM.	BRODEAU François	Mathématiques (IUT B)
	BUISSON Roger	Physique
	BUTEL Jean	Orthopédie
	CHAMBAZ Edmond	Biochimie médicale
	CHAMPETIER Jean	Anatomie et organogénèse
	CHARDON Michel	Géographie
	CHERADAME Hervé	Chimie papetière
	CHIAVERINA Jean	Biologie appliquée (EFP)
	COHEN-ADDAD Jean-Pierre	Spectrométrie physique
	COLOMB Maurice	Biochimie médicale
	CORDONNIER Daniel	Néphrologie
	COULOMB Max	Radiologie
	CROUZET Guy	Radiologie
	CYROT Michel	Physique du solide
	DELOBEL Claude	M.I.A.G.
	DENIS Bernard	Cardiologie
	DOUCE Roland	Physiologie végétale
	DUSSAUD René	Mathématiques (CUS)
Mme	ETERRADOSSI Jacqueline	Physiologie
MM.	FAURE Jacques	Médecine légale
	FONTAINE Jean-Marc	Mathématiques pures
	GAUTIER Robert	Chirurgie générale
	GENSAC Pierre	Botanique
	GIDON Maurice	Géologie
	GRIFFITHS Michaël	Mathématiques appliquées
	GROS Yves	Physique (stag.)
	GUITTON Jacques	Chimie
	HICTER Pierre	Chimie
	IVANES Marcel	Electricité
	JALBERT Pierre	Histologie
	KOLODIE Lucien	Hématologie
	KRAKOWIAK Sacha	Mathématiques appliquées
Mme	LAJZEROWICZ Jeannine	Physique
MM.	LEROY Philippe	Mathématiques
	MACHE Régis	Physiologie végétale
	MAGNIN Robert	Hygiène et médecine préventive
	MARECHAL Jean	Mécanique
	MARTIN-BOUYER Michel	Chimie (CUS)
	MICHOULIER Jean	Physique (IUT A)
Mme	MINIER Colette	Physique
MM.	NEGRE Robert	Mécanique
	NEMOZ Alain	Thermodynamique
	PARAMELLE Bernard	Pneumologie
	PECCOUD François	Analyse (IUT B)
	PEFFEN René	Métallurgie
	PERRET Jean	Neurologie
	PHELIP Xavier	Rhumatologie
	RACHAIL Michel	Médecine interne
	RACINET Claude	Gynécologie et obstétrique
	RAMBAUD Pierre	Pédiatrie
Mme	RENAUDET Jacqueline	Bactériologie
MM.	ROBERT Jean-Bernard	Chimie-Physique

MM.	ROMIER Guy	Mathématiques (IUT B)
	SHOM Jean-Claude	Chimie générale
	STIEGLITZ Paul	Anesthésiologie
	STOEBNER Pierre	Anatomie pathologique
	VROUSOS Constantin	Radiologie

MAITRES DE CONFERENCES ASSOCIES

MM.	COLE Antony	Sciences nucléaires
	FORELL César	Mécanique
	MOORSANI Kishin	Physique

CHARGES DE FONCTIONS DE MAITRES DE CONFERENCES

MM.	BOST Michel	Pédiatrie
	CONTAMIN Charles	Chirurgie thoracique et cardio-vasculair
	FAURE Gilbert	Urologie
	MALLION Jean-Michel	Médecine du travail
	ROCHAT Jacques	Hygiène et hydrologie

Fait à Saint Martin d'Hères, OCTOBRE 1974.

"MEMBRES DU CORPS ENSEIGNANT DE L'I.N.P.G."

PROFESSEURS TITULAIRES

MM. BENOIT Jean	Radioélectricité
BESSON Jean	ELectrochimie
BONNETAIN Lucien	Chimie Minérale
BONNIER Etienne	Electrochimie, Electrométallurgie
BRISSONNEAU Pierre	Physique du solide
BUYLE-BODIN Maurice	ELectronique
COUMES André	Radioélectricité
FELICI Noël	Electrostatique
PAUTHENET René	Physique du solide
PERRET René	Servomécanismes
SANTON Lucien	Mécanique
SILBER Robert	Mécanique des fluides

PROFESSEUR ASSOCIE

M. BOUDOURIS Georges	Radioélectricité

PROFESSEURS SANS CHAIRE

MM. BLIMAN Samuel	Electronique
BLOCH Daniel	Physique du solide et cristallographie
COHEN Joseph	Electrotechnique
DURAND Françis	Metallurgie
MOREAU René	Mécanique
POLOUJADOFF Michel	ELectrotechnique
VEILLON Gérard	Informatique fondamentale et appliquée
ZADWORNY François	Electronique

MAITRES DE CONFERENCES

MM. BOUVARD Maurice	Génie mécanique
CHARTIER Germain	Electronique
FOULARD Claude	Automatique
GUYOT Pierre	Chimie minérale
JOUBERT Jean Claude	Physique du solide
LACOUME Jean Louis	Géophisique
LANCIA Roland	Physique atomique
LESPINARD Georges	Mécanique
MORET Roger	Electrotechnique nucléaire
ROBERT François	Analyse numérique
SABONNADIERE Jean Claude	Informatique fondamentale et appliquée
Mme SAUCIER Gabrièle	Informatique fondamentale et appliquée

MAITRE DE CONFERENCES ASSOCIE

M. LANDAU Ioan Doré	Automatique

CHARGE DE FONCTIONS DE MAITRES DE CONFERENCES

M. ANCEAU François	Mathématiques appliquées

Fait à Saint Martin d'Hères JANVIER 1974

TABLE DES MATIERES
-:-:-:-:-:-:-:-:-:-:-

Je tiens à remercier :

Monsieur L. BOLLIET qui m'a fait l'honneur de présider le jury de cette thèse, Messieurs G. VEILLON et Ph. JORRAND qui ont bien voulu accepter de faire partie du jury, Monsieur B. WILLIS qui m'a fourni les idées de départ de cette thèse, Monsieur M. GRIFFITHS, mon directeur de thèse, dont les encouragements amicaux m'ont été précieux.

Je tiens à exprimer ma reconnaissance à Michel et Hélène qui m'ont aidé à améliorer la présentation de ma thèse.

Je remercie enfin vivement Renée CARRE PIERRAT pour sa patience et son travail de dactylographie qui a conduit en un temps record à la réalisation matérielle de cet ouvrage.

M. EXEL

A Hélène

1 - INTRODUCTION -

Pendant mon séjour à Grenoble j'ai eu l'occasion de participer aux travaux de Monsieur Bruce WILLIS (voir sa thèse [63]) qui m'a suggéré de m'attaquer aux problèmes d'optimisation des programmes.

Mon travail de recherche sur ce sujet m'a conduit à cette thèse.

1.1 - Amélioration des programmes -

La présente thèse a pour sujet l'amélioration des programmes écrits dans des langages de programmation de type procédural (PL/1, ALGOL, FORTRAN ...). Dans la littérature, le terme qui semble être accepté pour désigner les traitements dont nous parlerons est celui d'optimisation des programmes. Il serait toutefois plus exact d'utiliser le terme d'amélioration de la performance et ceci pour les deux raisons négatives que nous développerons ainsi :

1) - Les auteurs, dans la littérature sur l'optimisation, n'ont pas cherché, et nous ne chercherons pas non plus, à définir mathématiquement une fonction d'optimalité d'un programme - c'est-à-dire une fonction qui mesure la qualité du programme relativement à des critères choisis -.

Ceci demanderait tout d'abord, en effet, un choix de critères d'optimalité d'un programme. La notion d'optimalité doit être envisagée dans le cadre d'une conception du programme, conception qui dépend essentiellement de la nature de l'utilisation du programme. On peut vouloir un programme facilement modifiable, restructurable (modulaire), adaptable à de nouvelles conditions d'exploitation, plus ou moins sûr...

Le cadre d'utilisation du programme étant précisé, un programme optimal est un programme efficace relativement à ce cadre. Les critères de l'efficacité sont ceux d'une bonne utilisation des ressources et du système et sont donc fondés essentiellement sur :

Le facteur-temps : - temps de compilation du programme-source en programme-objet.

- temps d'exécution du programme en fonction des données à l'entrée.

Le facteur-espace : - espace nécessaire pour la compilation du programme ;
- espace nécessaire pour le code du programme et ses données statiques ;

- espace minimal nécessaire pour le code du programme à l'exécution (on pense ici aux techniques de recouvrement).
- espace nécessaire pour les données dynamiques au cours d'exécution du programme (zones de transfert, recouvrements dynamiques, piles, etc...).

De nouveau ces critères d'efficacité peuvent être sujets au cadre d'utilisation du programme : on peut vouloir une compilation rapide, une exécution rapide, un espace minimal....

Les différents paramètres d'une expression d'efficacité que nous avons énumérés ci-dessus, selon leurs dépendances au facteur temps ou espace, peuvent ainsi recevoir une pondération en fonction du cadre d'utilisation.

Comme on le voit, la notion d'optimalité est conçue comme une notion d'efficacité dans un cadre donné, cadre dont le choix dépend essentiellement de considérations subjectives et pratiques.

Un problème se pose ensuite quand il s'agit d'exprimer l'efficacité. Une approximation très simple pourrait être le calcul du produit du temps par l'espace, plus exactement, l'intégrale de l'espace dans le temps d'exécution du programme. Même dans ce cas simple, on peut se demander quel espace prendre en considération :

- l'espace affecté au programme, selon une allocation initiale par le système et ainsi rendu inutilisable par d'autres consommateurs de l'espace. (Ici se pose également le problème de la mémoire partagée) ;

- l'espace effectivement alloué au cours de l'exécution par le programme. Il s'agit ici des sous-allocations dans l'espace qui a été affecté initialement au programme par le système ;

- l'espace effectivement nécessaire au cours de l'exécution, c'est-à-dire l'espace contenant des valeurs encore utiles, des données encore "vivantes".

Le problème provient de la nécessité d'une gestion de l'espace soit par le système soit par le programme particulier de l'utilisateur. Cette gestion est souvent organisée sous forme d'allocations et de désallocations, à différents niveaux, s'échelonnant du système au programme et donnant lieu, ainsi, à des sous-allocations en série. L'efficacité de cette gestion pourrait se mesurer par un rapport de l'espace effectivement nécessaire ou effectivement alloué à l'espace affecté au programme.

A supposer qu'une expression d'efficacité soit finalement donnée, il faudrait disposer de méthodes de mesure réalistes de ses paramètres. D'autre part, il faudrait avoir suffisamment d'informations sur le programme, telles des informations statistiques sur les données possibles à l'entrée (ce qui permettrait de déduire la fréquence de différentes séquences d'exécution possibles du programme).

2) - Le terme optimisation peut faire sous-entendre l'existence d'un procédé effectif (algorithme) déterminant un ou plusieurs hypothétiques programmes optimaux, (à supposer que l'on ait défini une fonction d'optimalité). Il n'en est rien. Un tel algorithme général n'existe pas. Il apparaît même que le problème général de trouver un programme optimal est non seulement indécidable mais n'est pas, non plus, semi-décidable. Ce résultat peut être déduit de l'indécidabilité du problème d'arrêt pour les machines de Turing.

On peut cependant concevoir des méthodes heuristiques adaptées à des cas particuliers et fondées sur des hypothèses restrictives adéquates qui permettent d'améliorer - relativement aux critères choisis - un programme donné.

Note : Malgrè ces réserves quant à l'utilisation du terme d'optimisation, nous nous sommes permis de l'utiliser par la suite pour des raisons de commodité.

1.2 - Programme et sa fonction -

Toute la littérature sur l'optimisation de même que cette thèse prennent pour hypothèse que le point de départ est un programme écrit dans un langage de programmation. L'information dont on dispose est donc celle que l'on peut déduire du programme : des informations sur les objets manipulés par le programme et des informations sur l'ordre des calculs à suivre.

On peut se demander d'une part si ces informations se présentent sous une forme adéquate et d'autre part si elles sont suffisantes pour permettre une amélioration maximale du programme. Notons qu'à l'origine du programme il y a un problème à résoudre ou une tâche à exécuter. Le programme (qui est la description d'un algorithme dans un langage donné) n'est, dans cette optique élargie, que le produit d'un processus de recherche d'une solution au problème initial. Ce processus s'effectue soit "manuellement" c'est-à-dire est mené à son terme par un programmeur humain soit automatiquement, par un système adéquat de résolution de problèmes. L'existence d'un programme peut impliquer que le processus ci-dessus a donné un résultat positif, c'est-à-dire a fourni une solution au problème initial, représenté par l'algorithme du programme. Nous supposerons effectivement par la suite que la solution existe et qu'elle est "bonne", c'est-à-dire que le programme est correct.

Il est évident que si l'on considère un programme dans cette optique élargie, les améliorations qui sont fondées sur le programme apparaissent dans certains cas dérisoires par rapport à celles qui se placeraient "en amont" du programme et qui pourraient évoluer dans le cadre, d'une part, des solutions possibles et d'autre part des représentations possibles de la solution choisie. Le choix de la solution, c'est-à-dire de l'algorithme et le choix de sa représentation, c'est-à-dire d'un langage de programmation adéquat peuvent certainement déterminer d'une façon essentielle l'efficacité du programme final. (voir à ce propos [48]).

Nous ne considérerons pas ici ces possibilités qui sont plutôt du domaine de l'intelligence artificielle donc d'un niveau où les méthodes ne sont pas les mêmes que les nôtres.

Revenons maintenant à notre programme et à la question que nous avons posée dans le premier paragraphe de cette partie.

Un programme-source est une description d'un algorithme. On peut généralement considérer, et c'est ce que nous ferons ici, un algorithme comme une définition implicite d'une correspondance (pour la terminologie voir [59]).

Rappelons qu'une correspondance est un triplet (A, B, GR) où A et B sont des ensembles et GR le graphe d'une relation R, soit un sous-ensemble du produit cartésien AxB. La relation R est ici une propriété caractérisant GR, en d'autres termes, R est un prédicat dont l'extension est GR. De plus, cette correspondance sera considérée fonctionnelle par rapport à B c'est-à-dire une application de A dans B ou encore une fonction f définie dans A et à valeurs dans B. Ceci veut dire que :

• A est l'ensemble de définition de la fonction (la fonction est partout définie dans A ou totale, l'algorithme (le programme) se termine)

• pour tout $x \in A$ le y correspondant est unique (la fonction est bien définie ou l'algorithme (le programme) est déterministe)

• l'extension du prédicat R(A, B) s'écrit :

$$GR = \{(x, y) \in A \times B \mid y = f(x)\}$$

La fonction f sera appelée la fonction de l'algorithme ou la fonction du programme. Notons qu'inversement, on pourrait parler de l'algorithme de la fonction c'est-à-dire de l'algorithme calculant les valeurs de la fonction. Si la fonction est calculable au sens de Turing ou encore récursive générale, on sait, par définition, que l'algorithme de la fonction existe.

Par contre étant donné un algorithme prétendant calculer les valeurs d'une fonction dont le domaine de départ et d'arrivée est l'ensemble d'entiers positifs, on sait (cf. [52]) qu'il est indécidable en général - c'est-à-dire quel que soit cet algorithme - de savoir s'il calcule effectivement la fonction considérée.

Par conséquent, un système général de résolution de problèmes ne peut exister. Nous supposerons ici que la fonction du programme est calculable ce qui justifie notre hypothèse avancée plus haut à savoir que le programme particulier est correct ou qu'il peut être prouvé qu'il est correct.

Il est essentiel de remarquer la différence de degré entre les deux aspects d'un programme : le programme en tant qu'une correspondance et en tant qu'une fonction.

La donnée de la correspondance, par la donnée de son graphe GR, définit le programme en tant qu'ensemble, cet ensemble étant en fait la relation R

assimilée à son graphe GR. Cet aspect ensembliste ou relationnel n'implique aucune "direction" pour le calcul de la relation (plus exactement pour le calcul de la fonction caractéristique de l'ensemble GR).

Par contre, lorsqu'on considère le programme en tant que fonction on introduit une notion d'opérateur et, partant de là, de calcul résultant de l'application de cet opérateur aux éléments de l'ensemble de définition. Ce sera l'aspect opérationnel du programme. C'est l'aspect "normal" d'un programme (d'un algorithme).

Evidemment, nous pouvons ramener l'aspect relationnel à l'aspect opérationnel en envisageant les deux familles "d'équations" associées à une relation R définie sur A x B :

1) - R(x, b)
2) - R(a, y)

où b et a, respectivement, sont des éléments fixes de B et de A.

Résoudre une équation R(x, b), c'est chercher la coupe suivant y = b, résoudre une équation R(a, y), c'est chercher la coupe suivant x = a. Avec une fonction de programme, cette symétrie se perd : on calcule seulement les coupes (à un élément) suivant un $a \in A$. (Notons encore que la solution du programme est la valeur de la fonction de programme pour un $a \in A$ donné, alors que la solution de l'équation associée à la fonction f : y = f(x), $y \in B$, est la coupe suivant y). On pourrait dire que l'aspect relationnel se place à un niveau d'appréhension plus élevé.

Nous dirons par la suite qu'une spécification du programme est relationnelle ou opérationnelle selon qu'elle correspond à l'aspect relationnel ou opérationnel du programme. Par extension, parlant du niveau d'une spécification, on pourra le qualifier de relationnel ou d'opérationnel.

A la lecture de certains articles récents [46, 42, 41, 45] il apparaît que ce niveau relationnel est nécessaire pour le traitement, dans le cadre d'un système formel adéquat, des propriétés des programmes non-interprétés (schémas [43, 47]) telles que terminaison ou équivalence. Il semble donc raisonnable de supposer que certaines améliorations - dont la mise en oeuvre est liée aux problèmes d'équivalence - portant sur des classes données de schémas de programmes doivent être abordées à ce niveau. Nous citerons encore

l'article de Manna [44] qui traite de la synthèse automatique des programmes dans laquelle la relation du programme est spécifiée au moyen des prédicats à l'entrée et à la sortie. Le synthétiseur peut produire différents programmes récursifs ou itératifs vérifiant la spécification ce qui donne une possibilité d'amélioration par le choix des programmes.

Pour conclure cette partie, nous dirons que la forme et le niveau (relationnel ou opérationnel) d'une spécification de programme déterminent d'une part, l'éventail des choix des améliorations et d'autre part, la richesse des informations relatives au programme, ce qui conditionne à son tour les améliorations possibles. (Nous pensons ici, par exemple, au fait qu'une formulation prédicative permet de définir explicitement les domaines de définition de divers opérateurs d'une spécification opérationnelle).

Dans le chapitre 2 sera donné un bref aperçu des optimisations traitées dans la littérature. Le chapitre 3 présentera les idées de départ, les sujets concrets traités par la thèse et les hypothèses et les choix initiaux. La thèse proprement dite commencera par le chapitre 4.

2 - OPTIMISATIONS DANS LA LITTERATURE -

Dans ce chapitre nous exposerons brièvement les optimisations des programmes traitées dans la littérature. Une classification de ces optimisations sera tentée et des considérations relatives à leur réalisation seront émises.

Le processus d'optimisation d'un programme consiste en des transformations appliquées au programme, transformations qui entraînent une amélioration des performances du programme.

Les transformations sont effectuées au cours des différentes phases de la traduction du programme-source en programme-objet. A chaque phase de la traduction nous disposons d'une certaine représentation du programme-source ; ces représentations peuvent s'échelonner de l'arbre syntaxique du programme-source au programme en langage-machine en passant éventuellement par plusieurs représentations intermédiaires. Une représentation donnée peut dépendre d'une part de la méthode d'implémentation du langage-source et, d'autre part, de la machine particulière sur laquelle on implémente le langage-source.

On peut ainsi établir - selon la nature de la représentation du programme qu'on optimise - une classification des optimisations en trois catégories :

- optimisations indépendantes de l'implémentation et du langage-machine ;
- optimisations indépendantes du langage-machine mais dépendantes de l'implémentation ;
- optimisations dépendantes du langage-machine (et de l'implémentation).

Une optimisation typique de la troisième catégorie est l'optimisation qui traite l'allocation des registres ; elle doit être entreprise au niveau d'une représentation en langage-machine symbolique, représentation dans laquelle les registres particuliers ne sont pas encore affectés à des données particulières. Des méthodes d'allocation des registres ont été abordées en [11, 35, 21, 15, 9, 28, 29 bis].

Une représentation du langage-source qui dépend de l'implémentation permet d'expliciter l'accès aux données complexes, la gestion de la mémoire et la gestion de l'environnement. On peut alors entreprendre une amélioration de cette gestion et une optimisation des calculs relatifs à l'adressage. De telles optimisations seront abordées dans cette thèse (cf. chapitre 5).

Les optimisations de la première catégorie consistent en des transformations algébriques et structurales.

Les transformations algébriques utilisent les lois algébriques (commutativité, associativité, distributivité, etc...) pour transformer les expressions du programme en accord toutefois avec la sémantique du langage-source. De nombreux articles ont été publiés à ce sujet [11, 32, 35, 31, 20, 10, 38, 8, 23, 30, 1]. Ils traitent de la détection des sous-expressions communes, de la factorisation et des évaluations en parallèle.

Une représentation en arbre semble convenir le mieux aux transformations algébriques (voir [39]).

L'amélioration obtenue par les transformations algébriques peut dépendre de l'allocation des registres et des instructions du langage-machine ce qui fait que ces transformations sont en pratique difficiles à classer et que le choix du niveau de la représentation du programme où elles doivent être appliquées se révèle problématique.

Le développement des boucles ("unrolling" en anglais), la mise en commun des boucles ("merging" en anglais, cf. [34]) et la "réduction de force" des opérateurs [3] ("strength reduction" en anglais) peuvent être considérés comme des transformations structurales. La réduction de force est une transformation spécifique des calculs dans les boucles "contrôlées" par une "variable de contrôle" incrémentée ou décrémentée d'une valeur constante (le "pas") ; ainsi des multiplications par la variable "de contrôle" peuvent être transformées en additions grâce au fait que le pas est constant.

D'autres transformations structurales consistent en le remplacement des appels par les corps des procédures appelées ou bien, inversement, en le remplacement des séquences communes d'instructions par des appels ("procédurage", cf. [23]). La transformation des programmes récursifs en des programmes itératifs [44, 37] est également une transformation structurale (voir à ce propos les travaux théoriques de Strong [49, 50]). L'amélioration est dans ces cas fortement dépendante de l'implémentation des appels de procédures ; on pourrait donc classer les transformations relatives aux procédures dans les optimisations de la seconde catégorie.

Les optimisations structurales doivent être abordées à un niveau de représentation où la structure relative aux boucles et aux procédures du programme-source, en particulier, est explicite. On pourrait utiliser une représentation en arbre du programme-source ; nous proposerons toutefois, dans les chapitres 6 et 7, une représentation intermédiaire qui consiste en des textes intermédiaires dans un langage intermédiaire d'une structure simple, textes intermédiaires associés aux corps des procédures du programme-source. Le langage intermédiaire proposé maintiendra la structure relative aux procédures, aux blocs et aux boucles grâce aux instructions "de structure" spéciales.

Un ensemble important d'optimisations - appelées optimisations topologiques [39] - est lié à l'ordre d'exécution des instructions du programme et à l'ordre d'évaluation des expressions du programme. L'ordre d'exécution des instructions est spécifié par les instructions de contrôle d'exécution (instructions de saut, instructions conditionnelles, boucles, etc...), par un symbole "continuer" (le symbole ";" en PL/1, par exemple) ou bien cet ordre n'est pas spécifié (voir les propositions collatérales en Algol 68, par exemple). On peut remarquer que l'ordre séquentiel qu'implique le symbole "continuer" peut ne pas être un ordre "essentiel".

Exemple : considérons la séquence d'instructions suivante (en PL/1) :

$$\underset{i_1}{A = 3} \; ; \; \underset{i_2}{C = 5} \; ; \; \underset{i_3}{B = A + C} \; ; \; \underset{i_4}{D = C * A} \; ; \; \underset{i_5}{E = B + D} \; ; \; \underset{i_6}{F = B + 1}$$

En supposant que les six variables sont "indépendantes" la séquence ci-dessus est équivalente, en particulier, à la séquence suivante :

$$i_2 \; ; \; i_1 \; ; \; i_4 \; ; \; i_3 \; ; \; i_6 \; ; \; i_5$$

La dépendance qui existe entre les instructions d'une telle séquence ne peut être en général représentée que par un graphe ; le graphe de "dépendance" de l'exemple ci-dessus est donné dans la figure 1.

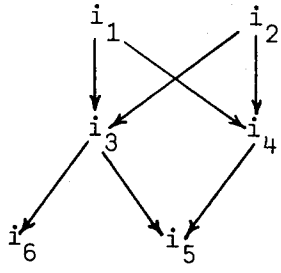


Figure 1.

Notons que l'utilisation des propositions collatérales de Algol 68 ne permet pas de spécifier complètement l'ordre (partiel) de dépendance que représente un graphe de dépendance ; en effet, nous pouvons écrire (en Algol 68) la séquence de l'exemple ci-dessus de deux façons :

par (i_1, i_2) ; par (i_3, i_4) ; par (i_5, i_6) ou bien

par (i_1, i_2) ; par $((i_3 ; i_6), i_4)$; i_5

On peut donc considérer que la sémantique du symbole "continuer", sémantique qui spécifie une exécution séquentielle donnée, est dans une certaine mesure arbitraire.

L'ordre d'évaluation d'une expression est régi par les priorités des opérateurs et par la sémantique du langage qui peut spécifier ou non une évaluation dans un certain ordre en présence des opérateurs de même priorité. On peut encore considérer qu'une sémantique qui spécifie un ordre précis dans le cas ci-dessus est arbitraire dans la mesure où elle ne tient pas compte des lois algébriques (commutativité, associativité, etc...).

L'ordre d'évaluation d'une expression qui est déterminé uniquement par les priorités des opérateurs est appelé ordre (partiel) de précédence.

Exemple : l'ordre de précédence des sous-expressions de l'expression :

A*B+C*D

peut être illustré par le graphe de la figure 2.

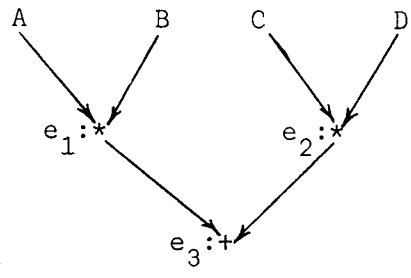


Figure 2.

Lorsqu'on se place à un niveau suffisamment "bas" de représentation du programme les deux ordres de dépendance et de précédence se ramènent à un même ordre que nous appellerons ordre essentiel (le terme est de [23]).

En effet, la séquence-source suivante :

A = A*B+C ; D = A+2 ; C = A*3

peut être représentée dans une représentation intermédiaire par :

i_1 : A*B→t_1

i_2 : t_1+C→t_2

i_3 : t_2→A

i_4 = A+2→t_3

i_5 : t_3*D

i_6 : A*3→t_4

i_7 : t_4→C

Le graphe traduisant l'ordre essentiel des instructions i_1 à i_7 est donné par la figure 3.

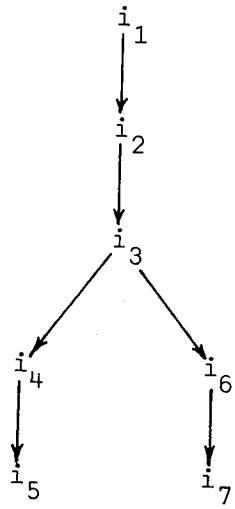


Figure 3.

Une optimisation possible d'une suite d'instructions devant être exécutées séquentiellement consiste alors en la recherche d'un ordre d'exécution des instructions de la suite qui "respecte" (c'est-à-dire qui est impliqué par) l'ordre essentiel ; en pratique, cette optimisation consiste en des déplacements des instructions et est suivie par des éliminations éventuelles des instructions redondantes.

L'amélioration obtenue par des déplacements est liée aux méthodes d'allocation de registres ; on peut par conséquent considérer qu'il s'agit d'une optimisation de la troisième catégorie.

Les optimisations topologiques par des déplacements et par des éliminations des instructions peuvent être étendues à tout le programme ; on parle alors de l'optimisation (topologique) globale. Les techniques de l'optimisation globale ont été conçues par Cocke, Schwartz et Allen [3, 5, 4, 12, 14, 13, 6, 7] et ont été développées ensuite par d'autres auteurs [29, 16, 33, 26] et ont été récemment appliquées également à l'allocation des registres [9].

Ces techniques seront aussi abordées dans cette thèse (chapitre 8). Les optimisations topologiques doivent être accomplies à un niveau de représentation qui permet de rendre explicites toutes les évaluations susceptibles d'être optimisées. Ces optimisations peuvent donc être classées dans les deux dernières catégories.

La première implémentation des méthodes globales d'optimisation semble avoir été réalisée dans le cadre d'un compilateur pour Fortran [32]. Les problèmes que pose l'inclusion des optimisations dans les compilateurs ont été discutés en [22, 27, 18].

3 - PRESENTATION DE LA THESE - HYPOTHESES ET CHOIX INITIAUX -

Dans ce chapitre nous exposerons :

- les idées sur lesquelles reposent la conception et la présentation de la thèse (en 3.1) ;
- les sujets concrets abordés par la thèse (en 3.2) ;
- les hypothèses et les choix initiaux de la thèse (en 3.3).

3.1 - Idées de départ -

La présente thèse - nous l'avons déjà dit dans le premier chapitre - traite de l'amélioration des programmes écrits dans des langages procéduraux de haut niveau.

Les problèmes que suscite ce sujet concernent :

- la définition du terme amélioration ; ce problème a été effleuré dans le premier chapitre ;
- les méthodes de transformation des programmes, transformations qui visent l'amélioration des programmes ;
- la réalisation pratique des procédés d'amélioration dans le cadre d'un compilateur du langage-source, langage dans lequel le programme à améliorer est rédigé.

Les méthodes d'amélioration existantes ont été brièvement rappelées dans le chapitre 2 ainsi que les approches possibles à leur insertion dans un compilateur.

On peut constater que les trois problèmes énoncés ci-dessus sont étroitement liés à ce que nous pensons être le problème central, à savoir le problème des informations. Ce problème sera le sujet principal de notre thèse.

Le problème des informations est triple : il concerne :

- la disponibilité des informations ;
- l'acquisition des informations ; et
- la manipulation des informations disponibles ou acquises.

Il peut s'agir ici de deux genres d'informations :

- des informations permettant de déterminer une fonction d'optimalité des programmes (cf. chapitre 1) et
- des informations nécessaires à la réalisation des optimisations.

Nous nous occuperons par la suite du deuxième genre d'informations.

Revenons donc au triple problème d'informations : la disponibilité des informations dépend du langage-source dans lequel est rédigé le programme à optimiser. Tout langage de programmation fournit, bien évidemment, de par sa sémantique, les informations qui sont nécessaires à son implémentation. Bon nombre d'informations qui seraient utiles à l'optimiseur (mais qui, du point de vue de l'implémentation, ne sont pas indispensables) ne sont cependant pas fournies par les langages de programmation courants ; il peut s'agir, par exemple, des informations suivantes :

- telle boucle est-elle toujours exécutée au moins une fois ? ;
- telle donnée à l'entrée, d'un type déclaré donné, peut-elle avoir toutes les valeurs que détermine son type ? ;
- telle occurrence d'opération peut-elle provoquer une interruption ? ;
- telle occurrence d'un argument passé par référence peut-elle être modifiée par l'appel où elle apparaît ? ;
- telle occurrence de saut référençant une étiquette variable (en PL/1) peut-elle provoquer le transfert à une étiquette constante donnée ? etc....

Quelle peut-être l'utilité des informations citées ci-dessus ? La première information peut permettre de transférer le test de la boucle à la fin de celle-ci, ce qui peut accroître les possibilités des optimisations relatives aux boucles (cf. chapitre 8). La deuxième information peut permettre de simplifier les tests. La troisième information permet de ne pas tenir compte des contraintes de sécurité relatives aux déplacements des instructions (cf. chapitre 8). La quatrième information peut permettre une simplification du passage des arguments (cf. chapitre 5.3). La cinquième information permet de simplifier dans certains cas la gestion de l'environnement (cf. chapitre 5).

L'optimiseur, bien évidemment, ne peut utiliser que les informations fournies par le programme ou les informations qu'il est possible de déduire du programme ; l'acquisition de ces informations - que nous appellerons informations déduites (cf. chapitre 4) - pourra s'effectuer au niveau de l'analyse syntaxi-

que du programme-source (cf. chapitre 4) et au niveau du programme intermédiaire (cf. chapitre 8).

Les informations fournies par le programme concernent les données d'une part et la structure du programme - structure relative au contrôle de l'exécution - d'autre part. Les informations déduites sont les informations qui ne sont pas données d'une façon immédiate par la sémantique du langage-source. Nous citerons quelques-unes de ces informations déduites :

- utilisation globale des variables ;
- modifications des variables par des appels ;
- l'ordre d'activation des procédures, etc....

La facilité de l'acquisition des informations déduites et l'exactitude de ces informations dépendent essentiellement du langage-source (de la "bonne structuration" de ce dernier, au sens de Dijkstra [55]) et des restrictions que l'on peut éventuellement imposer à ce dernier. Il existe donc (cf. ch. 1) une dépendance étroite entre l'optimisation des programmes et la conception des langages de programmation. Cette thèse essaiera d'examiner quelques aspects de cette dépendance.

L'acquisition des informations s'effectuera - nous l'avons dit plus haut - à des niveaux différents de représentation du programme ; cette thèse mettra en évidence deux niveaux possibles : celui du langage-source et celui d'un langage intermédiaire. Ces niveaux peuvent correspondre d'une part aux phases du compilateur et d'autre part - voir le chapitre 2 - aux classes d'optimisations.

Le problème de manipulation des informations - immédiates ou déduites - sera compris ici comme étant un problème de transfert des informations du niveau du langage-source au niveau du langage intermédiaire. Ce problème de transfert, nous semble-t-il, n'a pas été suffisamment mis en évidence dans la littérature sur l'optimisation. Le problème de transfert est bien sûr inexistant lorsque existe, pendant la compilation, un seul niveau de représentation du programme, celui du langage-source ou lorsque les informations déduites sont acquises indépendamment à chacun des niveaux successifs de représentation du programme-source. La première hypothèse - c'est-à-dire un seul niveau de représentation - n'est pas réaliste : tout compilateur tant soit peu complexe met en évidence plusieurs niveaux intermédiaires de représentation (pour ne pas parler du niveau

du langage-objet). La deuxième hypothèse entraîne au moins une duplication du travail de l'optimiseur : elle suppose, en effet, qu'on "oublie", à un niveau de représentation donné, les informations acquises au niveau précédent.

Deux "transferts" d'informations seront abordés dans cette thèse : le transfert des informations relatives à la modification des variables-source vers le niveau du langage intermédiaire et le transfert des informations sur la structure (structure des blocs et des boucles) du programme-source vers le niveau du langage intermédiaire.

3.2 - Sujets concrets abordés -

Le but de cette thèse est l'élucidation du problème triple des informations exposé en 3.1. La confection d'un optimiseur pour un langage donné, en particulier, n'est pas notre but.

Les sujets traités seront, d'une part :

- l'acquisition des informations déduites au niveau du langage-source (cf. ch. 4) et au niveau du langage intermédiaire (cf. 8.1.1 et 8.2.2) et, d'autre part
- l'utilisation des informations déduites dans les optimisations suivantes :
  - . l'optimisation de l'implémentation d'un programme-source donné (cf. ch. 5) ;
  - . les optimisations consistant en des déplacements et des éliminations des instructions du programme-intermédiaire (cf. ch. 8).

Les optimisations seront indépendantes d'un langage-machine particulier.

Le chapitre 4 décrira l'acquisition - au niveau du langage-source - des informations déduites suivantes :

- le graphe potentiel d'appels entre les procédures. Un algorithme permettant de l'obtenir sera donné ;
- les informations relatives aux modifications indirectes des arguments d'un appel et aux modifications implicites dues au référençage global des variables.

Le chapitre 5 décrira une implémentation améliorée d'un sous-ensemble de PL/1. L'amélioration portera sur :

- la gestion de l'environnement ;
- le passage des arguments ;
- l'allocation de la mémoire, allocation effectuée au moyen des méthodes de recouvrement.

Le chapitre 6 décrira un langage intermédiaire et le chapitre 7 mettra en évidence la structure de ce langage - structure matérialisée par des graphes de programmes et fondée sur la structure du programme-source.

Le chapitre 8 proposera une nouvelle approche aux optimisations connues de déplacement et d'élimination. La nouveauté de l'approche réside dans le fait qu'elle est fondée sur la structure du programme-source.

Le chapitre 9 tentera de dégager les conséquences qu'impliquera notre examen du problème des informations et tâchera d'entrevoir les possibilités d'un développement futur des optimisations.

3.3 - Hypothèses et choix initiaux -

Le langage de programmation traité par la thèse est un sous-ensemble de PL/1. Ce choix a été essentiellement arbitraire ; toutefois nous pourrions dire que le choix de PL/1 se révèle heureux dans le sens suivant : les irrégularités de structure de PL/1 nous permettrons précisément de souligner l'importance - pour la facilité de l'acquisition des informations déduites et pour l'exactitude de ces informations - d'une "bonne structuration" d'un langage de programmation.

Des restrictions syntaxiques et sémantiques ont été introduites - définissant ainsi un sous-langage de PL/1 appelée SPL/1 par la suite - pour les raisons suivantes :

- pour simplifier l'acquisition des informations déduites,
- pour écarter les problèmes qui ne seront pas abordés dans cette thèse et
- pour faciliter la définition d'un langage intermédiaire.

Les restrictions portent sur :

- les types et la structure des données ;
- la sémantique des instructions de transfert de contrôle (GOTO, STOP) et sur
- l'éventail des instructions et des options admises en SPL/1.

Nous énumérerons par la suite ces restrictions en essayant de les motiver. On se réfèrera à la brochure sur le PL/1 citée en [58].

Un programme-source en SPL/1 est composé d'un ensemble de procédures, la plus "externe" ayant l'attribut MAIN. On n'admet donc pas de compilation séparée.

L'option TASK n'est pas admise (c'est-à-dire point de tâches multiples).

Les restrictions relatives aux données portent sur :

- les attributs de portée ;
- les "types" ;
- les attributs de durée de vie ;
- les attributs d'alignement, DEFINED et INITIAL ;
- les conversions.

Ces restrictions sont motivées par le fait que nous ne traiterons pas le problème des dépendances entre les données-source, dépendances relatives à la structuration des données, au recouvrement mutuel des données (cf. l'attribut DEFINED) et à l'adressage (cf. les attributs POINTER, AREA, BASED, CTL). Ces problèmes de dépendance ont été abordés en [33, 2, 36].

Le seul attribut de portée est INTERNAL.

Les attributs de durée de vie sont STATIC et AUTOMATIC.

Les attributs relatifs aux "types" sont explicites :

- FLOAT DEC(6),
- BIN FIXED(31), et BIN FIXED(15) pour les bornes,
- dimension,
- LABEL(étiq.-constante [, étiq.-constante] ...)

ou implicites :

- les constantes entières sont FIXED BIN(31),
- les bornes constantes sont BIN FIXED(15),
- les autres constantes sont DEC FLOAT(6).

Les données sont alignées (l'attribut ALIGNED).

L'attribut de recouvrement DEFINED spécifie uniquement le recouvrement "direct par correspondance" (cf. [58]). Les étiquettes et les paramètres ne peuvent être déclarés avec l'attribut DEFINED.

L'option CALL n'est pas possible pour l'attribut INITIAL.

Les seules données composées sont les tableaux.

Les restrictions relatives aux expressions logiques et de comparaison se déduisent de l'absence du type "booléen". Les seules conversions sont celles entre les valeurs entières et réelles.

Les restrictions relatives aux étiquettes et aux sauts simplifient l'acquisition des informations déduites relatives au contrôle de l'exécution :

- toute déclaration d'une étiquette variable EV (l'attribut LABEL, voir ci-dessus) doit spécifier la liste des étiquettes constantes qui sont les valeurs possibles de EV ;
- toute étiquette variable est une donnée simple (c'est-à-dire pas de tableaux d'étiquettes) ;
- une étiquette ne peut être passée en argument, ne peut être le résultat délivré par une procédure-fonction et ne peut être déclarée avec l'attribut DEFINED ;
- tout saut (l'instruction GOTO) doit être interne à l'activation courante de la procédure qui contient l'occurrence de l'instruction GOTO en question. Cette restriction sémantique exigera* un test à l'exécution (cf. 5.3.7).

Les restrictions relatives aux procédures sont les suivantes :

- toute procédure, sauf celle ayant l'attribut MAIN, est "interne" ;
- la procédure ayant l'attribut MAIN ne peut avoir d'options RECURSIVE et RETURNS ;
- toute procédure possède un seul point d'entrée (un seul nom d'entrée (principale)) ;

* C'est mon choix ; on aurait pu changer la sémantique de PL/1 de façon à supprimer la nécessité des tests à l'exécution.

- les options et les attributs admis sont : RECURSIVE, RETURNS (attribut) et ENTRY [liste d'att. [, liste d'att.]...].

Comme la procédure MAIN est la seule procédure externe, l'attribut ENTRY sans la liste d'attributs n'est spécifiable que pour des paramètres ;

- seules les instructions END, RETURN et STOP permettent de quitter directement une procédure-sous programme ;
- la dernière instruction exécutée par une procédure-fonction est une occurrence de l'instruction RETURN (expression).

On ne peut quitter - directement ou indirectement - une procédure-fonction par une instruction STOP. Cette restriction exigera* un test à l'exécution ;

- l'ordre d'évaluation des arguments d'un appel est indéfini. Cette "restriction" sémantique sera exploitée en 5.4.2.2 ;
- toute procédure susceptible d'être appelée récursivement doit** être "déclarée" avec l'option RECURSIVE. La propriété que représente l'option RECURSIVE ne peut être déduite par une analyse statique simple. Nous illustrerons ce fait par l'exemple du programme donné par la figure 1 (cet exemple est pris en [33]).

* Des restrictions plus sévères auraient pu être faites, de façon à supprimer la nécessité des tests à l'exécution.

** Ce choix - qui peut paraître arbitraire - impliquera un test à l'exécution (voir plus loin).

```
PG : PROCEDURE OPTIONS (MAIN) ;
     DCL BOOL ; ...
     CALL A ; ...
     A : PROCEDURE ;
          IF BOOL THEN CALL B ; ELSE CALL C ; ...
          B : PROCEDURE ; ...
          IF BOOL THEN CALL C ; ...
          END B ; ...
          C : PROCEDURE ; ...
          IF ¬BOOL THEN CALL B ; ...
          END C ; ...
     END A ; ...
END PG ;
```

Figure 1.

Le graphe des appels (cf. 4.2.1) que l'on peut construire à partir de l'exemple de la figure 1 est donné dans la figure 2.

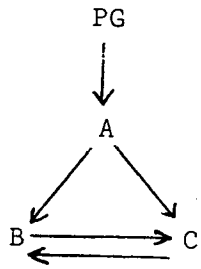


Figure 2.

Le graphe de la figure 2 suggère qu'un circuit [B, C] "d'activation" des procédures est possible. Or il n'en est rien ; l'examen du programme de la figure 1 nous montre que seuls les chemins "d'activation" :

A→B→C et
A→C→B

sont possibles (en supposant que la variable BOOL n'est pas modifiée par les procédures A, B et C). Il s'ensuit que l'option RECURSIVE est nécessaire : dans l'exemple de la figure 1 les deux procédures B et C n'ont pas d'option RECURSIVE ; on en déduit par conséquent que le circuit [B, C] est impos-

sible et que le graphe des appels ne représente qu'une information déduite imparfaite (voir à ce propos 4.2.1 et 5.3.4). Un test à l'exécution devra être prévu pour vérifier l'admissibilité d'une activation récursive d'une procédure.

Nous donnerons par la suite une liste des attributs, des options et des instructions du sous-langage SPL/1.

Les attributs : AUTO, BIN, DEC, DEFINED, dimension, ENTRY, FIXED, FLOAT, INITIAL, INTERNAL, LABEL, ALGINED, précision, RETURNS, STATIC.

Les instructions : DCL, affectation, BEGIN (sans option), DO (une instruction DO répétitive sera appelée une boucle DO), END, GET LIST, PUT LIST, GOTO, IF, THEN, ELSE, PROCEDURE, CALL (sans option), RETURN, STOP et instruction vide.

Les options : MAIN, RECURSIVE.

Nous ferons en outre les hypothèses générales suivantes :

- tout programme en SPL/1 est correct et se termine (cf. ch. 1) ;
- toute instruction du programme est "atteinte" (c'est-à-dire pour toute instruction i du programme il existe une activation du programme qui exécute i).

L'acceptation de ces hypothèses générales est motivée par la disponibilité limitée des informations (voir 3.1) que peut fournir le langage PL/1.

Note 1 : le reste de la thèse comprend les chapitre 4 à 9, deux annexes et la bibliographie.

Note 2 : nous utiliserons pour les graphes la terminologie de Berge [51].

4 - INFORMATIONS DEDUITES AU NIVEAU DE L'ANALYSE D'UN PROGRAMME-SOURCE EN SPL/1 -

Ce chapitre est composé de deux parties : la première introduira la terminologie relative à la structure des programmes-source en SPL/1 ; la deuxième décrira les informations qui peuvent être déduites au niveau de l'analyse d'un programme-source en SPL/1.

4.1 - Terminologie relative à la structure des programmes-source en SPL/1 -

Nous présenterons ici la terminologie relative à la structure d'un programme-source ; cette terminologie sera utilisée en 4.2 ainsi que dans les chapitres 5 et 6. Certes, on pourra faire quelques objections sur le choix des termes ; quoiqu'il en soit l'introduction de ces termes nous semble indispensable pour une description précise et concise. Nous nous sommes inspirés parfois de la terminologie française relative au langage Algol 68.

Un programme-source en SPL/1 est constitué d'un ensemble de procédures (procédures-sous-programmes ou, brièvement, sous-programmes et procédures-fonctions ou, brièvement, fonctions) dont une a l'attribut MAIN.

On appellera région le texte d'un bloc PROCEDURE (région de procédure) ou d'un bloc BEGIN (région de bloc-début) ; ce texte inclut les textes de tous les blocs PROCEDURE et BEGIN textuellement internes.

On appellera corps d'une procédure (ou corps de procédure) le texte d'une procédure, à l'exclusion des procédures qui sont textuellement internes à la procédure donnée (cette notion de corps de procédure est analogue à la notion de fief en Algol 68, au niveau des procédures).

On appellera arbre d'inclusion statique des corps de procédures (notation abrégée : ASC) d'un programme l'arbre qui traduit l'inclusion textuelle des corps de procédures d'un programme-source. La racine de cet arbre est le corps de la procédure à l'attribut MAIN.

Relativement à l'arbre ASC on associe à tout corps de procédure son niveau procédural qui est la distance dans ASC de la racine (cette dernière ayant le niveau 0).

On appellera graphe potentiel d'appel (GPA) le graphe qui représente les appels directs (c'est-à-dire "non-transitifs") possibles entre les procédures d'un programme-source : les sommets du graphe GPA sont les corps des procédures du programme et un arc de cp_1 à cp_2 indique que cp_1 contient au moins une occurrence d'appel de procédure qui peut donner lieu à un appel à la procédure dont le corps de procédure est cp_2 (notons que le SPL/1 admet des paramètres à l'attribut ENTRY). Le graphe GPA indique seulement les activations directes des procédures du programme-source et n'indique pas les retours des procédures. L'obtention de GPA est exposée en 4.2 et une utilisation de GPA est décrite en 5.4.1.

On appellera fief-procédure d'un corps de procédure cp_i le texte du bloc PROCEDURE associé au cp_i dont sont exclus les textes des blocs BEGIN et des boucles DO textuellement internes.

On appellera fief-début d'un corps de procédure cp_i le texte d'un bloc BEGIN (faisant partie du texte cp_i) dont sont exclus les textes des blocs BEGIN et des boucles DO textuellement internes.

On appellera fief de portée un fief-procédure ou un fief-début.

L'inclusion statique des fiefs de portée d'un corps de procédure cp_i définit l'arbre d'inclusion statique des fiefs de portée (noté $ASFP_i$) du cp_i. Relativement à l'arbre $ASFP_i$ on associe à tout fief de portée f_j de $ASFP_i$ son niveau de fief (noté NF_j) dont la définition est analogue à celle du niveau procédural. Notons que tout fief-procédure est de niveau 0.

On appellera corps de boucle d'un corps de procédure cp_i le texte d'une boucle DO ; ce texte inclut les textes des blocs BEGIN et des boucles DO textuellement internes à la boucle DO en question.

On appellera fief de boucle d'un corps de procédure cp_i le texte d'une boucle DO dont sont exclus les textes des blocs BEGIN et des boucles DO textuellement internes. Une définition plus exacte du terme fief de boucle sera donnée en 6.2.7.

On appellera fief un fief de portée ou un fief de boucle.

L'inclusion statique des fiefs d'un corps de procédure cp_i définit l'arbre d'inclusion statique des fiefs (noté ASF_i) du cp_i. On parlera également des niveaux dans un ASF.

Un exemple est donné par les figures 1 et 2 : la figure 1 représente le schéma d'un corps de procédure et la figure 2 l'arbre ASF associé.

```
P1 : PROCEDURE ... ;
     .
     .
     B1 : BEGIN ;
          .
          .
          L1 : DO ... ;
               .
               .
               L2 : DO ... ;
                    .
                    .
                    END ;
               .
               .
               L3 : DO ... ;
                    .
                    .
                    END ;
               .
               .
               END L1 ;
          .
          .
          END B1 ;
     .
     .
     L4 : DO ... ;
          .
          .
          B2 : BEGIN ;
               .
               .
               B3 : BEGIN ;
                    .
                    .
                    END ;
               .
               .
               B4 : BEGIN ;
                    .
                    .
                    END ;
               .
               .
               END B2 ;
          .
          .
          END L4 ;
     .
     .
END P1 ;
```

Figure 1.

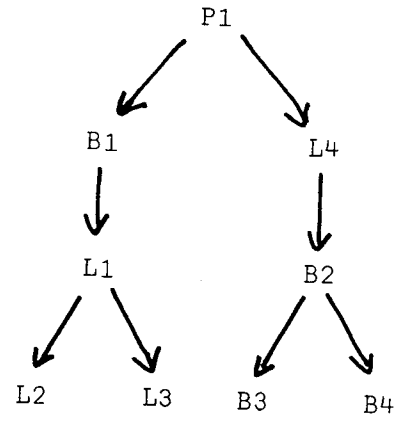


Figure 2. : L'arbre ASF de P1.

D'autres termes, servant uniquement à la description de l'implémentation du SPL/1 seront introduits dans le chapitre 5.

4.2 - Informations déduites au niveau de l'analyse d'un programme-source -

Nous appellerons information déduite toute information qui a trait à la sémantique statique [56] d'un programme-source. Il est donc, par définition, possible d'obtenir toutes les informations déduites au niveau de l'analyse du programme-source ou encore, à un niveau où on ne dispose pas explicitement d'un graphe du programme (par graphe de programme on peut entendre ici l'organigramme ou flowchart, en anglais).

Les informations déduites seront exploitées au niveau de l'implémentation d'un programme-source donné et au niveau des optimisations effectuées sur le texte intermédiaire obtenu à partir d'un programme-source donné.

Nous présenterons par la suite les informations déduites dont l'obtention n'est pas immédiate ou triviale. D'autres informations déduites seront exposées dans le chapitre 5 qui présentera l'implémentation optimisée du SPL/1.

Les informations déduites seront encodées sous forme des graphes, des attributs, des listes, des ensembles ou des valeurs entières.

Par la suite nous exposerons en 4.2.1 l'obtention du graphe potentiel d'appels et des ensembles AP_k (ces derniers seront utilisées en 5.3.3), la détermination de l'attribut MOD relatif à une occurrence d'argument passé par référence (en 4.2.2.1) et la détermination des autres informations relatives aux modifications des variables-source d'un programme-source donné (en 4.2.2.2).

Une conclusion sera présentée en 4.2.2.3.

4.2.1 - Obtention du graphe GPA et des ensembles AP_k -

Nous exposerons ici l'algorithme A1* qui permet de déduire le graphe potentiel des appels des procédures (GPA) d'un programme donné et, pour toute occurrence d'appel formel (le terme est expliqué par la suite), l'ensemble des noms des

* La complexité - relativement à l'algorithme classique - de l'algorithme A1 présenté résulte de la nature du langage considéré (un sous-ensemble de PL/1).

procédures pouvant être appelées par cette occurrence d'appel formel (les ensembles AP_k, cf. 5.3.3).

Nous introduirons d'abord les termes et les définitions utilisés dans la description et dans la justification de l'algorithme.

Chacune des procédures du programme-source est identifiée par son nom de procédure (on supposera ce nom unique) qui est l'étiquette de l'instruction PROCEDURE correspondante. Les noms des procédures dénoteront également les corps des procédures associés.

Soit $CP = \{cp_i\}_{i = 1, \ldots, II}$ l'ensemble des noms des procédures du programme-source donné.

On appelle paramètre ENTRY un paramètre dont le nom (qu'on supposera unique) figure dans une liste des paramètres d'une procédure du programme-source considéré et qui est déclaré avec l'attribut ENTRY.

Soit $PARAM = \{par_j\}_{j = 1, \ldots, JJ}$ l'ensemble des noms des paramètres ENTRY de toutes les procédures du programme-source considéré. On a $CP \cap PARAM = \emptyset$.

On appelle appel non-formel une occurrence d'appel d'une procédure du programme-source dans laquelle la procédure appelée est identifiée par son nom de procédure ; on appelle appel formel une occurrence d'appel d'une procédure du programme-source dans laquelle la procédure appelée est désignée par un paramètre ENTRY.

Toutes les occurrences des appels formels d'un programme-source donné seront numérotées de 1 à KK où KK est le nombre total des occurrences des appels formels du programme-source donné.

Nous supposerons que le programme-source considéré est composé d'une seule procédure externe et que, pour toute procédure, on connaît le nombre et la position - dans la liste des paramètres correspondante - de ses paramètres ENTRY. On supposera, en outre, que l'on ait su identifier, quelque soit le contexte, les noms des procédures et les paramètres ENTRY apparaissant en tant qu'arguments dans les listes des arguments du programme-source.

Un paramètre ENTRY par_j qui est le $k^{\text{ième}}$ paramètre d'une procédure cp_i ne peut recevoir de valeur que par un appel de procédure de la forme suivante :

$cp_i(..., cp_{i1}, ...)$ où cp_{i1} est le $k^{ième}$ argument de la liste ;

ou $cp_i(..., par_{j1}, ...)$ où par_{j1} est le $k^{ième}$ argument de la liste ;

ou $par_{j2}(..., \begin{Bmatrix} cp_{i1} \\ par_{j1} \end{Bmatrix}, ...)$ où cp_{i1} ou par_{ji} est le $k^{ième}$ argument de la liste et où cp_i est une valeur possible de par_{j2}.

Nous dirons que cp_{i1} (ou par_{j1}) est passé à par_j. Les passages où sont passés des noms de procédures seront enregistrés dans un tableau MP2 de dimension IIxJJ et les passages où sont passés des paramètres ENTRY seront enregistrés dans un tableau MP1 de dimension JJxJJ.

L'algorithme construira deux tableaux :

- le tableau MA1 de dimension IIxII indiquant, par un élément (i_1, i_2) de valeur booléenne vrai, que le corps de procédure cp_{i1} peut appeler directement (c'est-à-dire par un appel effectué dans le corps cp_{i1}) la procédure dénotée par cp_{i2} ; MA1 représente donc la matrice associée au graphe GPA cherché.

- le tableau MA2 de dimension KKxII indiquant, par un élément (k, i) de valeur booléenne vrai, que l'appel formel du numéro k peut être un appel à la procédure dénotée par cp_i.

L'algorithme examine d'abord les appels non-formels du programme-source et itère ensuite sur les appels formels en mettant à jour les tableaux MA1, MA2, MP1 et MP2. Pour chaque appel traité l'algorithme compare le nombre d'arguments au nombre de paramètres de la procédure appelée, le nombre d'arguments ENTRY (un argument ENTRY est soit un paramètre ENTRY soit un nom de procédure) au nombre de paramètres ENTRY et les positions des arguments ENTRY de la liste des arguments de l'appel aux positions des paramètres ENTRY de la liste des paramètres de la procédure appelée. Dans le cas d'un appel formel cette comparaison permet d'éliminer certaines valeurs (qui sont des noms des procédures) à priori possibles du paramètre ENTRY "appelé" par l'appel formel.

L'algorithme examine un appel non-formel une seule fois et le nombre d'examens d'un appel formel est égal au nombre de valeurs que peut prendre le paramètre ENTRY "appelé" par l'appel formel. Le nombre total d'examens est ainsi inférieur ou égal à LL (le nombre d'appels non-formels) + KK*JJ ; en conséquence l'algorithme se termine.

Les données à l'entrée sont encodées de la façon suivante :

- pour toute procédure $cp_i \in CP$ on dispose du nombre de ses paramètres (NP) et du nombre de ses paramètres ENTRY (NE1 ; avec NE1 ≤ NP) ; lorsque NE1 ≠ 0 on dispose aussi d'un vecteur booléen EP de longueur NP indiquant les positions des paramètres ENTRY et d'un vecteur entier PX de longueur NE1 indiquant les indices des paramètres ENTRY de la liste des paramètres de cp_i ;

- pour tout appel non-formel on dispose de l'indice i ($1 \leq i \leq II$) du nom de la procédure appelée, de l'indice i_1 ($1 \leq i_1 \leq II$) du corps de la procédure contenant l'appel, du nombre d'arguments de la liste des arguments de l'appel (NA) et du nombre d'arguments ENTRY de la liste des arguments de l'appel (NE). Lorsque NE ≠ 0 on dispose aussi d'un vecteur booléen EA de longueur NA indiquant les positions des arguments ENTRY, d'un vecteur booléen TA de longueur NE indiquant soit que l'argument ENTRY du numéro ℓ($1 \leq \ell \leq NE$) est un nom de procédure (TA(ℓ) = faux) soit que l'argument ENTRY du numéro ℓ est un paramètre ENTRY (TA(ℓ) = vrai) et d'un vecteur entier EX de longueur NE indiquant les indices des arguments ENTRY.

- pour tout appel formel on dispose de l'indice j($1 \leq j \leq JJ$) du paramètre ENTRY "appelé" (PFA), de l'indice i($1 \leq i \leq II$) du corps de la procédure contenant l'appel (CA) et des informations NA, NE, EA, TA et EX qui sont les mêmes que celles relatives à un appel non-formel.

L'algorithme a été implémenté en Algol 60 et a été testé sur plusieurs exemples. Il utilise les procédures INPUT et OUTPUT du rapport sur l'Algol 60 [60] pour effectuer les entrées et les sorties. Pour abréger le texte de l'algorithme présenté ci-dessous nous utiliserons deux procédures hypothétiques IN et OUT permettant de lire et d'imprimer aussi bien des tableaux que des variables simples. La procédure WARSHALL de l'algorithme est celle de la référence [54].

L'algorithme A1 :

```
début entier II, JJ, KK, LL, MAX1, MAX2, MAX3, MAX4 ;
     comment LL : le nombre d'appels non-formels
           MAX1 : le nombre maximal d'arguments d'une liste des arguments
           MAX2 : le nombre maximal de paramètres d'une liste des paramètres
           MAX3 : le nombre maximal d'arguments ENTRY d'une liste des arguments
           MAX4 : le nombre maximal de paramètres ENTRY d'une liste des para-
                  mètres
           MP2B : tableau indiquant les éléments de MP2 déjà traités
           NAPF : NAPF(j), 1 ≤ j ≤ JJ, est le nombre d'appels formels au
                  paramètre ENTRY d'indice j. Les KK appels formels sont
                  ordonnés dans l'ordre croissant des indices j, 1 ≤ j ≤ II,
                  des paramètres ENTRY désignant les procédures formellement
                  appelées ;
LECT : IN (II, JJ, KK, LL, MAX1, MAX2, MAX3, MAX4) ;
début tableau booléen  MA1[1 : II, 1 : II], MP1[1 : JJ, 1 : JJ],
                       MA2[1 : KK, 1 : II], MP2[1 : II, 1 : JJ],
                       MP2B[1 : II, 1 : JJ],
     EA[1 : KK, 1 : MAX1], TA[1 : KK, 1 : MAX3], EP[1 : II, 1 : MAX2] ;
     tableau entier NAPF[1 : JJ], OFAPF[1 : JJ], PFA[1 : KK],
     CA[1 : KK], NA[1 : KK], NE[1 : KK], NP[1 : II], NE1[1 : II],
     EX[1 : KK, 1 : MAX3], PX[1 : II, 1 : MAX4] ;
     entier I, J, K, ITERCPT ;
     booléen TMP1, TMP2, TEST, TMA1, TMA2 ;
procédure WARSHALL ;
     début tableau booléen B[1 : JJ] ;
     pour I := 1 pas 1 jusquà JJ faire
          début pour J := 1 pas 1 jusquà JJ faire B[J] := MP1[I, J] ;
                pour J := 1 pas 1 jusquà JJ faire
                si MP1[J, I] alors
                pour K := 1 pas 1 jusquà JJ faire
                MP1[J, K] := MP1[J, K] ou B[K]
          fin
fin WARSHALL ;
```

```
procédure MP1MP2 ; comment changer MP2 en fonction de MP1 ;
     début pour I := 1 pas 1 jusquà JJ faire
           pour J := 1 pas 1 jusquà JJ faire
           si MP1[J, I] alors
           pour K := 1 pas 1 jusquà II faire
          début TMP2 := TMP2 ou ¬(MP2[K, I] equiv MP2[K, J]) ;
          MP2[K, I] := MP2[K, I] ou MP2[K, J]
          fin
     fin MP1MP2 ;
procédure SORTIR ;
     début OUT (ITERCPT) ; OUT (MA1, MP1, MP2, MA2)
     fin SORTIR ;
procédure LIRE ;
     début IN(PFA[K], CA[K], NA[K], NE[K]) ;
          si NE[K] ≠ 0 alors
          début pour I := 1 pas 1 jusquà NA[K] faire IN(EA[K, I]) ;
                pour I := 1 pas 1 jusquà NE[K] faire IN(TA[K, I]) ;
                pour I := 1 pas 1 jusquà NE[K] faire IN(EX[K, I])
          fin
     fin LIRE ;
procédure SET ;
     début comment si l'appel au paramètre k peut convenir :
     MA1 et MA2 changent et MP1 et/ou MP2 peuvent changer ;
     entier L ;
     si NA[K] ≠ NP[I] ou NE[K] ≠ NE1[I] alors allera EXIT1 ;
     si NE[K] ≠ 0 alors
          pour L := 1 pas 1 jusquà NA[K] faire
          si ¬(EA[K, L] equiv EP[I, L]) alors allera EXIT1 ;
     TMA1 := TMA1 ou ¬MA1[CA[K], I] ;
     MA1[CA[K], I] := vrai ;
     si TEST alors
     début TMA2 := TMA2 ou ¬MA2[K, I] ;
           MA2[K, I] := vrai
     fin ;
     si NE[K] = 0 alors allera EXIT1 ;
     pour L := 1 pas 1 jusquà NE[K] faire
          début entier IND, IND1 ;
          IND := PX[I, L] ; IND1 := EX[K, L] ;
          si TA[K, L] alors début TMP1 := TMP1 ou ¬MP1[IND1, IND] ;
                                  MP1[IND1, IND] := vrai fin
```

```
            sinon début TMP2 := TMP2 ou ¬MP2 [IND1, IND] ;
                        MP2[IND1, IND] := vrai fin
            fin boucle L ; EXIT1 :
        fin SET ;
comment initialiser à faux MA1, MP1, MP2, MP2B ;
pour I := 1 pas 1 jusquà II faire pour J := 1 pas 1 jusquà II faire
MA1[I, J] := faux ;
pour I := 1 pas 1 jusquà JJ faire pour J := 1 pas 1 jusquà JJ faire
MP1[I, J] := faux ;
pour I := 1 pas 1 jusquà II faire pour J := 1 pas 1 jusquà JJ faire
        début MP2[I, J] := faux ; MP2B[I, J]:= faux fin ;
pour I := 1 pas 1 jusquà KK faire pour J := 1 pas 1 jusquà II faire
MA2 [I, J] := faux ;
comment lire les données relatives aux procédures ;
pour I := 1 pas 1 jusquà II faire
        début IN(NP[I], NE1[I]) ;
            si NE1[I] ≠ 0 alors
            début pour J := 1 pas 1 jusquà NP[I] faire IN(EP[I, J]) ;
                  pour J := 1 pas 1 jusquà NE1[I] faire IN(PX[I, J])
            fin
        fin ;
comment lire les données relatives aux appels non-formels et initialiser
          MA1, MP1 et MP2 ;
K := 1 ; TMP1 := faux ; TMP2 := faux ; TEST := faux ;
pour J := 1 pas 1 jusquà LL faire
        début LIRE ; I := PFA[1] ; SET
        fin note : un élément de MA1 est mis à vrai dès qu'un appel non-formel
            adéquat convient ;
TEST := vrai ;
comment lire les données relatives aux appels formels ;
pour I := 1 pas 1 jusquà JJ faire IN(NAPF[I], OFAPF[I]) ;
pour K := 1 pas 1 jusquà KK faire LIRE ;
comment la fin des lectures ;
si TMP1 alors début WARSHALL ; MP1MP2 fin ;
ITERCPT := 0 ; SORTIR ;
si ¬TMP2 alors allera FINI ;
```

```
ITER : TMP2 := faux ; TMP1 := faux ; TMA1 := faux ; TMA2 := faux ;
       pour I := 1 pas 1 jusquà II faire
       pour J := 1 pas 1 jusquà JJ faire
     début
     si MP2[I, J] alors
     début si MP2B[I, J] alors allera EXIT ;
           MP2B[I, J] := vrai ;
           pour K := OFAPF [J] pas 1 jusquà
           OFAPF[J] + NAPF[J] - 1 faire
           SET ; comment on vient d'examiner tous les appels formels au
                       paramètre j ;
     fin ; EXIT :
     fin boucles sur I, J ;
ITERCPT := ITERCPT + 1 ;
si TMP1 alors WARSHALL ;
si TMP1 ou TMP2 alors MP1MP2 ;
si TMP2 alors
début SORTIR ; allera ITER
fin
sinon si ¬TMA1 et ¬TMA2 alors
      début OUT (ITERCPT) ; allera FINI
      fin ;
SORTIR ;
FINI :
fin
fin du programme ;
```

L'algorithme ne vérifie pas si les références aux paramètres ENTRY et aux noms des procédures satisfont aux règles de protée. Nous admettrons que ces règles sont satisfaites.

En outre l'algorithme suppose implicitement - du fait qu'il traite les différentes occurrences des appels indépendamment du graphe de contrôle du programme - que, pour tout point du programme, il existe un chemin dans le graphe du programme permettant d'atteindre ce point depuis l'entrée du programme. Nous admettrons donc que cette hypothèse raisonnable est vérifiée. Le coeur de l'algorithme se situe dans l'examen itératif des éléments du tableau MP2. Il est clair qu'il suffit d'examiner un élément donné de valeur

vrai de MP2 une seule fois. Le traitement d'un élément vrai de MP2 peut entraîner des modifications des tableaux MP2, MP1, MA1 et MA2. Une modification de MP1 peut à son tour entraîner une modification de MP2 ; cette possibilité est reflétée dans l'appel de la procédure MP1MP2 en fin d'une itération.

Si le tableau MP2 n'est modifié, au cours d'une itération, ni directement ni indirectement (par MP1MP2), la variable TMP2 est à faux et l'algorithme s'arrête. Aucune modification subséquente des tableaux MA1, MA2, MP1, MP2 n'est plus possible.

Notons que la dernière itération (celle où TMP2 garde la valeur faux) peut néanmoins modifier MA1 ou MA2 ou les deux ; il est donc nécessaire d'effectuer un dernier appel à SORTIR avant la fin de l'algorithme.

Nous illustrerons le fonctionnement de l'algorithme par un exemple.

Considérons donc le schéma d'un programme-source présenté sous forme de l'arbre ASC dans la figure 1. (Le programme-source correspondant est donné par la figure 1').

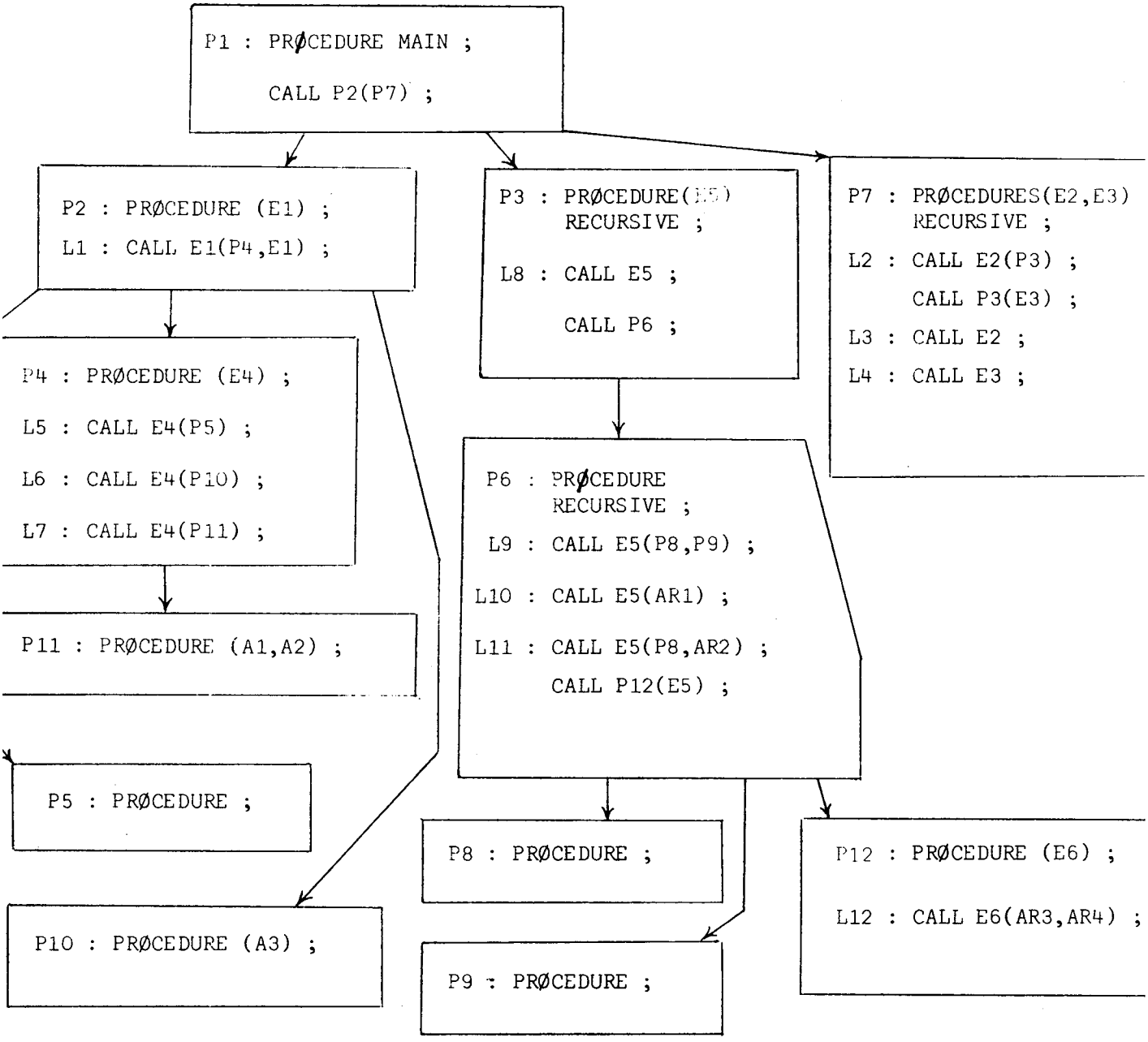

P1 : PRØCEDURE MAIN ;
CALL P2(P7) ;
P2 : PRØCEDURE (E1) ;
L1 : CALL E1(P4,E1) ;
P3 : PRØCEDURE(E5)
RECURSIVE ;
L8 : CALL E5 ;
CALL P6 ;
P7 : PRØCEDURES(E2,E3)
RECURSIVE ;
L2 : CALL E2(P3) ;
CALL P3(E3) ;
L3 : CALL E2 ;
L4 : CALL E3 ;
P4 : PRØCEDURE (E4) ;
L5 : CALL E4(P5) ;
L6 : CALL E4(P10) ;
L7 : CALL E4(P11) ;
P6 : PRØCEDURE
RECURSIVE ;
L9 : CALL E5(P8,P9) ;
L10 : CALL E5(AR1) ;
L11 : CALL E5(P8,AR2) ;
CALL P12(E5) ;
P11 : PRØCEDURE (A1,A2) ;
P5 : PRØCEDURE ;
P8 : PRØCEDURE ;
P12 : PRØCEDURE (E6) ;
L12 : CALL E6(AR3,AR4) ;
P10 : PRØCEDURE (A3) ;
P9 : PRØCEDURE ;

Figure 1.

Les paramètres E1, ..., E6 sont supposés être déclarés avec l'attribut ENTRY ; les paramètres A1, ..., A3 et les arguments AR1, ..., AR4 ne sont pas ENTRY. La numérotation des appels formels est indiquée par les étiquettes Li, $1 \leq i \leq 12$ (avec KK=12).

```
P1 : PROCEDURE MAIN ;
     .
     .
     CALL P2(P7) ;
     .
     .
     P2 : PROCEDURE(E1) ;
          .
          .
          L1 : CALL E1(P4, E1) ;
          .
          .
          P5 : PROCEDURE ;
               .
               .
               END ;
          P10 : PROCEDURE(A3) ;
               .
               .
               END ;
          P4 : PROCEDURE(E4) ;
               .
               .
               L5 : CALL E4(P5) ;
               .
               .
               L6 : CALL E4(P10) ;
               .
               .
               L7 : CALL E4(P11) ;
               .
               .
               P11 : PROCEDURE(A1, A2) ;
                     .
                     .
                     END ;
               .
               .
               END ;
          .
          .
          END ;
          .
          .
          .
```

(voir page suivante)

```
P3 : PROCEDURE(E5) RECURSIVE ;
     .
     .
     L8 : CALL E5 ;
     .
     .
     CALL P6 ;
     .
     .
     P6 : PROCEDURE RECURSIVE ;
          .
          .
          L9 : CALL E5(P8, P9) ;
          .
          .
          L10 : CALL E5(AR1) ;
          .
          .
          L11 : CALL E5(P8, AR2) ;
          .
          .
          CALL P12(E5) ;
          .
          .
          P8 : PROCEDURE ; ... END ;
          P9 : PROCEDURE ; ... END ;
          P12 : PROCEDURE(E6) ;
                .
                .
                L12 : CALL E6(AR3, AR4) ;
                .
                .
                END ;
          END ;
     .
     .
     END ;
     .
     .
P7 : PROCEDURE(E2, E3) RECURSIVE ;
     .
     .
     L2 : CALL E2(P3) ;
     .
     .
     CALL P3(E3) ;
     .
     .
     L3 : CALL E2 ;
     .
     .
     L4 : CALL E3 ;
     .
     .
     END ;
.
.
END ;
```

L'algorithme A1 fournit la matrice associée MA1 du graphe GPA ainsi que les ensembles AP_k, k = 1, ... KK, avec KK = 12, dont les éléments sont indiqués par les éléments de valeur vrai des lignes du tableau MA2.

Le graphe GPA correspondant à l'exemple de la figure 1 est donné par la figure 2.

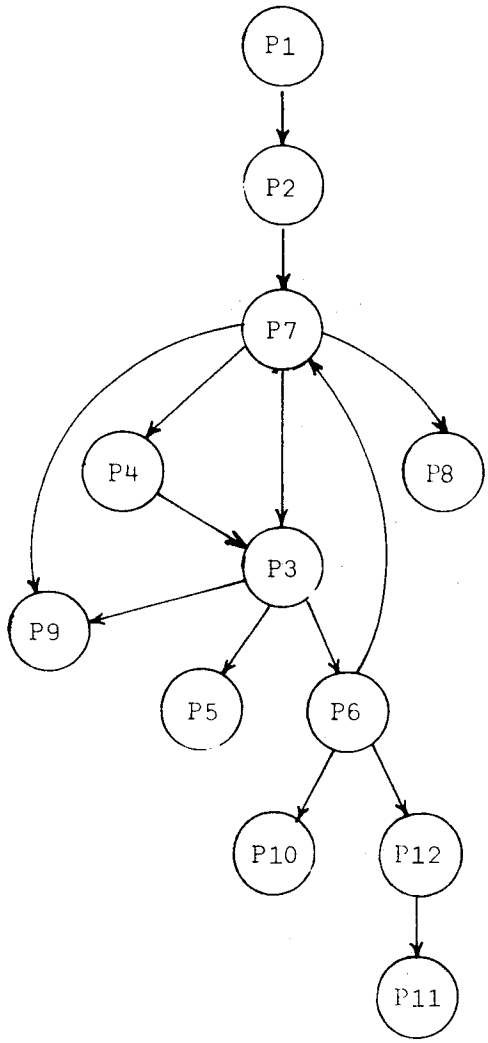


Figure 2.

Les ensembles AP_k obtenus sont les suivants :

AP_1 = {P7}
AP_2 = {P4}
AP_3 = {P8}
AP_4 = {P9}
$AP_5 = AP_6 = AP_7$ = {P3}
AP_8 = {P5, P9}
AP_9 = {P7}
AP_{10} = {P10}
AP_{11} = { }
AP_{12} = {P11}.

Notons que le fait que l'ensemble AP_{11} est vide nous conduit à déduire que le programme est erroné ou que le point L11 du programme n'est jamais atteint.

Dans la figure 3 nous illustrons par un graphe les séquences possibles des activations des procédures du programme schématisé dans la figure 1, séquences obtenues indépendamment du graphe de contrôle de ce programme. Tout arc du graphe de la figure 3 représente un appel non-formel (l'arc porte dans ce cas l'étiquette NF) ou un appel formel (l'arc est dans ce cas étiqueté par l'indice de l'appel formel) ; nous indiquons également, pour tout appel, les passages éventuels des arguments ENTRY.

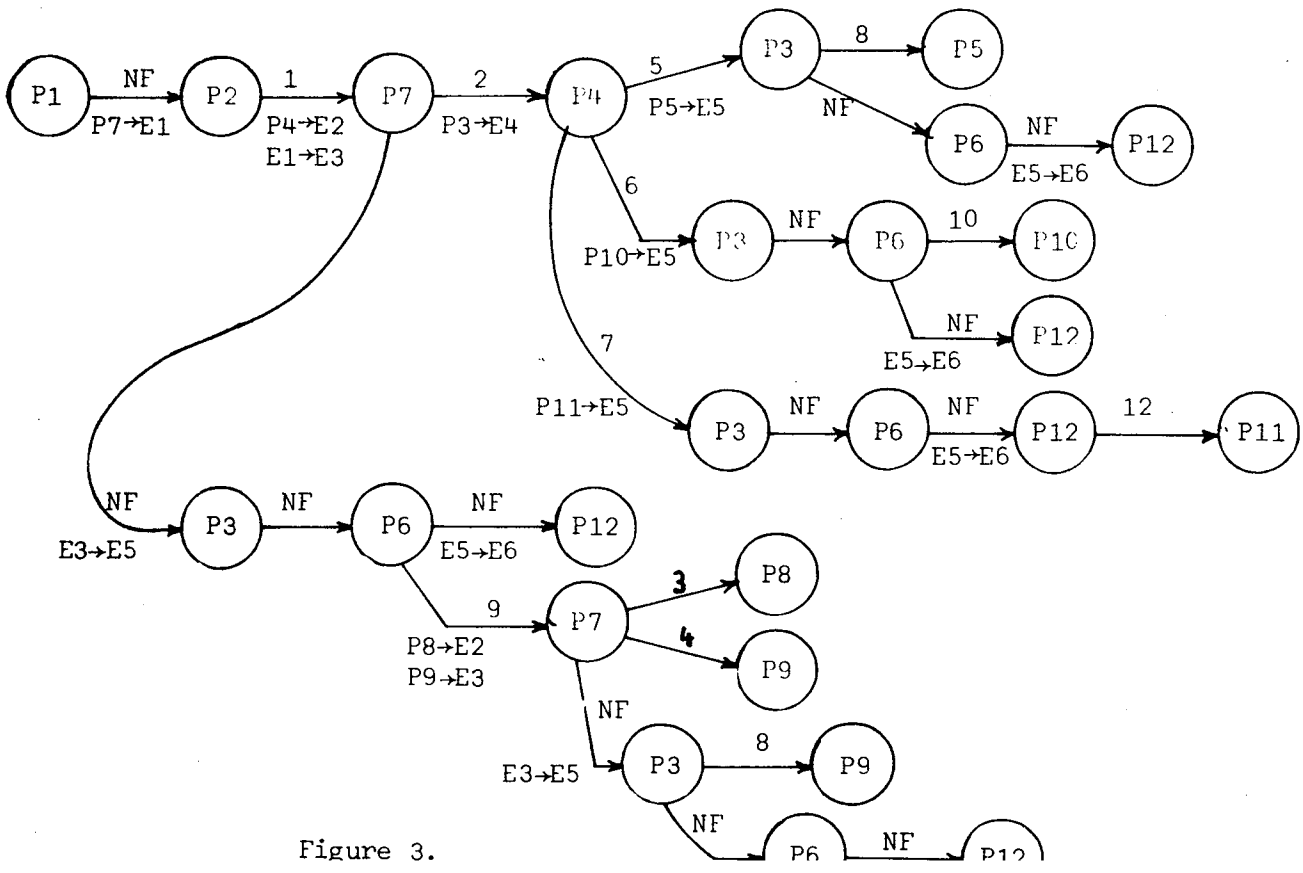


Figure 3.

On peut vérifier que le graphe de la figure 3 confirme les résultats de l'algorithme A1 (le GPA et les ensembles AP_k). D'autre part, pour cet exemple particulier, nous pouvons constater que le graphe de la figure 3 permet d'infirmer la possibilité du circuit [P7, P4, P3, P6] du graphe GPA.

Une liste (listing en anglais) complète de l'algorithme A1 est donnée en annexe 1 ainsi que les données à l'entrée et les résultats partiels et définitifs relatifs à l'exemple de la figure 1.

4.2.2 - Modifications des variables au niveau du programme-source -

Nous considérerons par la suite les modifications (des valeurs) des variables-source. Des informations déduites relatives aux modifications des variables-source déterminées ici seront exploitées dans les optimisations du programme en langage intermédiaire.

Définissons d'abord les termes utilisés dans ce chapitre.

On appellera variable-source un nom unique déduit d'un identificateur-source spécifiant une donnée dont l'attribut n'est ni LABEL ni ENTRY.

On distinguera dans l'ensemble VS des variables-source le sous-ensemble PAR = $\{para_j\}_j$ des paramètres des procédures du programme-source considéré (les paramètres sont les variables-source figurant dans les listes des paramètres des instructions PROCEDURE).

Dans l'ensemble VS on distinguera le sous-ensemble VP des variables-source qui apparaissent au moins une fois en tant qu'arguments (c'est-à-dire dans une liste des arguments d'un appel de programme-source) passés par référence (c'est-à-dire modifiables par l'appel où elles sont référencées).

On dénotera par ARGS l'ensemble des occurrences des arguments passés par référence - occurrences qui désignent les variables de VP - de tous les appels du programme-source considéré. Notons que ARGS correspond à l'ensemble VP où l'on distingue les occurrences.

On dira qu'une instruction d'affectation dont la destination est une variable-source v (ou un élément d'une variable-source v dans le cas où cette dernière est un tableau) modifie explicitement v et toute variable-source v' déclarée DEFINED sur v. Nous ne traiterons pas les modifications des variables-source

dues à l'attribut INIT de l'instruction de déclaration correspondante.

On appellera fief de déclaration de v ϵ VS le fief dont le texte contient la déclaration de v.

On appellera corps de déclaration de v ϵ VS le corps de procédure contenant le fief de déclaration de v.

On appellera portée de v la plus petite région contenant le fief de déclaration de v.

On dira que v ϵ VP est modifiée indirectement par une occurrence d'appel ap_k si v^i est une occurrence d'argument ϵ ARGS en position p dans la liste des arguments de l'appel ap_k et si ap_k est tel qu'il existe un cp ϵ AP_k (cf. 4.2.1) dont le $p^{ième}$ paramètre $para_j$ ϵ PAR est modifié par la suite. On peut dire plus simplement que v (par son occurrence v^i) est passée à un paramètre $para_j$ qui est modifié par la suite. Une définition plus précise de la modification indirecte sera donnée en 4.2.2.1.

On dira que v ϵ VS est modifiée localement si elle n'est pas modifiée globalement et s'il existe une modification explicite ou indirecte dans le corps de déclaration de v. Notons qu'ici la localité est relative aux corps des procédures.

On dira que v ϵ VS est modifiée globalement si elle n'est pas modifiée localement et s'il existe une modification explicite ou indirecte dans la portée de v.

Nous parlerons aussi des modifications locales explicites et indirectes et des modifications globales explicites et indirectes. La signification de ces termes découle des définitions précédentes.

On dira que v ϵ VS est modifiée implicitement par une instruction BEGIN - notée $bloc_i$ - si v est modifiée localement ou globalement dans la région de bloc-début (cf. 4.1) associée à l'instruction $bloc_i$ en question. Il faut bien évidemment que l'instruction $bloc_i$ se trouve dans la portée de v ; toutefois $bloc_i$ ne doit pas correspondre au fief de déclaration de v. Une instruction $bloc_i$ représentera (cf. ch. 6) le bloc BEGIN correspondant au niveau de l'entité (bloc BEGIN, bloc PROCEDURE ou boucle DO) directement englobante.

On dira également que $v \in VS$ est modifiée implicitement par une occurrence d'appel - notée ap_i - si v est modifiée localement ou globalement dans l'une au moins des procédures qui peuvent être activées (directement ou indirectement) par l'appel ap_i en question.

On dira que $v \in VS$ est modifiée implicitement par une boucle DO - notée $boucle_i$ - si la boucle DO est dans la portée de v et si v est modifiée explicitement, indirectement ou implicitement dans le corps de la boucle en question. Une instruction $boucle_i$ représentera la boucle DO correspondante au niveau de l'entité directement englobante.

Nous nous intéresserons par la suite à la détermination des informations suivantes :

- les points (c'est-à-dire les occurrences d'appels) des modifications indirectes des variables-source ;
- l'ensemble $\underline{MODB_i}$ des variables-source modifiées implicitement par une instruction $bloc_i$ ou $boucle_i$;
- l'ensemble $\underline{MODAP_j}$, $j = 1, \ldots J$, des variables-source modifiées implicitement par l'occurrence d'appel ap_j du programme-source.

Nous commencerons par la détermination des modifications indirectes (en 4.2.2.1). Nous utiliserons, pour ce faire, les résultats de l'algorithme A1 (les ensembles AP_k) et nous supposerons que les modifications explicites (locales et globales) sont facilement déterminées par l'analyse du programme-source.
Ayant déterminé les modifications indirectes nous ne considérerons par la suite que les variables-source de VS qui sont modifiées au moins une fois (explicitement ou indirectement, localement ou globalement) : soit $\underline{VSM} \subset VS$ l'ensemble de ces variables-source. Nous supposerons que l'ensemble VSM est facile à déterminer.

La détermination des ensembles $MODB_i$ et $MODAP_j$ sera faite (en 4.2.2.2) en plusieurs étapes :

- nous traiterons d'abord les variables-source de VSM ayant les propriétés suivantes (soit $\underline{VSML}$ l'ensemble de ces variables-source) :

  $v \in VSML$ si et seulement si :

  - toutes les modifications de v sont locales (indirectes ou explicites) et

• si le corps de déclaration de v correspond à une procédure ayant l'attribut RECURSIVE alors v est déclarée avec l'attribut AUTOMATIC

- nous traiterons ensuite les variables-source de l'ensemble VSM - VSML selon leurs corps de déclaration et selon leurs attributs (AUTOMATIC ou STATIC).

4.2.2.1 - Modifications indirectes : la détermination de l'attribut MOD -

Rappelons qu'il s'agit ici de déterminer les occurrences des arguments d'un programme-source - occurrences qui désignent des variables-source dont l'attribut n'est ni ENTRY ni LABEL et qui sont passées par référence - qui peuvent être modifiées par les appels qui les référencent.

A cet effet nous avons défini, en 4.2.2, l'ensemble ARGS et l'ensemble PAR. Notons que l'ensemble ARGS, aux occurrences près, est l'ensemble VP (cf. 4.2.2) et que les ensembles VP et PAR peuvent avoir des éléments en commun.

En utilisant les résultats de A1 (les AP_i, cf. 4.2.1) et en considérant les appels du programme (dont les occurrences seront dénotées par ap_k) nous pouvons construire les deux tableaux booléens suivants :

- ARGUM, de dimension card(ARGS) x card(PAR), est défini de la façon suivante :

ARGUM(n, p) = vrai (et faux sinon) si l'appel ap_k référençant $arg_n \in$ ARGS en position q est tel qu'il existe un cp $\in AP_k$ dont le $q^{ième}$ paramètre est $para_p \in$ PAR c'est-à-dire l'argument arg_n est passé au paramètre $para_p$;

- MP3, de dimension card(PAR) x card(PAR), défini ainsi :

MP3(i, j) = vrai (et faux sinon) s'il existe au moins un appel ap_k dont l'argument $arg_n \in$ ARGS en position p est une occurrence d'un $para_i \in$ PAR et s'il existe un cp $\in AP_k$ dont le $p^{ième}$ paramètre est $para_j \in$ PAR. Autrement dit : le $para_i$ est passé au $para_j$.

Notons qu'on pourrait inclure l'obtention de MP3 dans l'algorithme A1 (MP3 est analogue à MP1).

On peut construire ensuite le vecteur-colonne booléen PARMOD, de dimension card(PAR) x 1, défini ainsi :

PARMOD(j) = vrai (et faux sinon) s'il existe au moins une modification explicite (locale ou globale) de $para_j \in PAR$. Notons que nous avons interdit les recouvrements (par DEFINED) des paramètres ; ceci justement dans le but de simplifier le travail présenté ici.

Soit finalement le vecteur-colonne booléen ARGSMOD, de dimension card(ARGS) x 1, défini ainsi :

ARGSMOD(n) = vrai (et faux sinon) si l'argument $arg_n \in ARGS$ est modifié par suite de l'appel ap qui le référence ; arg_n reçoit alors l'attribut MOD.
Si $v \in VP$ est la variable-source dont arg_n (dont l'attribut est MOD) est une occurrence nous dirons que v est modifiée indirectement par ap (cf. 4.2.2).

Le vecteur ARGSMOD s'obtient comme suit :

PARMOD := $\widehat{MP3}$ * PARMOD $\vee$ PARMOD ; $\widehat{MP3}$ étant la fermeture transitive de MP3 ; et ensuite :

ARGSMOD := ARGUM * PARMOD ;

Pour les arguments simples de ARGS nous définissons l'attribut VAL $\equiv \neg$MOD qui signifie que l'argument simple est candidat pour un passage par valeur (voir 5.3.3).

4.2.2.2 - Détermination des modifications implicites (les ensembles $MODB_i$ et $MODAP_j$) -

Rappelons tout d'abord (cf. 4.2.2) que la détermination des modifications implicites sera faite en plusieurs étapes : nous traiterons d'abord les variables-source de l'ensemble VSML et nous traiterons ensuite :

- les modifications implicites, par des occurrences d'appels, des variables de l'ensemble VSM - VSML = VSG $\cup$ VSPEC (avec VSG $\cap$ VSPEC = $\emptyset$) ; l'ensemble VSPEC est l'ensemble des variables-source ayant l'attribut AUTOMATIC et dont les corps de déclaration correspondent aux procédures ayant l'attribut RECURSIVE. Nous traiterons séparément les variables de VSG et de VSPEC.

Les variables de VSG seront traitées selon leurs corps de déclaration et celles de VSPEC, selon leurs fiefs de déclaration ;

- les modifications implicites, par des instructions $bloc_i$ et $boucle_i$, des variables de VSG $\cup$ VSPEC.

Nous utiliserons les notations suivantes :

- $CP = \{cp_i\}_i$ est l'ensemble des corps des procédures du programme-source considéré (cf. 4.2.1) ;
- ASF_i est l'arbre des fiefs du cp_i (cf. 4.1) ; en tant que graphe orienté (l'orientation est dans le sens des niveaux croissants) nous l'écrirons encore (X_i, V_i, Γ_i) où $X_i = \{f^i_j\}_j$ c'est-à-dire l'ensemble des fiefs de l'arbre ASF_i et où f^i_1 est le fief-procédure de ASF_i. A tout fief $f^i_j \in X_i$, $j \geq 2$, correspond une instruction $bloc^i_j$ ou une instruction $boucle^i_j$ (cf. 4.2.2).
- $VSML_i$ est l'ensemble des variables-source de VSML ayant cp_i pour corps de déclaration ;
- $FR = \{fr_j\}_j$ est l'ensemble des fiefs de portée contenus dans les corps des procédures ayant l'attribut RECURSIVE ;
- VSG_i est l'ensemble des variables-source de VSG ayant cp_i pour corps de déclaration ;
- $VSPEC_j$ est l'ensemble des variables-source de VSPEC ayant fr_j pour fief de déclaration, $fr_j \in FR$. Nous ne considérerons par la suite, bien évidemment, que les ensembles VSG_i, $VSML_i$ ($i = 1, \ldots card(CP)$) et $VSPEC_j$ ($j = 1, \ldots card(FR)$) non vides ;
- $EFD = \{fd_k\}_k$ est l'ensemble des fiefs-début et des fiefs des boucles qui correspondent aux instructions $bloc_k$ et $boucle_k$ du programme-source considéré.

Nous encoderons les informations initiales relatives aux modifications explicites et indirectes (ces dernières ayant été déterminées en 4.2.2.1) dans les tableaux booléens suivants (dont les éléments sont initialement tous <u>faux</u>).

- les tableaux ML_i, $i = 1, \ldots, card(CP)$, des dimensions respectives $card(X_i) \times card(VSML_i)$. Un élément $ML_i(k, j)$ est initialisé à <u>vrai</u> si la variable $v_j \in VSML_i$ est modifiée explicitement ou indirectement dans le fief f^i_k et si f^i_k - dans le cas où f^i_k est un fief-début - <u>n'est pas</u> le fief de déclaration de v_j (rappelons que seules les modifications implicites seront étudiées ici) ;
- les tableaux MG_i des dimensions respectives $card(CP_i) \times card(VSG_i)$; CP_i est l'ensemble des corps des procédures se trouvant dans les portées des variables de VSG_i plus le corps cp_i (c'est l'ensemble des sommets du sous-arbre de racine cp_i de l'arbre ASC, cf. 4.1). Si $v_j \in VSG_i$ est déclarée avec l'attribut AUTOMATIC alors un élément $MG_i(k, j)$ est mis à <u>vrai</u> si v_j

est modifiée globalement dans le corps $cp_k \in CP_i$; sinon (c'est-à-dire si $v_m \in VSG_i$ est déclarée avec l'attribut STATIC) un élément $MG_i(k, m)$ est mis à vrai si v_m est modifiée localement ou globalement dans $cp_k \in CP_i$;

- les tableaux $MGSP_j$ des dimensions respectives $card(CSP_j) \times card(VSPEC_j)$; CSP_j est l'ensemble des corps des procédures se trouvant dans la portée des variables de $VSPEC_j$. Un élément $MGSP_j(i, k)$ est mis à vrai si $v_k \in VSPEC_j$ est modifiée globalement (v_k a l'attribut AUTO) dans le corps $cp_i \in CSP_j$;

- le tableau MGF de dimension card(EFD) x card(VSM-VSML) ; un élément MGF(k, j) est initialisé à vrai si la variable $v_j \in$ VSM-VSML est modifiée explicitement ou indirectement dans le fief fd_k et si fd_k - dans le cas où fd_k est un fief-début - n'est pas le fief de déclaration de v_j.

Les valeurs des tableaux ML_i et MGF pourront être mises à jour par la suite.

Nous déterminerons d'abord les modifications implicites des variables de VSML ; pour ce faire nous propagerons les informations initiales (les éléments de valeur vrai) des tableaux ML_i de la façon suivante :

- soit $ML_i(k, j)$ un élément vrai d'un ML_i et
- soit alors $F = \{f_n\}_n \subset \hat{\Gamma}_i^{-1}$ l'ensemble des fiefs prédécesseurs, dans ASF_i, du fief f_k^i, tels que $\forall n$, f_n est dans la portée de la variable v_j et - dans le cas où f_n est un fief-début - f_n n'est pas le fief de déclaration de v_j. La mise à jour de ML_i qui correspond à la propagation de la valeur vrai de l'élément $ML_i(k, j)$ est alors effectuée par la mise à vrai de tous les éléments de la colonne j de ML_i, éléments correspondant, par leurs indices des lignes, aux fiefs $f_n \in F$.

Le traitement des variables-source de l'ensemble VSML est ainsi terminé. En effet, les modifications (explicites et indirectes) de ces variables étant uniquement locales et leurs corps de déclaration, si elles sont déclarées avec l'attribut STATIC, n'ayant pas l'attribut RECURSIVE, il est évident que leurs modifications implicites éventuelles sont dues uniquement aux instructions $bloc_i$ et $boucle_i$ de leurs corps de déclaration.

Les ensembles $MODB_i$ (cf. 4.2.2), initialement vides, sont alors mis à jour de la façon suivante :

- $\forall i, \forall k, \forall j$: si $ML_i(k, j)$ est vrai alors $MODB_n := MODB_n \cup \{v_j\}$ où $MODB_n$ est l'ensemble correspondant à l'instruction $bloc_n = bloc_k^i$ ou à l'instruction $boucle_n = boucle_k^i$ et $v_j \in VSML_i$.

Un exemple est donné dans la figure 1. L'arbre ASF_i du corps de procédure cp_i associé à la procédure P1 de la figure 1 est donné dans la figure 2. En fin du traitement des variables de $VSML_i$ les ensembles $MODB_4$, $MODB_2$ et $MODB_5$ contiennent la variable V($\in VSML_i$) de la figure 1.

```
P1 : PROCEDURE ; DCL V ;
     .
     .
     V := 0 ;
     .
     .
     B2 : BEGIN ;
          .
          .
          B3 : BEGIN ;
               .
               .
               END ;
          .
          .
          L4 : DO ... ;
               .
               .
               V := A+2 ; /* modification explicite*/
               .
               .
               END ;
          .
          .
          END B2 ;
     .
     .
     B5 : BEGIN ;
          P2 : PROCEDURE(PAR) ;
               .
               .
               PAR := PAR+1 ;
               .
               .
               RETURN(PAR*PAR) ;
               END P2 ;
          .
          .
          V1 := P2(V) ; /* modification indirecte ; l'occurrence de V a
                           l'attribut MOD */
          .
          .
          END B5 ;
.
.
END P1 ;
```

Figure 1.

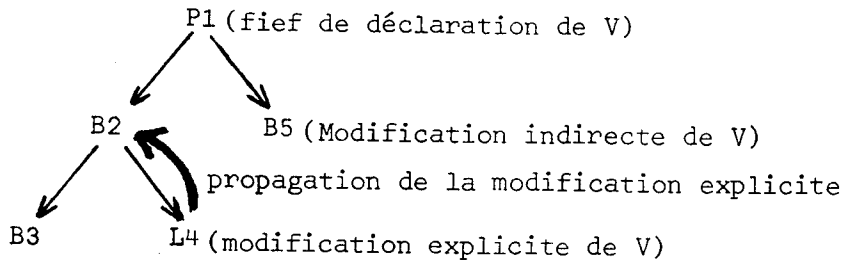


Figure 2.

Les modifications implicites des variables d'un ensemble VSG_i par des occurrences d'appels formels et non-formels (cf. 4.2.1) sont dues aux modifications locales ou globales explicites et indirectes encodées dans le tableau MG_i.

Les occurrences d'appels à considérer sont celles qui se trouvent dans les corps de l'ensemble CP_i : soit EAP_i l'ensemble de ces occurrences d'appels.

Nous encoderons les informations initiales relatives aux appels de EAP_i dans un tableau booléen MAP_i, de dimension $card(EAP_i) \times card(CP)$, dont la signification initiale sera celle du tableau MA2 défini en 4.2.1 : un élément $MAP_i(k, j)$ initialisé à vrai indique que l'appel $ap_k \in EAP_i$ peut être un appel direct à la procédure dont le corps est dénoté par $cp_j \in CP$.

Le tableau MAP_i sera ensuite mis à jour de façon à ce que sa signification définitive soit la suivante : un élément $MAP_i(k, j)$ de valeur vrai indiquera que l'appel ap_k peut être un appel direct ou indirect à la procédure dont le corps est dénoté par cp_j. La valeur finale de MAP_i sera donc définie par :

$$MAP_i := MAP_i \vee (MAP_i * MA1) \equiv MAP_i * (\widehat{MA1} \vee I_i) \text{ où}$$

$\widehat{MA1}$ est la fermeture transitive de la matrice associée au graphe GPA (cf. 4.2.1) et où I_i est la matrice unité d'ordre card (CP).

Comme les variables de VSG_i ne peuvent être modifiées localement ou globalement que dans les corps de CP_i nous utiliserons par la suite le tableau $MAP1_i$, de dimension $card(EAP_i) \times card(CP_i)$, obtenu du tableau MAP_i par la suppression des colonnes correspondant aux éléments de l'ensemble $CP-CP_i$.

Les modifications implicites des variables de VSG_i par les appels de EAP_i sont alors définies par un tableau booléen $MODAPPELS_i$ de dimension $card(EAP_i) \times card(VSG_i)$ obtenu par :

$$MODAPPELS_i := MAP1_i * MG_i$$

Un élément $MODAPPELS_i(k, j)$ est <u>vrai</u> si $v_j \in VSG_i$ est modifiée implicitement par l'occurrence d'appel $ap_k \in EAP_i$.

Pour une variable $v_j \in VSG_i$ donnée, les seules occurrences d'appels de EAP_i qui nous intéressent sont celles qui se trouvent dans la portée de v_j ; par conséquent nous éliminerons, dans $MODAPPELS_i$, les appels qui ne se trouvent pas dans la portée de v_j.

Soit alors $FILTRE_i$ le tableau booléen de la dimension de $MODAPPELS_i$ dont un élément $FILTRE_i(j, k)$ est <u>vrai</u> (et <u>faux</u> sinon) si cp_j est dans la portée de $v_k \in VSG_i$. Le tableau $MODAPPELS_i$ devient :

$$MODAPPELS_i := MODAPPELS_i \wedge FILTRE_i$$

Un exemple illustrant le cas des appels modifiant implicitement une variable AUTO d'un VSG_i et se trouvant en dehors de la portée de cette variable est donné dans les figures 3 et 4. La figure 4 représente (en traits gras) l'arbre ASC (cf. 4.1) du programme de la figure 3 et les activations directes et certaines des activations indirectes dues aux appels (en traits fins). Les appels indirects sont marqués par un astérisque ; la lettre M indique, pour les sommets, la présence d'une modification explicite ou indirecte dans le corps de procédure correspondant et, pour les arcs, les appels modifiant implicitement la variable V de la figure 3. Tous les cinq appels (étiquetés par AP_i) modifient implicitement V ; EAP_1 = {AP3, AP4, AP5} ; seuls les appels AP3 et AP4 se trouvent dans la portée de V ; l'appel AP5 sera donc éliminé de la colonne correspondant à V du tableau $MODAPPELS_1$.

```
.
.
PO : PROCEDURE(E1) RECURSIVE ; DCL E1 ENTRY ;
     .
     .
     AP1 : CALL E1 ;
     .
     .
     AP2 : CALL P1 ;
     .
     .
     END PO ;
P1 : PROCEDURE ;
     .
     .
     B1 : BEGIN ; DCL V AUTO ;
          .
          .
          AP3 : CALL PO(P2) ;
          .
          .
          AP4 : CALL P3(P2) ;
          .
          .
          P2 : PROCEDURE ;
               .
               .
               V := V + 1 ; /* modification globale explicite */
               .
               .
               END P2 ;
          .
          .
          END B1 ;
     P3 : PROCEDURE(E2) ; DCL E2 ENTRY ;
          .
          .
          AP5 : CALL E2 ;
          .
          .
          END P3 ;
     .
     .
     END P1 ;
.
.
```

Figure 3.

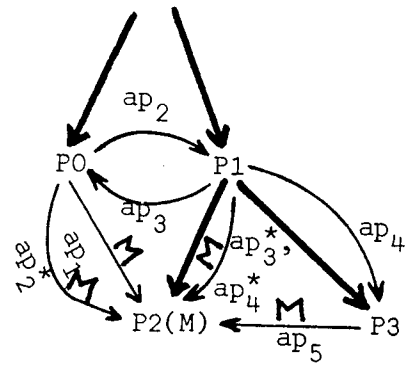


Figure 4.

Un deuxième exemple, illustrant les modifications implicites, par des appels, d'une variable STATIC d'un VSG_i est donné dans les figures 5 à 7. La signification de la figure 6 est la même que celle de la figure 4. La figure 7 donne une séquence d'activations possible, permettant à l'appel AP4 de modifier implicitement la variable V de la figure 5.

```
.
.
P1 : PROCEDURE RECURSIVE ; DCL V STATIC ;
     .
     .
     AP1 : CALL P2(P1) ;
     .
     .
     AP2 : CALL P6(P3) ;
     .
     .
     P2 : PROCEDURE(E1) RECURSIVE ; DCL E1 ENTRY ;
          .
          .
          AP3 : CALL E1 ;
          .
          .
          AP4 : CALL P2(P4) ;
          .
          .
          END P2 ;
     P4 : PROCEDURE RECURSIVE ;
          .
          .
          AP5 : CALL P2(P1) ;
          .
          .
          END P4 ;
     P3 : PROCEDURE ;
          .
          .
          V := V+1 ; /* modification globale */
          .
          .
          AP6 : CALL P5 ;
          .
          .
```

```
        .
        .
        P5 : PROCEDURE ;
             .
             .
             END P5 ;
        .
        .
        END P3 ;
   .
   .
   END P1 ;
P6 : PROCEDURE(E2) ; DCL E2 ENTRY ;
   .
   .
   AP7 : CALL E2 ;
   .
   .
   END P6 ;
.
.
```

Figure 5.

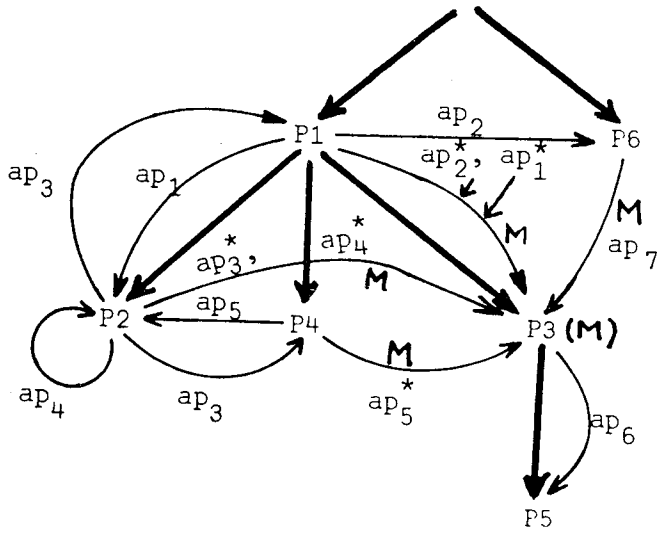
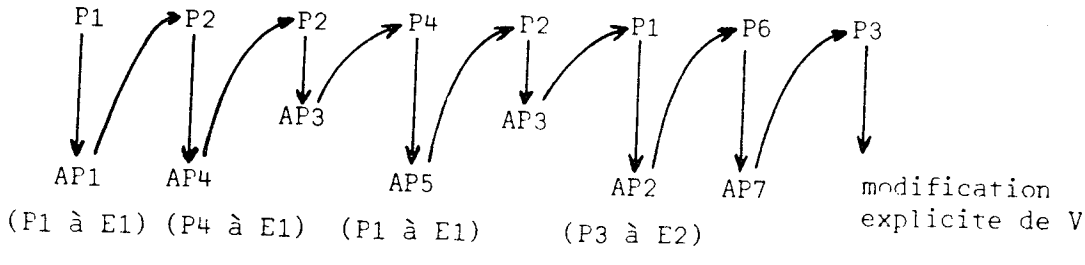


Figure 6.



Notons que tous les appels, sauf AP7, de la figure 5 sont dans la portée de V et appartiennent à EAP_1 ; tous les appels - excepté AP6 - modifient implicitement V.

Les modifications implicites des variables d'un ensemble $VSPEC_j$ par des occurrences d'appels formels et non-formels sont dues aux modifications globales encodées dans le tableau $MGSP_j$.

La détermination de ces modifications est analogue à celle des modifications implicites, par des appels, des variables des ensembles VSG_i ; la différence essentielle réside dans la définition de l'ensemble EAP'_j des occurrences d'appels à prendre en considération.

Avant de définir EAP'_j nous introduirons les notations suivantes :

- VP_j désigne l'ensemble des valeurs d'un paramètre par_j (voir 4.2.1) ; $VP_j \subset CP$ est déterminé par les éléments vrai de la colonne j de la matrice MP2 obtenue par l'algorithme A1 (cf. 4.2.1) ;

- un argument ENTRY a_i détermine un ensemble de valeurs $VA_i \subset CP$ qui est soit un ensemble $\{cp_k\}$, $cp_k \in CP$, si l'argument a_i est un nom de procédure, soit un ensemble VP_j si l'argument a_i est une occurrence de par_j.

L'ensemble EAP'_j est alors défini ainsi : un appel $ap_k \in EAP'_j$ si et seulement si :

- ap_k se trouve dans la portée des variables de $VSPEC_j$ et
- un élément au moins de l'ensemble CSP_j (qui est l'ensemble des corps des procédures se trouvant dans la portée des variables de $VSPEC_j$) doit être contenu :

- dans l'ensemble $AP_k \subset CP$ (cf. 4.2.1) des corps des procédures pouvant être appelées par ap_k ou
- dans l'ensemble $\cup_i VA_i$ des corps des procédures "passées" par ap_k, où les a_i correspondants sont les arguments ENTRY de l'appel ap_k.

La détermination des modifications implicites, par des appels, des variables de $VSPEC_j$ se poursuit ensuite de la même façon que pour les variables d'un VSG_i. Un tableau booléen MAP'_j de dimension $card(EAP'_j) \times card(CP)$ encodera les informations initiales relatives aux appels de EAP'_j. La valeur finale de MAP'_j est définie par : $MAP'_j := MAP'_j * (\hat{MA1} \vee I_j)$.

Un tableau booléen $MAP1'_j$, de dimension $card(EAP'_j) \times card(CSP_j)$ est ensuite extrait de MAP'_j et les modifications implicites, par des appels, des variables de $VSPEC_j$ sont alors définies par un tableau booléen $MODAPPELS'_j$ de dimension $card(EAP'_j) \times card(VSPEC_j)$ obtenu par :

$$MODAPPELS'_j := MAP1'_j * MGSP_j$$

Un exemple illustrant les modifications implicites, par des appels, d'une variable de $VSPEC_j$ est donné dans les figures 8 à 11. Les notations sont celles des exemples précédents. La figure 9 représente tous les appels directs du programme de la figure 8 et le graphe dont la matrice associée est $\widehat{MA1}$ (le graphe est restreint aux procédures du programme de la figure 8) ; la figure 10 représente la valeur finale de MAP'_1.

```
.
.
P0 : PROCEDURE(E1) RECURSIVE ; DCL E1 ENTRY ;
     .
     .
     AP1 : CALL P1 ;
     .
     .
     P1 : PROCEDURE RECURSIVE ;
          .
          .
          AP2 : CALL E1 ;
          .
          .
          B1 : BEGIN ; DCL V AUTO ;
               .
               .
               AP3 : CALL E1 ;
               .
               .
               AP4 : CALL P0(P2) ;
               .
               .
               AP5 : CALL P3 ;
               .
               .
               P2 : PROCEDURE ;
                    .
                    .
                    V := V+1 ; /* modification globale */
                    .
                    .
                    END P2 ;
               .
               .
               END B1 ;
          .
          .
          P3 : PROCEDURE ;
               .
               .
               AP6 : CALL E1 ;
               .
               .
               END P3 ;
          .
          .
          END P1 ;
     .
     .
     END P0 ;
.
.
```

Figure 8.

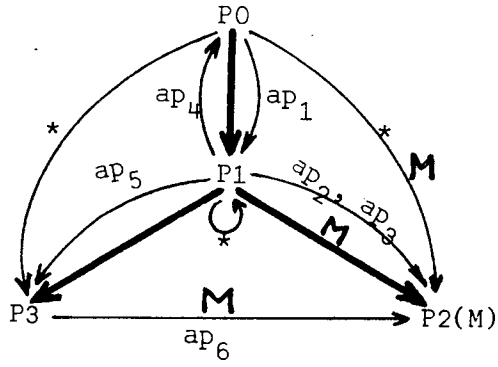


Figure 9.

L'ensemble EAP'_{B1} de l'exemple contient AP3 et AP4 ; AP3 peut, en effet, appeler P2 et AP4 "passe" P2 et le corps de procédure associé à la procédure P2 appartient à CSP_{B1}.

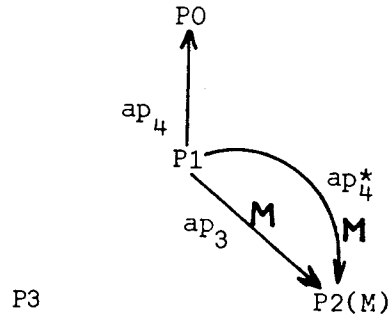


Figure 10.

D'après les figures 9 et 10 tous les appels du programme de la figure 8 peuvent modifier implicitement la variable V. Toutefois la figure 11 montrera, par le schéma des séquences possibles d'activations (les occurrences d'activations sont distinguées par des indices) que seuls AP1 et AP4 peuvent modifier implicitement la variable V (le schéma des activations ne tient pas compte du graphe de programme).

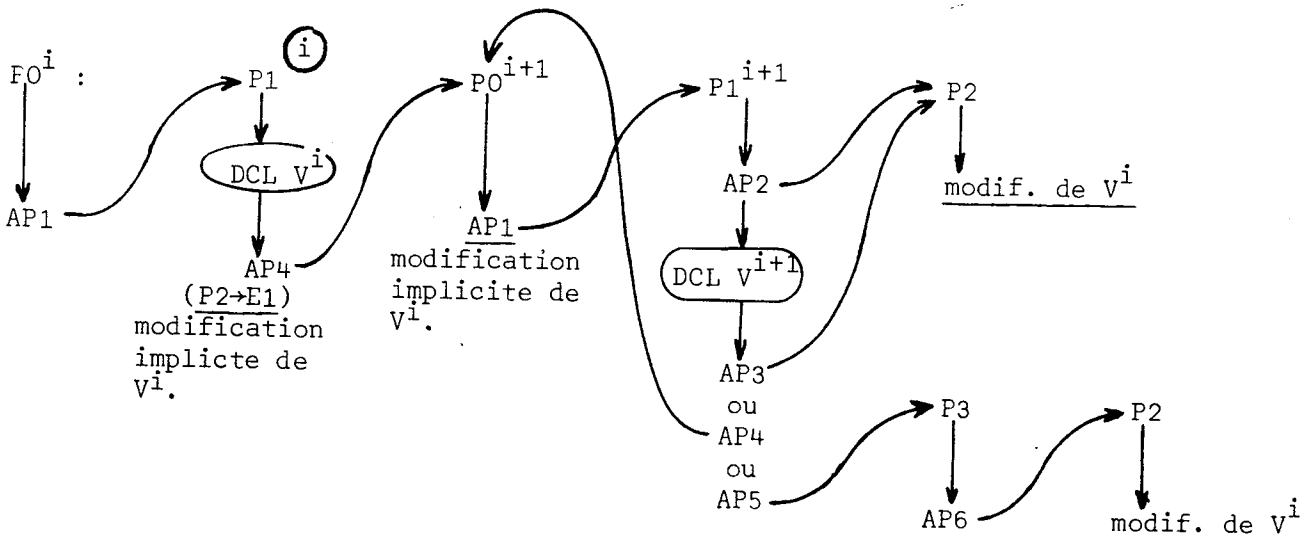


Figure 11.

Nous pouvons en déduire que la définition de EAP'_j, donnée précédemment, n'est pas suffisamment restrictive.

Par la suite nous restreindrons l'ensemble EAP'_j en tenant compte des deux propositions suivantes :

Proposition 1 : un appel $ap_k \in EAP'_j$ ne peut modifier implicitement aucune variable de $VSPEC_j$ si :

C1 : ap_k est un appel formel et

C2 : ap_k se trouve dans le corps de déclaration des variables de $VSPEC_j$ et

C3 : aucun argument de l'appel ap_k n'est le nom d'une procédure dont le corps de procédure correspondant appartient à l'ensemble CSP_j.

Proposition 2 : un appel $ap_k \in EAP'_j$ ne peut modifier implicitement aucune variable de $VSPEC_j$ si :

C3 : (de la proposition 1) et

C4 : l'ensemble AP_k (cf. 4.2.1) ne contient aucun corps de procédure appartenant à CSP_j (c'est-à-dire : $AP_k \cap CSP_j = \emptyset$)

Les deux propositions résultent des considérations relatives aux occurrences d'activations du corps de déclaration des variables de $VSPEC_j$. Un exemple, donné dans les figures 12 et 13 établit la nécessité de la conjonction des trois conditions de la proposition 1 ; il nous montre que les appels AP5, AP2 et AP4 (vérifiant respectivement $C1 \wedge \neg C2 \wedge C3$, $C1 \wedge C2 \wedge \neg C3$ et $\neg C1 \wedge C2 \wedge C3$) peuvent modifier implicitement la variable V de la figure 12.

```
.
.
P1 : PROCEDURE(E1, E2) RECURSIVE ; DCL (E1, E2) ENTRY ;
     DCL V AUTO ;
     .
     .
     AP1 : CALL E1(E2) ;
     .
     .
     AP2 : CALL E1(P3) ;
     .
     .
     AP3 : CALL P1(P2,P3) ;
     .
     .
     AP4 : CALL P3 ;
     .
     .
     P2 : PROCEDURE(E3) ; E3 ENTRY ;
          .
          .
          AP5 : CALL E3 ;
          .
          .
          END P2 ;
     P3 : PROCEDURE ;
          .
          .
          V := V+1 ; /* modification globale */
          .
          .
          END P3 ;
     .
     .
     END P1 ;
.
.
```

Figure 12.

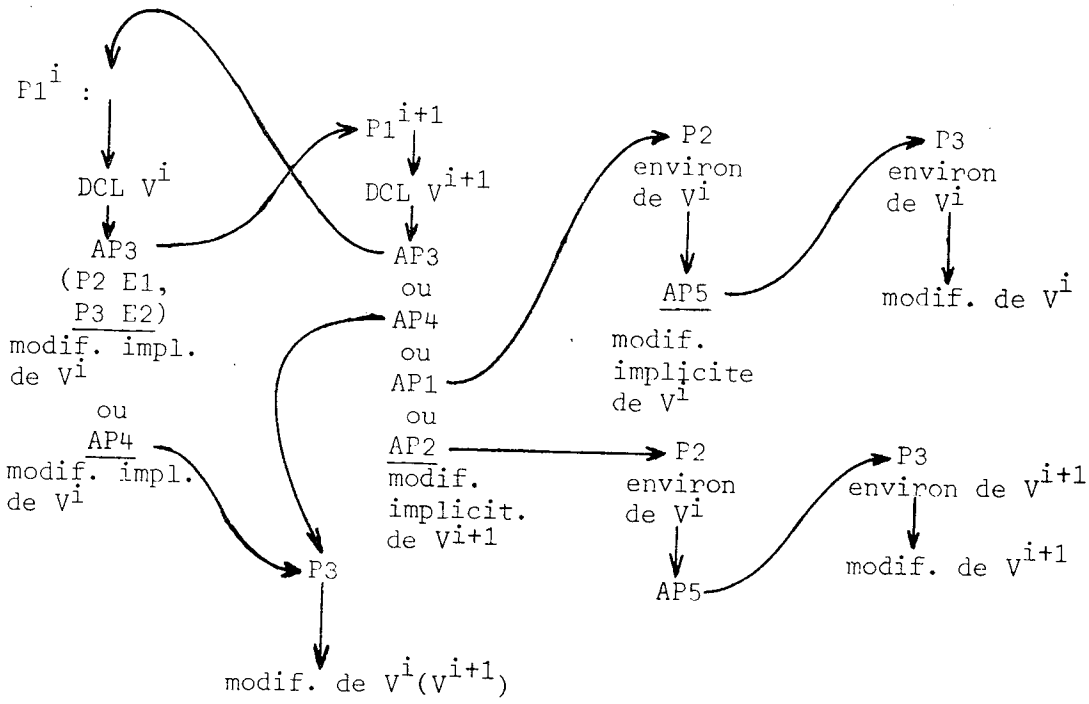


Figure 13.

Nous déterminerons, par conséquent, le tableau $MODAPPELS'_j$ en partant de l'ensemble $EAPR_j$ défini par : $ap_k \in EAPR_j$ si et seulement si :

- $ap_k \in EAP'_j$ et
- ap_k ne vérifie pas les conditions de la proposition 1 ou celles de la proposition 2.

Toutes les informations relatives aux modifications implicites, par des appels, des variables de VSM-VSML seront rassemblées dans un tableau booléen MODAPPELS de dimension J x card(VSM-VSML) où J est le nombre total des occurrences d'appels ; ce tableau résume les informations des tableaux $MODAPPELS_i$ ($i = 1, \ldots$ card(CP)) et $MODAPPELS'_j$ ($j = 1, \ldots$ card(FR)). Un élément MODAPPELS(k, j) est vrai si $v_j \in$ VSM-VSML est modifiée implicitement par l'occurrence d'appel ap_k.

Les ensembles $MODAP_j$ (cf. 4.2.2), $j = 1, \ldots J$ sont alors déterminés par les éléments vrai des lignes du tableau MODAPPELS.

Les modifications implicites - par des occurrences d'appels - de l'ensemble VSM-VSML que nous avons déterminées ci-dessus peuvent avoir pour conséquence de nouvelles modifications implicites par des instructions $bloc_i$ et $boucle_i$

lorsque les fiefs $fd_i \in EFD$ correspondant à ces dernières contiennent des occurrences d'appels.

Il s'agit donc de mettre à jour le tableau MGF (voir le début de ce chapitre) encodant les modifications implicites, par des instructions $bloc_i$ et $boucle_i$, dues initialement aux seules modifications explicites et indirectes.

Nous indiquerons brièvement une façon de procéder :

- soit APB le tableau booléen de dimension card(EFD) x J ; un élément APB(i, est vrai si le fief $fd_i \in EFD$ contient l'occurrence d'appel ap_j et faux sinon
- soit FILTRE le tableau booléen de dimension card(EFD) x card(VSM-VSML) dont l'élément FILTRE(i, j) est faux si le fief $fd_i \in EFD$ est un fief-début et est le fief de déclaration de la variable v_j et vrai sinon.

Le tableau MGF sera alors mis à jour de la façon suivante :

$$MGF := MGF \vee ((APB * MODAPPELS) \wedge FILTRE).$$

Il nous reste à propager les éléments vrai du tableau MGF et de mettre à jour les ensembles $MODB_i$; la façon de procéder est celle décrite pour les tableaux ML_i au début de ce chapitre.

4.3 - Conclusion : une évaluation des informations déduites obtenues -

Nous considérerons ici brièvement l'exactitude des informations déduites obtenues par les procédés décrits en 4.2.1 et 4.2.2.

Nous dirons qu'une information déduite d'un programme-source donné est exacte s'il existe une exécution du programme qui confirme cette information. Par exemple, si le graphe GPA indique que la procédure P1 peut appeler la procédure P2, cette information est exacte s'il existe une exécution du programme dans laquelle P1 appelle effectivement P2.

Nous ne parlerons pas ici du problème des informations relatives aux données à l'entrée d'un programme. Nous avons déjà vu (cf. chapitre 3 à propos de l'attribut RECURSIVE) qu'une information déduite peut être inexacte si l'on ne dispose pas d'informations suffisantes sur les données à l'entrée d'un programme.

Considérons d'abord l'ensemble des informations déduites que représentent le graphe GPA, l'attribut MOD et les ensembles $MODAP_i$. Les processus de la détermination de ces informations déduites sont en effet interdépendants.

Etant donné les considérations et les exemples de 4.2.1 et de 4.2.2 nous formulerons la conjecture suivante : les informations déduites citées ci-dessus sont exactes si les deux propriétés énoncées par la suite sont satisfaites.

Propriété 1 : Si un programme-source contient une procédure P ayant un paramètre ENTRY (cf. 4.2.1) E alors toute activation de la procédure P provoque l'exécution d'une occurrence d'appel ap_k (dans la portée de E) telle que :

- E est un argument de l'appel ap_k ou
- ap_k est un appel formel à E (c'est-à-dire la procédure appelée par ap_k est désignée par E, cf. 4.2.1).

Cette propriété correspond tout simplement à l'exigence raisonnable suivante : la valeur d'un argument ENTRY passé à une procédure du programme doit être utilisée par toute activation de cette procédure.

Nous définirons par la suite l'environnement d'une occurrence d'appel et l'environnement d'une procédure (d'un nom d'une procédure, cf. 4.2.1) :

- soit act_i^j la $j^{ième}$ activation de la procédure correspondant au corps de procédure cp_i ;
- on appellera environnement une liste $env_k = (act_o^{j_o}, act_1^{j_1}, \ldots, act_k^{j_k})$ telle que la liste associée des corps des procédures $(cp_o, cp_1, \ldots, cp_k)$ représente un chemin dans l'arbre ASC (cf. 4.1) tel que cp_o correspond à la racine de ASC.

L'environnement d'une occurrence d'appel ap_i est un environnement env_k dans lequel cp_k est le corps de procédure contenant l'appel ap_i.

L'environnement d'une procédure - notons-la par son corps cp_i - est un environnement env_k dans lequel cp_k est le corps de procédure correspondant au prédécesseur dans ASC de cp_i.

Propriété 2 : Toute procédure P - soit cp_p son corps de procédure - appelée par l'exécution d'une occurrence d'appel formel ap_i dont l'environnement

courant est env_i provoque l'établissement d'un environnement env_p de P qui est tel que :

- $card(env_i) \geq card(env_p)$ et
- la liste env_p est une sous-liste de env_i ou est identique à env_i.

Cette propriété signifie, en particulier, que l'ensemble E_k des procédures qu'il est possible d'appeler relativement aux règles de portée par une occurrence d'appel formel ou non-formel ap_k est tel que $AP_k \subseteq E_k$ (cf. 4.2.1).

L'utilité des informations données par les ensembles $MODB_i$ sera considérée ultérieurement (cf. 8.1.2.2.3).

5 - IMPLEMENTATION DU SPL/1 -

5.1 - Introduction -

Le chapitre 5 décrit une implémentation de SPL/1. Une présentation de l'implémentation s'impose pour deux raisons : d'une part, le texte d'un programme-source dans le langage intermédiaire dépendra de l'implémentation de SPL/1 et, d'autre part, nous montrerons comment on peut améliorer l'implémentation en tenant compte des informations déduites (cf. 4.2) relatives à un programme-source particulier.

Le texte intermédiaire d'un programme-source opérera sur des données simples. Toute manipulation sur des données composées y sera donc traduite en des manipulations sur des données simples. Le texte intermédiaire contiendra également des instructions représentant certaines opérations "administratives" telles initialisation des variables, établissement des listes d'arguments, adressage global, etc.... Par conséquent, pour définir la traduction d'un programme-source (en SPL/1) en un texte intermédiaire il faut spécifier l'implémentation : l'accès aux éléments des tableaux, le passage des arguments dans les appels de procédures, les prologues et les épilogues des blocs-source, etc....

L'implémentation qui sera décrite en 5.3 tiendra compte des informations déduites. Ces informations permettront d'améliorer l'implémentation d'un programme-source particulier.

Certaines informations déduites ont été déjà décrites en 4 (le graphe GPA, l'attribut MOD, les ensembles MODB et MODAP) ; d'autres seront introduites dans ce chapitre. Elles concerneront :

- les utilisations globales des variables ;
- les ensembles des destinations possibles des instructions de saut (GOTO) ;
- la nature des sauts relativement à leurs destinations possibles ;
- la nécessité des allocations dynamiques lors des activations des fiefs de portée, etc....

Les deux sous-chapitres suivants présenteront les originalités de l'implémentation définie (5.2) et décriveront cette implémentation indépendamment d'une machine particulière (5.3). (Nous supposons toutefois que les instructions de la machine à laquelle pourra être appliquée cette implémentation opèrent sur des "mots" (d'une taille en bits donnée), que la machine possède plusieurs

"registres" (des mémoires à accès rapide) et que l'adressage peut être réalisé au moyen d'un "registre de base" et d'un déplacement). Le dernier sous-chapitre (5.4) sera un essai de systématisation de la méthode d'allocation (statique) par recouvrements.

5.2 - Points originaux de l'implémentation définie -

Nous présenterons dans ce chapitre une gestion de l'environnement particularisée réalisée au moyen des vecteurs locaux réduits d'environnement.

Nous présenterons ensuite une implémentation améliorée du passage de certains arguments et finalement des extensions de la méthode d'allocations statiques par recouvrements.

Dans cette présentation certains termes seront introduits pour faciliter la description de l'implémentation.

5.2.1 - Gestion de l'environnement réalisée au moyen des vecteurs locaux réduits d'environnement -

a) - Terminologie -

A un corps de procédure (cf. 4.1) correspondra sa zone locale de procédure (ZL) c'est l'espace à taille statiquement connue alloué statiquement ou dynamiquement (selon l'attribut RECURSIVE de la procédure) à chaque activation de la procédure Une zone locale de procédure ne comprend pas les variables-source STATIC déclarées dans la procédure correspondante (cf. 5.3.1).

A la zone locale d'une procédure on fera correspondre un attribut AD (allocation dynamique) ou AS (allocation statique). C'est l'attribut d'allocatio d'une ZL.

- Le vecteur global d'environnement (VG) est l'ensemble des adresses (déterminabl au chargement) des zones locales des procédures qui ont l'attribut AS.

- Le vecteur local d'environnement d'un corps de procédure $P(VL_p)$ est la liste des adresses des zones locales correspondant aux corps des procédures à l'attribut AD qui sont les prédécesseurs de P dans l'arbre ASC (cf. 4.1). Soit adr_i une adresse de la liste $\ell = (adr_1, \ldots, adr_p)$ et soit cp_i le corps de procédure

dont la zone locale se situe à l'adresse adr_i ; si n_i est le niveau procédural (cf. 4.1) associé au corps cp_i alors l'ordre des adresses de la liste ℓ est celui des entiers n_i croissants ($i = 1, \ldots, p$).

On remarque qu'une adresse d'un VL est nécessaire seulement si la zone locale adressée contient une donnée qui est référencée globalement, c'est-à-dire dans le corps d'une procédure dont le niveau procédural est supérieur à celui de la procédure correspondant à la zone locale en question. Pour savoir s'il est nécessaire de disposer de l'adresse d'une ZL, il faut associer à tout identificateur-source un attribut nous renseignant sur le référençage global (éventuel) de cet identificateur. Les identificateurs-source à considérer sont ceux déclarés explicitement ou implicitement avec l'attribut AUTOMATIC (ceci inclut les identificateurs des étiquettes constantes qui préfixent les instructions du programme y compris les instructions PROCEDURE), ensuite tous les paramètres et tous les identificateurs DEFINED sur des données AUTO (un identificateur déclaré par DCL id_1 DEFINED id_2, doit être considéré comme étant déclaré dans le corps de procédure déclarant id_2). Un tel identificateur ID aura un attribut RG($i, j_1, j_2, \ldots, j_k$), $k \geq 1$, les indices correspondant bijectivement aux corps des procédures du programme, si ID est déclaré dans le $corps_i$, et est référencé au moins une fois dans chacun des corps j_ℓ, $\ell = 1, \ldots, k$ (tout corps j_ℓ étant un successeur dans ASC du $corps_i$).

L'ensemble des corps des procédures qui correspondent aux zones locales dont les adresses devront faire partie de certains vecteurs locaux d'environnement est donc $GL = \{corps_k\}$, $k \in I$ tel que $\forall k \in I$: $corps_k$ est AD, où I est un ensemble d'indices tel que $\forall i \in I \iff \exists$ au moins un identificateur-source ayant l'attribut RG($i, \ldots$).

• On réduira donc les vecteurs locaux et on appellera le vecteur local réduit d'environnement d'une procédure P (VLR) le VL_p auquel on a soustrait les adresses des zones locales correspondant aux corps des procédures non inclus dans l'ensemble GL.

Un VLR_p peut ainsi être vide.

• On appellera environnement d'une procédure P l'adresse de la ZL correspondant au corps de procédure qui est le prédécesseur de P dans ASC.

b) - Adressage -

L'adressage des données dans les zones locales est relatif aux adresses de ces zones locales. L'adresse d'une ZL est soit dans le pointeur courant (PC)- qui est un registre réservé - si la ZL correspond à la dernière activation du corps de la procédure dont l'exécution est en cours, soit dans le vecteur global ou dans le vecteur local réduit de la ZL courante lorsque la ZL adressée ne correspond pas au corps de la procédure en cours d'exécution.

c) - Gestion des vecteurs locaux réduits -

Le vecteur local réduit d'une procédure est alloué dans la ZL de cette procédure. Le VLR s'établit dans le prologue du corps de la procédure P en recopiar partiellement le VLR de la procédure qui "environne" P et en utilisant le pointeur d'environnement (PE) passé à P (PE est un registre réservé et contier l'environnement de P). Le PE est établi lorsqu'un nom de procédure est référencé (en argument ou non) dans une instruction d'appel.

D'autres détails concernant la gestion des vecteurs locaux réduits d'environnement se trouveront dans le chapitre 5.3.4. En comparant cette gestion particularisée d'environnement à celles proposées dans Gries [40] on constate que l'amélioration concerne d'une part l'espace des données (la réduction des vecteurs locaux) et d'autre part le temps d'exécution (la gestion, à l'exécution des VLRs est plus simple que celle des VLs).

La méthode de la gestion d'environnement présentée n'est, bien évidemment, qu'une variante des méthodes classiques.

5.2.2 - Passage des arguments et le retour du résultat d'une procédure-fonctio

En PL/1 on a essentiellement un seul mode de passage des arguments : par référence. Nous distinguerons toutefois (voir 5.3.3) les cas où un argument "dummy" (une traduction possible : bidon) est créé ou non ; ainsi nous parlerons dorénavant de passage par référence (c'est-à-dire sans création de "dummy") et de passage par "dummy". En PL/1 un passage par "dummy" peut être implicite (l'argument doit subir une conversion ou est une expression, etc...) ou explicite (l'argument est mis entre parenthèses). On est censé utiliser un passage par "dummy" explicite lorsque l'argument peut être modifié par la procédure appelée (ou par des procédures subséquemment appelées) et que cette modification est indésirable.

Nous proposons d'introduire un mode de passage appelé passage par valeur pour des arguments simples dont la valeur occupe un ou deux mots ; la valeur est alors passée directement dans la liste des arguments et elle est copiée dans le paramètre correspondant. La valeur d'un tel argument n'est alors plus récupérable par la procédure passant l'argument et l'amélioration consiste dans l'accès plus rapide au paramètre correspondant : on gagne une indirection.

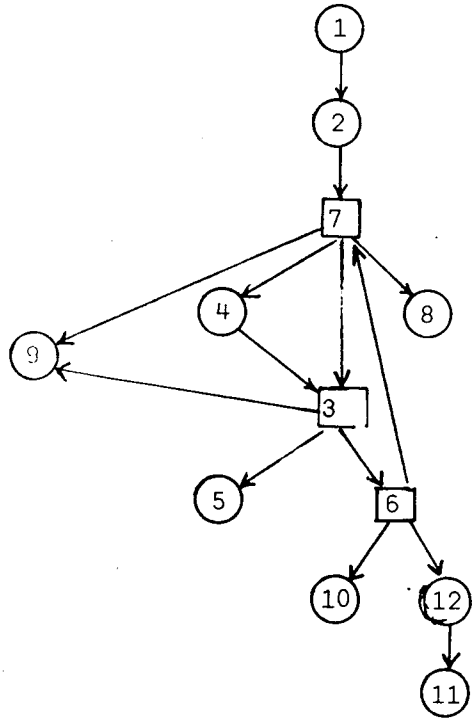
Le passage par valeur est applicable à un argument simple qui est un argument "d'entrée" pour la procédure appelée c'est-à-dire un argument dont la valeur n'est plus récupérable (l'argument est originellement passé par "dummy") ou dont la valeur n'est pas modifiée par la procédure appelée (l'argument étant originellement passé par référence). Les arguments simples passés par "dummy" sont déterminables statiquement ; les arguments simples, non ENTRY, non modifiés par l'appel ont été déterminés en 4.2.2.1 (l'attribut MOD). Dans le chapitre 5.3.3, nous indiquerons les conditions permettant de transformer les passages des arguments "d'entrée" en des passages par valeur. Notons que tous les arguments ENTRY sont passés par valeur.

Le résultat d'une procédure - fonction sera retourné dans l'un des deux registres réservés à cet effet (voir 5.3.6).

5.2.3 - Méthodes d'allocation statique par recouvrements -

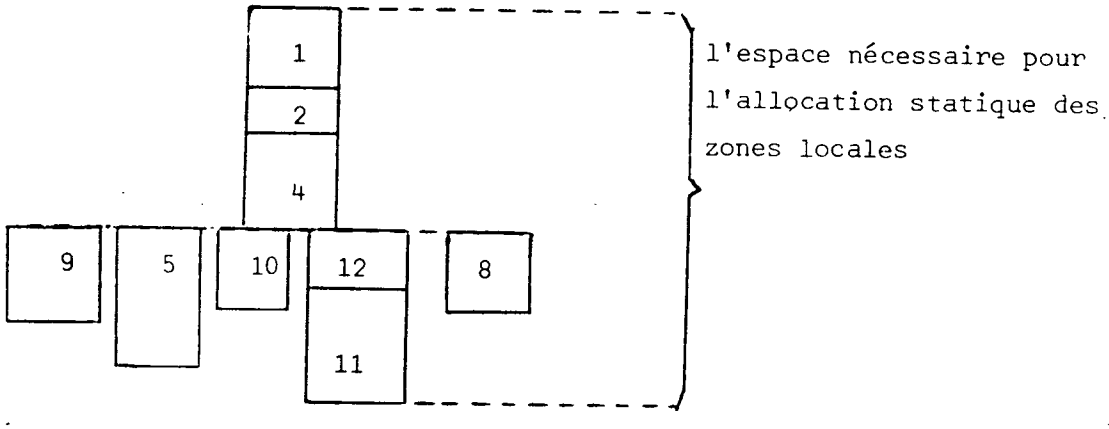
L'allocation statique par recouvrements des données est applicable aux données dont les tailles sont connues à la compilation et se fonde sur un ordre d'utilisation de ces données. L'avantage de tout recouvrement de données est un gain en espace alors que l'inconvénient en est l'impossibilité d'initialisation statique des données.

Exemple : considérons le programme schématisé par la figure 1 de 4.2.1. Le graphe GPA associé (voir la figure 2 de 4.2.1) est reproduit ci-dessous et traduit l'ordre d'utilisation des zones locales des procédures (cf. 5.2.1).



Proposons-nous de trouver une allocation statique par recouvrements des données que sont les zones locales des procédures ayant l'attribut AS (une zone locale de procédure ayant l'attribut AS est représenté par un sommet ◯ dans le GPA et une zone locale de procédure ayant l'attribut AD est représentée par un sommet □) ; les tailles de ces données sont en effet connues à la compilation (cf. 5.3.1).

On pourra constater - en examinant le graphe GPA - que le schéma ci-dessous traduit une allocation statique par recouvrements des zones locales des procédures ayant l'attribut AS (notons - cf. 5.3.1 - que les variables-source STATIC ne font pas partie des zones locales des procédures).



L'allocation statique par recouvrements peut se faire à plusieurs niveaux ; nous considérerons surtout les deux niveaux suivants : celui des zones locales (correspondant aux corps des procédures) et celui des différentes zones définies au niveau d'un fief (cf. 5.4.2). A chaque niveau, l'allocation par recouvrements est définie par un graphe acyclique de recouvrements dont les sommets représentent des données (les espaces contigus qu'elles occupent et leurs tailles). Nous appellerons parfois ce graphe graphe d'allocation. Il s'agira donc de définir, pour les niveaux considérés, des graphes de recouvrement.

Les zones locales à l'attribut AS peuvent être allouées par recouvrements moyennant les informations concernant l'ordre d'activation possible des corps de procédures correspondants. Le graphe qui traduit cet ordre est le graphe GPA défini en 4.2.1. Le graphe de recouvrement des zones locales AS sera obtenu à partir du GPA.

A l'intérieur d'une zone locale nous définirons plusieurs zones de données au niveau d'un fief de portée : zone de fief, zone des temporaires, zone de communs et zone des arguments (cf. 5.4.2). Le graphe de recouvrement de ces zones sera défini tout naturellement au moyen de l'arbre d'inclusion statique des fiefs de portée (cf. 4.1). L'allocation à l'intérieur des zones relatives à un fief de portée ne sera pas davantage détaillée, exception faite de la zone des arguments où on pourra ainsi distinguer un troisième niveau d'allocation par recouvrements.

5.3 - Description de l'implémentation -

5.3.1 - Allocations statiques et dynamiques -

Allocation statique

Les données dont la taille est statiquement connue sont allouées statiquement. Ces données sont soit des variables STATIC déclarées dans le programme-source soit des zones définies par l'implémentation. Nous distinguerons ainsi deux zones statiques :

a) - La zone statique 1 (ZS1) comprendra les zones locales à l'attribut d'allocation AS qui y seront allouées par recouvrement.

b) - La zone statique 2 (ZS2) comprendra :

- les variables-source STATIC
- le vecteur global d'environnement
- certains vecteurs d'accès (voir 5.3.8) constants
- des constantes et des valeurs initiales.

L'allocation dans cette zone sera sérielle.

Allocation dynamique

L'allocation dynamique sera réalisée au moyen d'une pile.

Cette allocation concernera :

- les zones locales des procédures ayant l'attribut d'allocation AD ;
- les données dont les tailles ne sont pas prévisibles statiquement :
  - les tableaux AUTO à taille dynamique
  - les arguments passés par "dummy" (cf. 5.3.3) à taille dynamique (c'est-à-dire variable).

La pile sera gérée au moyen d'un pointeur de pile PP.

L'idée principale de la méthode d'allocation esquissée est d'allouer statiquement si c'est possible : ceci permet d'espérer un gain en temps d'exécution du programme, l'inconvénient étant une perte possible de l'espace (voir toutefois 5.4.1).

On considérera que toutes les données élémentaires occupent un mot-mémoire à l'exception des bornes (un demi-mot), des étiquettes variables et des arguments ou paramètres ENTRY (deux mots).

5.3.2 - Terminologie et informations déduites relatives aux corps des procédures et aux fiefs de portée -

5.3.2.1 - Corps des procédures -

Nous donnerons par la suite une liste d'informations déduites relatives à un corps de procédure cp_i :

- les attributs AD et AS ; ils correspondent aux attributs d'allocation associés à la zone ZL_i (cf. 5.2.1) ;

- l'attribut indiquant la nécessité d'un vecteur local réduit d'environnement à allouer dans la zone locale correspondant au cp_i : VLE $\equiv$ $VLR_i \neq \emptyset$. Si le cp_i est VLE on définit :

  - l'entier indiquant le déplacement par rapport à VLR_i de l'élément devant recevoir la valeur du pointeur d'environnement à l'activation du cp_i : $DEPL_i$. Si cette information ($DEPL_i$) est absente, le PE n'est pas copié dans VLR_i ;
  - l'entier indiquant le nombre d'éléments (contigus) à copier dans le VLR_i du VLR_j où le cp_j est le prédécesseur dans ASC (cf. 4.1) du cp_i : $COPIE_i$;
  - l'entier indiquant le niveau procédural (cf. 4.1) du cp_i : NP_i ;
  - l'entier NNF_i indiquant le nombre de niveaux de l'arbre $ASFP_i$ (cf. 4.1). Si les fiefs de portée qui sont les sommets de $ASFP_i$ sont numérotés de 1 à N alors :

$$NNF_i = \max_{j \in [1, N]} (NF_j) + 1 \qquad \text{(cf. 4.1 pour } NF_j\text{)}$$

Dans la ZL_i du cp_i on allouera NNF_i pointeurs. Chacun des pointeurs indiquera pour le (les) fief(s) de portée correspondant(s) le sommet de la pile d'allocation. Cet ensemble de pointeurs alloués dans la ZL_i est appelé la zone de pointeurs ZP_i (de longueur NNF_i).

Un pointeur p_j de la ZP_i associé au $fief_j$ est mis à jour (en même temps que le pointeur de pile PP) :

- lors des empilages de données à taille dynamique déclarées dans le $fief_j$;
- après l'allocation éventuelle de la ZL_i si le $fief_j$ est le fief-procédure du cp_i. Le p_j n'est pas mis à jour lors des empilages de certains arguments relatifs aux appels faits dans le $fief_j$ (cf. 5.4.2.2).

Notons que même s'il n'y a aucune allocation en pile pour le compte du $fief_j$, le p_j est mis à jour : il reçoit la valeur du p_{j-1} qui correspond au fief-prédécesseur dans $ASFP_i$ du $fief_j$ (voir aussi 5.3.4).

Les pointeurs de la ZP sont utilisés dans l'implémentation des sauts (cf. 5.3.7) et par les épilogues des fiefs de portée.

5.3.2.2 - Fiefs de portée -

Note : le terme fief utilisé par la suite (et jusqu'à la fin du chapitre 5) aura la signification du terme fief de portée (il désignera donc un fief-procédure ou un fief-début, cf. 4.1).

A tout fief $fief_j$ est associée l'information déduite que représente l'attribut DYN.

La signification de l'attribut DYN est la suivante : si le $fief_j$ a l'attribut DYN alors une exécution du $fief_j$ peut entraîner des allocations dynamiques.

Ces allocations dynamiques peuvent provenir :

- du fait que le $fief_j$ contient des déclarations des données ayant des tailles dynamiques ;
- du fait que le $fief_j$ est un fief-procédure d'un corps de procédure ayant l'attribut AD.

Notons que les allocations dynamiques éventuelles dues aux instructions d'appel du $fief_j$ ne sont pas considérées.

Si le $fief_j$ a l'attribut DYN le pointeur p_j d'une ZP associé au $fief_j$ est modifié par l'exécution du $fief_j$.

5.3.3 - Implémentation des instructions CALL (c'est-à-dire des appels des procédures-sous-programmes) et des appels des procédures-fonctions -

A toute occurrence d'appel, notée ap_i on associera :

- l'ensemble des corps des procédures appelables $AP_i = \{cp_k\}$.
  Un ensemble AP_i (cf. 4.2.1) sera également associé à toute occurrence d'appel non-formel (il aura alors un seul élément) et à tout paramètre ENTRY (l'ensemble AP_j associé au paramètre par_j est alors déterminé par la colonne j de la matrice MP2, cf. 4.2.1).
- le corps de procédure contenant l'occurrence d'appel ap_i : cp_i ;
- lorsque AP_i contiendra un seul élément cp_k, on dénotera le prédécesseur de cp_k dans ASC par cp_e.

L'appel construit la liste des arguments et "passe" à la procédure appelée le pointeur de la liste des arguments PA (qui est un registre réservé) et éventuellement le pointeur d'environnement PE.

En ce qui concerne le passage des arguments nous distinguerons trois modes de passage : par référence (l'adresse de l'argument lui-même est passée dans la liste des arguments ou, dans le cas de tableau, l'adresse du vecteur d'accès est passée), par "dummy" (un emplacement pour la valeur de l'argument est alloué et l'adresse de cet emplacement est passée dans la liste des arguments) et par valeur. Dans ce dernier mode, la valeur de l'argument occupe un ou deux mots et est passée directement dans la liste des arguments. Ce mode de passage s'appliquera toujours aux arguments ENTRY (la valeur occupe deux mots) et aussi à d'autres arguments simples si certaines conditions, qui sont décrites par la suite, sont remplies.

Pour un programme-source définissons :

- $G_1^n = (AP_1^n, CP_1^n, U_1^n)$: le graphe biparti sans orientation où l'ensemble AP_1^n est l'ensemble des occurrences d'appels des procédures-sousprogrammes à n arguments, l'ensemble CP_1^n est l'ensemble des corps des procédures-sous-programmes à n paramètres et U_1^n est l'ensemble des arêtes telles qu'un sommet $ap_i \in A_1^n$ est jointif à un sommet $cp_j \in CP_1^n$ si et seulement si $cp_j \in AP_i$ (c'est-à-dire le cp_j est appelable par l'appel ap_i) ;
- $G_2^m = (AP_2^m, CP_2^m, U_2^m)$: le graphe biparti défini comme le G_1^n en considérant les procédures -fonctions du programme-source ;
- $F_1 = \{G_1^n\}_{n \in N}$, $F_2 = \{G_2^m\}_{m \in M}$; N et M dépendant du programme-source donné ;
- $G^k = (AP^k, CP^k, U^k)$ dénote une composante connexe quelconque d'un graphe biparti G_i^k d'une famille F_i, i = 1, 2 ;
- CC : l'ensemble total des composantes connexes des graphes bipartis des deux familles.

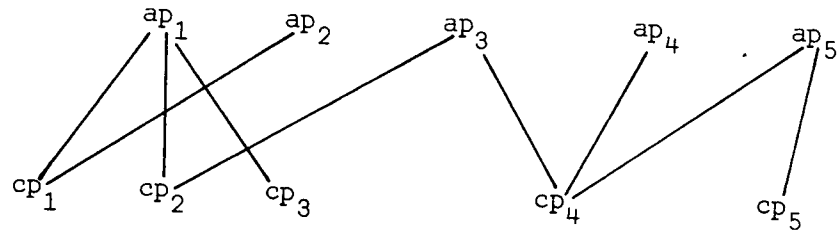
Pour que, étant donné une composante connexe $G^k \in CC$, le mode de passage par valeur puisse s'appliquer à l'ensemble des paramètres en position $j \leq k$ des corps des procédures CP^k et à l'ensemble des arguments en position j des appels AP^k les conditions suivantes doivent être satisfaites :

1) - $\forall cp_i \in CP^k$: le $paramètre_{i,j}$ (i dénotant la procédure) est simple,

2) - $\forall ap_i \in AP^k$: l'$argument_{i,j}$ (i dénotant l'appel) doit être scalaire et doit être originellement passé par "dummy" ou avoir l'attribut VAL (cf. 4.2.2.1).

Après avoir considéré l'applicabilité de la transformation de certains passages des arguments par "dummy" ou par référence en des passages par valeur on déduira l'information concernant les modes de passage : à chaque position d'argument de toute occurrence d'appel (c'est-à-dire à chaque occurrence d'argument) on associera l'information déduite indiquant l'un des trois modes de passage de l'argument correspondant ; de même à chaque paramètre de chacun des corps des procédures on associera l'information déduite indiquant le passage (ou plutôt la réception) par valeur ou non.

Exemple : Soit G^k la composante connexe suivante :



Dans ce graphe biparti les appels ap_1, ap_3 et ap_5 sont certainement formels. Pour que, au paramètre$_{2,\,j}$ simple par exemple, puisse s'appliquer le passage par valeur il ne suffit pas d'avoir un passage originel par "dummy" pour les arguments $arg_{1,\,j}$ et $arg_{3,\,j}$; il faut en plus que les deux conditions ci-dessus, concernant G^k soient satisfaites.

Nous reprendrons par la suite l'exemple donné par la figure 1 du sous-chapitre 4.2.1. La figure 1 représente le schéma d'un programme-source ; toutes les procédures de ce programme-source sont des procédures-sous-programmes. Il s'ensuit que la famille F_2 est vide. Par contre nous avons :

$$F_1 = \{G_1^1, G_1^2\}$$

et, avec des notations évidentes :

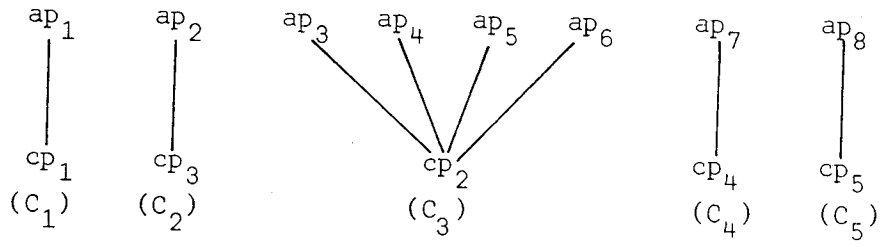
$$G_1^1 : (AP_1^1, CP_1^1, U_1^1), \text{ avec :}$$

AP_1^1 = {ap_1 : CALL P2(P7),
ap_2 : L2,
ap_3 : CALL P3(E3),
ap_4 : L5,
ap_5 : L6,
ap_6 : L7,
ap_7 : L10
ap_8 : CALL P12(E5)} ;

$CP_1^1 = \{cp_1 : P2, cp_2 : P3, cp_3 : P4, cp_4 : P10, cp_5 : P12\}$;

$U_1^1 = \{[ap_1, cp_1], [ap_2, cp_3], [ap_3, cp_2], [ap_4, cp_2], [ap_5, cp_2], [ap_6, cp_2], [ap_7, cp_4], [ap_8, cp_5]\}$.

Le graphe G_1^1, avec ses composantes connexes, est donné ci-dessous :



La seule composante à prendre en considération est C_4 ; pour que l'on puisse passer l'argument AR1 de l'appel L10 par valeur il faut que le paramètre A3 de la procédure P10 et l'argument AR1 soient scalaires et que AR1 ait l'attribut VAL (il n'est pas passé par "dummy").

Considérons maintenant le graphe G_1^2 :

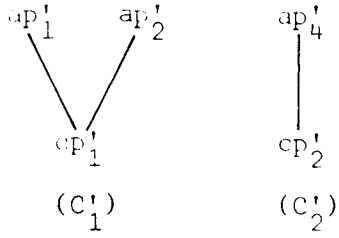
$G_1^2 = (AP_1^2, CP_1^2, U_1^2)$, avec

$AP_1^2 = \{ap_1' : L1,$
$ap_2' : L9,$
$ap_3' : L11,$
$ap_4' : L12\}$;

$CP_1^2 = \{cp_1' : P7, cp_2' : P11\}$;

$U_1^2 = \{[ap_1', cp_1'], [ap_2', cp_1'], [ap_4', cp_2']\}$.

Le graphe G_1^2 est donné ci-dessous :



La seule composante connexe à prendre en considération est C'_2 : les deux conditions du passage par valeur doivent être examinées pour chacun des deux couples paramètre-argument suivants : (A1, AR3) et (A2, AR4).

En ce qui concerne le passage du pointeur d'environnement PE on distingue les cas suivants :

a) - L'appel ap_i dans cp_i est non-formel. On distinguera les cas suivants (avec les notations du début de ce chapitre) :

Cas 1 : Le cp_e est AS : le PE n'est pas passé ; dans le prologue du cp_k l'environnement, si nécessaire, sera pris dans le VG.

Cas 2 : Le cp_e est AD et $NP_K = NP_i + 1$ et le cp_k est VLE ($VLR_k \neq 0$, cf. 5.3.2.1) : le PE est passé et sa valeur est celle du PC.

Cas 3 : Le cp_e est AD et $NP_k \neq NP_i + 1$ et le cp_k est VLE : le PE est passé et sa valeur est prise dans le VLR_i.

Cas 4 : Le cp_e est AD et le cp_k n'est pas VLE : le PE n'est pas passé.

Si, dans les cas 2 et 3 le $cp_e \notin GL$ (cf. 5.2.1) la valeur du PE passé ne fera pas partie du VLR_k.

b) - L'appel ap_i dans cp_i est formel. On distinguera :

Cas 1 : $\exists\ cp_k \in AP_i$ qui est VLE et tel que son prédécesseur cp_e dans ASC est AD : le PE est passé et sa valeur est prise dans le paramètre ENTRY qui désigne la procédure appelée (un paramètre ENTRY - ainsi qu'un argument ENTRY - est implémenté en deux mots ; le premier contient l'adresse-code d'un corps de procédure et le deuxième l'environnement de ce même corps de procédure).

Cas 2 : La condition du cas 1) ci-dessus n'est pas vérifiée : le PE n'est pas passé.

c) - Lors du passage-par valeur-d'un argument de type ENTRY on procède d'une façon analogue :

- si l'argument est une constante ENTRY (c'est-à-dire un nom de procédure) on considère le corps de procédure cp_k associé à la constante ENTRY et on applique le cas a) où l'on remplace le passage du PE par la mise à jour du mot-environnement de l'argument ;

- si l'argument est un paramètre ENTRY on considère l'ensemble AP_i associé au paramètre et on applique le cas b) où l'on remplace le passage du PE par la copie du mot-environnement du paramètre dans le mot-environnement de l'argument.

Dans cette implémentation le mot-environnement d'un argument dans une liste d'arguments peut avoir une valeur indéfinie. Lors d'un appel formel (voir le le cas 1 du b)) cette valeur indéfinie peut même être passée au PE. La solution implémentée n'est donc pas optimale et elle ne peut l'être, dans le cas général.

5.3.4 - Prologues des corps des procédures - Un exemple de gestion des vecteurs locaux réduits d'environnement

La partie initiale de toute zone locale ZL_i d'un corps de procédure comprend :

- un mot devant contenir la valeur du pointeur p_k (d'une ZP) associé au $fief_k$ activant le cp_i : $PPILE_i$. Cette valeur-qui peut différer de la valeur du PP à l'entrée du cp_i - est utilisée lors de la désactivation du cp_i (voir 5.3.6) ;
- un pointeur vers la ZL du corps de procédure qui précède cp_i dynamiquement. C'est le chaînage en arrière CHA_i. Ce chaînage en arrière dynamique des zones locales est utilisé pour les retours des corps des procédures ;
- la zone de sauvegarde des registres REGS ;
- éventuellement le vecteur local réduit d'environnement VLR_i ;
- la zone des pointeurs ZP_i (cf. 5.3.2) ;
- la zone des paramètres $ZPAR_i$ de la procédure correspondant au cp_i.

Le prologue d'un cp_i fait partie du fief-procédure correspondant au cp_i. Il effectue les tâches suivantes :

- il sauvegarde les registres dans la zone REGS de la ZL du cp_j qui vient d'activer le cp_i ;
- si le cp_i est AD il alloue la ZL_i et met à jour le PP ainsi que le premier pointeur de la ZP_i (le pointeur associé au fief-procédure du cp_i) si le fief-procédure ne contient pas de déclarations des données à taille dynamique ;

- il met à jour le pointeur courant PC ;
- il affecte au pointeur CHA_i la valeur du PC à l'entrée du prologue ;
- il affecte au mot $PPILE_i$ la valeur du pointeur p_k (de la ZP_j) associé au $fief_k$ du cp_j qui vient d'activer le cp_i. Si le cp_i est AS et son fief-procédure est ¬DYN la valeur de $PPILE_i$ est copiée dans le premier pointeur de la ZP_i ;
- il met à jour le VLR_i si le cp_i est VLE ;
- il copie la liste d'arguments pointée par PA dans $ZPAR_i$.

Pour établir le VLR_i on utilise les informations déduites relatives au corps de procédure cp_i (cf. 5.3.2.1). Ces informations sont reprises ici, en dénotant par cp_e le prédécesseur dans ASC du cp_i :

- $COPIE_i$ est le nombre de corps des procédures qui appartiennent à GL (cf. 5.2.1), qui sont du niveau $\leq NP_i-2$ et qui sont les prédécesseurs dans l'arbre ASC du corps cp_i. C'est aussi le nombre d'éléments du VLR_e ;
- si $COPIE_i \neq 0$ le VLR_e est copié dans les $COPIE_i$ premiers éléments du VLR_i. L'adresse du VLR_e est déduite soit du PE reçu, soit de l'environnement du cp_i qui est dans le VG ;
  - l'absence du $DEPL_i$ signifie : le $cp_e \notin GL$; le VLR_i n'a pas d'élément correspondant au niveau NP_e ; le PE reçu n'est pas copié dans le VLR_i ; le nombre d'éléments du VLR_i est égal à $COPIE_i$;
  - la présence du $DEPL_i$ signifie : le $cp_e \in GL$; la valeur du PE reçu est copiée dans le dernier élément du VLR_i dont le $DEPL_i$ indique le déplacement relatif à l'adresse du VLR_i ; $DEPL_i$ est égal à $COPIE_i*4$ (tout élément du VLR_i occupe un mot) ; le nombre d'éléments du VLR_i est égal à $COPIE_i+1$.

Un exemple montrant la gestion des VLR est donné ci-dessous et illustré dans les figures 1, 2 et 3.

Soit le schéma d'un programme-source présenté sous forme de l'arbre ASC :

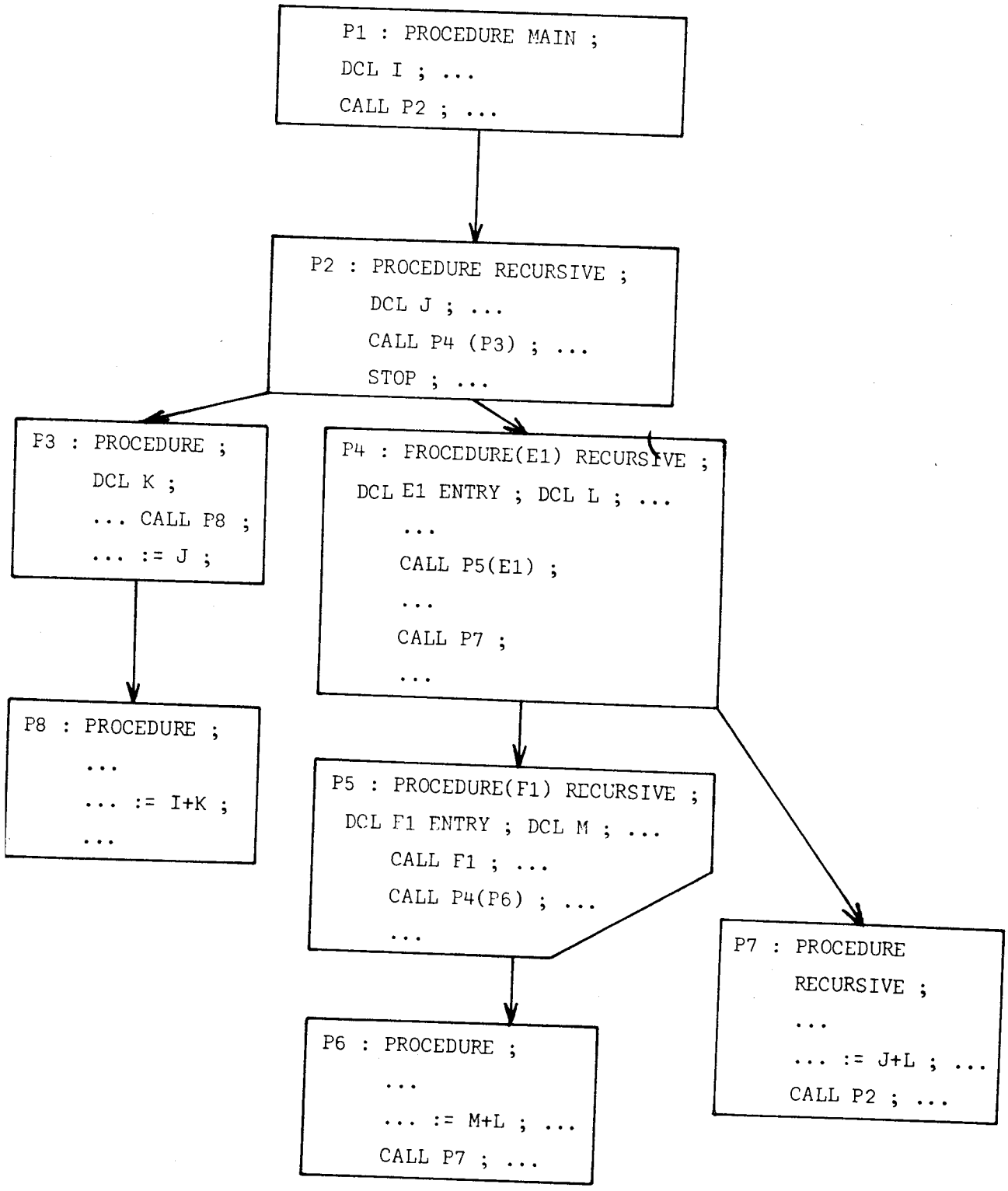


Figure 1.

- les corps cp_2, cp_4, cp_5 et cp_7 sont AD, les autres corps sont AS ;
- cp_3 référence J de cp_2 ;
- cp_8 référence I de cp_1 et K de cp_3 ;
- cp_5 référence P_4 de cp_2 ;
- cp_7 référence J de cp_2, L de cp_4 et P2 de cp_1 ;
- cp_6 référence M de cp_5, L de cp_4 et P7 de cp_4.

L'ensemble des corps AD déclarant les identificateurs globalement référencés est donc GL = $\{cp_2, cp_4, cp_5\}$.

Le VG est la liste $(adr(ZL_1), adr(ZL_3), adr(ZL_8), adr(ZL_6))$.

Les VLR_i sont comme suit :

- VLR_1, VLR_2 sont $\emptyset$; VLR_8 est $\emptyset$;
- $VLR_3 = (adr(ZL_2))$ avec $COPIE_3 = 0$, $DEPL_3$ (en mots) = 0 ;
- $VLR_4 = (adr(ZL_2))$ avec $COPIE_4 = 0$, $DEPL_4 = 0$;
- $VLR_5 = (adr(ZL_2), adr(ZL_4))$ avec $COPIE_5 = 1$, $DEPL_5 = 1$;
- $VLR_6 = (adr(ZL_2), adr(ZL_4), adr(ZL_5))$ avec $COPIE_6 = 2$, $DEPL_6 = 2$;
- $VLR_7 = (adr(ZL_2), adr(ZL_4))$, avec $COPIE_7 = 1$, $DEPL_7 = 1$.

Le GPA déduit :

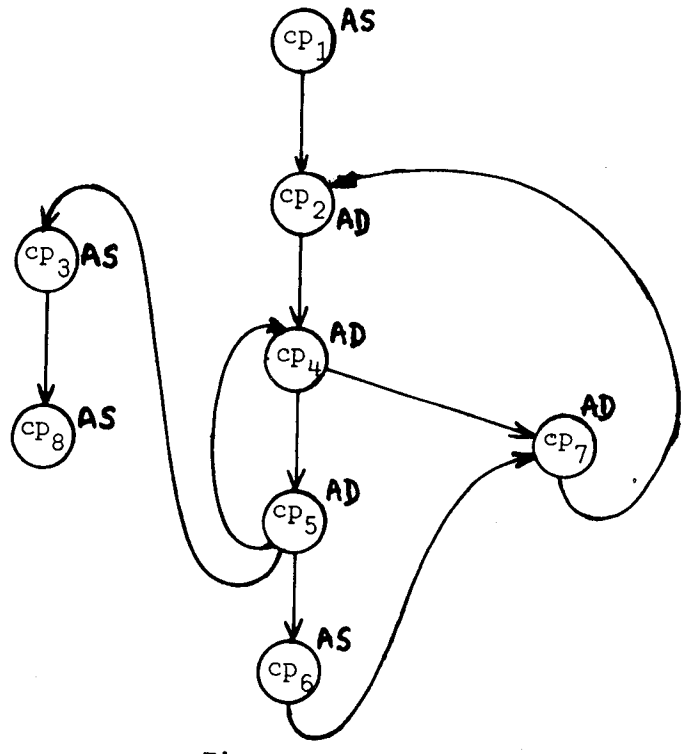


Figure 2.

On note que le GPA fait apparaître le cp_6 comme étant potentiellement récursif. Ce n'est que par l'absence de l'attribut-source RECURSIVE dans l'instruction P_6 : PROC ; qu'on peut déduire que cette potentialité n'est qu'apparente.

Montrons maintenant, en considérant une séquence d'activations des corps des procédures possible d'après le GPA la construction des VLR_i. (Notons que l'attribut AS du corps cp_6 indique que ce corps ne peut apparaître qu'une fois dans un "chemin activant" donné de la figure 3).

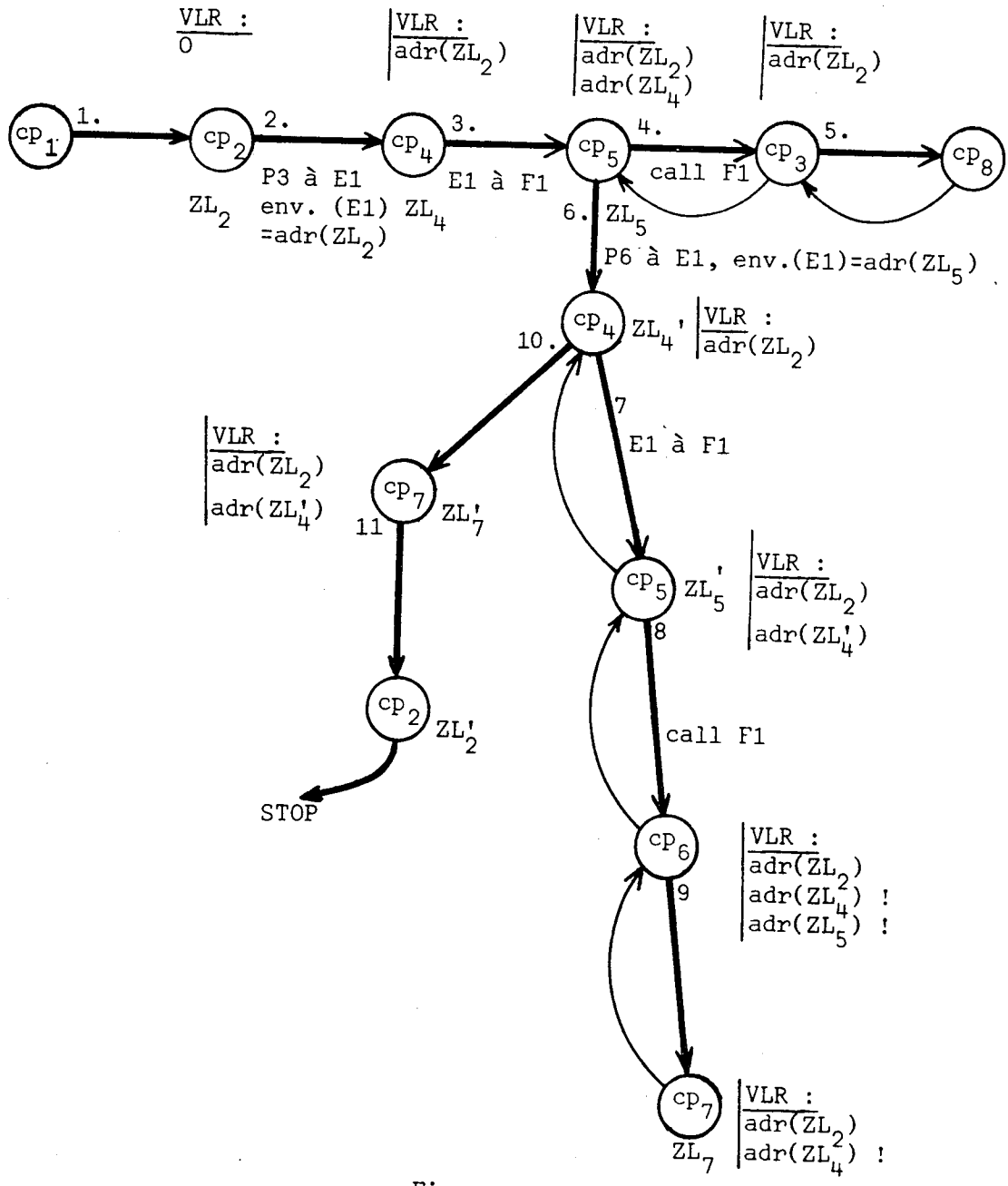


Figure 3.

Les arcs en traits gras indiquent les "activations" (les appels) et les arcs en traits fins indiquent les "désactivations" (les retours).

5.3.5 - Prologues des fiefs -

Un fief-procédure commence par un prologue de fief qui est constitué du prologue du corps de procédure associé (voir 5.3.4) et, éventuellement, des instructions réalisant des initialisations et des allocations en pile pour les données à taille dynamique locales au fief-procédure.

Un fief-début commence par un prologue de fief constitué, éventuellement, des instructions réalisant des initialisations et des allocations en pile pour les données à taille dynamique locales au fief-début.

Tout prologue d'un $fief_j$ met à jour le pointeur p_j de la ZP_i correspondant au $fief_j$. On distingue les cas suivants (en notant par cp_i le corps de procédure correspondant au $fief_j$) :

- si le $fief_j$ est un fief-procédure ne contenant pas de déclarations de données à taille dynamique, le p_j est mis à jour par le prologue du corps de procédure cp_i (voir 5.3.4) ;
- si le $fief_j$ est un fief-procédure ou un fief-début contenant des déclarations de données à taille dynamique le prologue du $fief_j$ se termine par l'affectation de la valeur courante du PP (c'est-à-dire la valeur du PP après l'allocation de toutes les données à taille dynamique locales au $fief_j$) au pointeur p_j ;
- si le $fief_j$ est un fief-début ne contenant pas de déclarations des données ayant des tailles dynamiques le p_j reçoit la valeur du p_{j-1} correspondant au fief qui est le prédécesseur du $fief_j$ dans l'arbre $ASFP_i$.

Nous remarquons que, pour un fief-début $fief_j$ qui n'est pas DYN la valeur du p_j peut ne pas être utilisée pour l'épilogage (voir 5.3.6 et 5.3.7) ; toutefois cette valeur peut être utilisée pour la mise à jour d'un p_{j+1} correspondant au fief qui est le successeur du $fief_j$ dans l'arbre $ASFP_i$. Nous n'avons pas examiné les cas ou la valeur d'un p_j n'est jamais utilisée (rendant ainsi superflue la mise à jour du p_j).

5.3.6 - Epilogues -

1) - Les définitions des attributs déduits -

- A un cp_i on associera l'un des trois attributs d'épilogage mutuellement exclusifs définis comme suit :
  - EP_1 : la procédure correspondant au cp_i n'est pas déclarée MAIN et est AD ou cette procédure n'est pas déclarée MAIN et est AS et le fief-procédure du cp_i est DYN ;
  - EP2 : la procédure correspondant au cp_i est déclarée MAIN ;
  - EP3 : la procédure correspondant au cp_i n'est pas déclarée MAIN et le cp_i est AS et son fief-procédure n'est pas DYN.
- A un cp_i on associera aussi l'un des attributs de la valeur-résultat retournée mutuellement exclusifs. Ces attributs sont déduits de l'attribut source RETURNS (ou par défaut) de l'instruction PROCEDURE correspondant au cp_i :
  - NIL : pas de valeur-résultat retournée, la procédure n'étant pas une fonction ;
  - FIXE : la valeur du résultat est un entier (en virgule fixe) ;
  - FLOT : la valeur du résultat est en virgule flottante.
- A toute occurrence $return_j$ d'une instruction RETURN on associera l'un de deux attributs d'épilogage mutuellement exclusifs. Considérons l'occurrence $return_j$ contenue dans le $fief_j$ du corps de procédure cp_i. Les deux attributs qu'on peut associer au $return_j$ sont :
  - RET1 : le $fief_j$ n'est pas DYN et aucun des prédécesseurs (éventuels) du $fief_j$ dans le $ASFP_i$ n'est DYN ;
  - RET2 : $\equiv \neg$RET 1.

A l'occurrence $return_j$ sera également associé l'un des trois attributs NIL, FIXE et FLOT, attribut associé au cp_i.

2) - Les actions en épilogue et le retour du résultat -

L'implémentation des instructions-source END, STOP et RETURN (expression) est considérée. Le retour du résultat et les actions en épilogue (désallocations, rétablissement de l'environnement) sont implémentés en ligne ("in-line") par un souci de simplification. Une telle implémentation réduit le temps d'exécution

mais rallonge l'espace-code. En pratique les désallocations seront probablemen implémentées par des appels des routines de librairie adéquates.

a) - Désactivation d'un fief-début (l'instruction END correspondant à une instruction BEGIN).

Cas 1) : Le $fief_i$ désactivé est DYN : en utilisant le pointeur de la zone des pointeurs (voir 5.3.2.1) associé au $fief_j$ qui est le prédécesseur dans l'arbre ASFP du $fief_i$ on désempile et on remet à jour PP.

Cas 2) : Le fief désactivé n'est pas DYN : aucune action.

Si l'instruction END définit une fermeture multiple on suit le cas 1 si l'un au moins des fiefs desactivés est DYN, sinon on suit le cas 2. Inversement, une séquence des instructions END du programme source sera supposée correctement "réduite".

b) -Désactivation d'un fief-procédure et du corps de procédure associé lorsque la procédure correspondante est une procédure-sousprogramme (l'instruction END correspondant à une instruction PROCEDURE).

Soit cp_i le corps de procédure (qui est NIL) désactivé.

Cas 1) : Le cp_i est EP1 : en utilisant PPILE (voir 5.3.4)désempiler et remettre à jour le PP ; en utilisant le chaînage en arrière CHA_i (voir 5.3.4) remettre à jour le PC, restaurer les registres et retourner (pour l'adresse de retour on utilisera un registre réservé AR).

Cas 2) : Le cp_i est EP2 : terminer l'exécution.

Cas 3) : Le cp_i est EP3 : voir le cas 1) sans désempilage.

Si l'instruction END définit une fermeture multiple telle que l'un au moins des fiefs-début désactivés est DYN et que le cas 3 ci-dessus s'applique on procède comme au cas 1, sinon le cas 1 ou le cas 2 ci-dessus s'applique normalement.

Inversement, une séquence des instructions END dont la dernière correspond à une instruction PROCEDURE sera supposée correctement "réduite".

c) - Désactivation d'un fief-procédure et du corps de procédure associé lorsque la procédure correspondante est une procédure -fonction (l'instruction END correspondant à une instruction PROCEDURE).

Toute procédure-fonction est désactivée par une instruction RETURN (expression), par conséquent l'instruction END correspondante n'est jamais exécutée et n'entraîne aucune action.

d) - Désactivation d'un corps de procédure lorsque la procédure correspondante est une procédure-sousprogramme (l'instruction RETURN).

Soit $return_j$ (il possède l'attribut NIL) une occurrence de l'instruction RETURN contenue dans le corps cp_i.

Cas 1) : Si $return_j$ est RET1 on procède comme au b) cas 3.

Cas 2) : Si $return_j$ est RET2 on procède comme au b) cas 1.

e) - Désactivation d'un corps de procédure lorsque la procédure correspondante est une procédure-fonction (l'instruction RETURN(expression)).

Soit $return_j$ (FIXE ou FLOT) l'occurrence de l'instruction RETURN considérée.

La valeur-résultat est établie et retournée dans le registre réservé RES1 si le $return_j$ est fixe et dans le registre réservé RES2 (flottant) si le $return_j$ est FLOT.
Lors de la restauration des registres on doit donc garder (c'est-à-dire ne pas restaurer) soit RES1 soit RES2.
En ce qui concerne le rétablissement de l'environnement on distingue :

Cas 1) : Le $return_j$ du corps cp_i est RET1 : on procède comme au b) cas 3 en tenant compte de la remarque ci-dessus concernant la restauration des registres.

Cas 2) : Le $return_j$ du corps cp_i est RET2 : on procède comme au b) cas 1 en tenant compte de la remarque concernant la restauration des registres.

5.3.7 - Sauts (les instructions GOTO) -

1) - Les définitions des attributs déduits -

A toute occurrence $goto_j$ d'une instruction GOTO contenue (l'occurrence) dans le $fief_j$ d'un corps de procédure cp_i on associera l'un des deux attributs GO1 et GO2 définis par la suite.

Cas 1) : Le $goto_j$ est un saut à une étiquette constante $etiq_k$ déclarée implicitement dans le $fief_k$ du cp_i. (Voir les restrictions exposées en 3). Soit : ch = ($fief_k$, $fief_{k+1}$, ..., $fief_j$) le chemin dans $ASFP_i$ allant du sommet $fief_k$ au sommet $fief_j$. Le $goto_j$ aura l'attribut GO1 si aucun des fiefs du sous-chemin ($fief_{k+1}$, ..., $fief_j$) de ch n'est DYN ou si k = j sinon le $goto_j$ sera GO2.

Cas 2) : Le $goto_j$ est un saut à une étiquette variable déclarée dans un fief du cp_i avec la liste des étiquettes constantes ℓ = ($etiq_1$, ..., $etiq_n$). Si pour $\forall$ k = 1, ..., n le $goto_j$, considéré comme le saut à l'étiquette constante $etiq_k$ de ℓ, est d'attribut GO1 alors le $goto_j$ recevra effectivement l'attribut GO1 sinon le $goto_j$ aura l'attribut GO2.

La signification des attributs GO1 et GO2 est la suivante : une occurrence $goto_j$ qui est GO1 n'entraîne pas de désempilage alors que l'attribut GO2 spécifie la nécessité de désempilage précédant le saut effectif.

2) - <u>L'implémentation des étiquettes constantes et variables et des affectations d'étiquettes</u> -

Une étiquette constante est implémentée en un mot dont la valeur (constante) est une adresse-code du programme objet.

Une étiquette variable est implémentée en deux mots, le premier étant une adresse-code et le deuxième étant la valeur du pointeur p_j (de la zone des pointeurs ZP) associé au $fief_j$ déclarant l'étiquette constante qui est la valeur courante de l'étiquette variable.
Lors de l'affectation d'une étiquette constante à une étiquette variable l'adresse du pointeur correct à affecter au deuxième mot est connue statiquement.

3) - <u>L'implémentation d'une occurrence $goto_j$</u> -

Cas 1) : Le $goto_j$ est GO1 : on effectue le branchement à l'adresse-code qui est la valeur de l'étiquette constante référencée par le $goto_j$ ou qui est la valeur du premier mot de l'étiquette variable référencée par le $goto_j$.

Cas 2) : Le $goto_j$ est GO2 : si l'étiquette référencée par le $goto_j$ est une constante déclarée implicitement dans un $fief_k$ du cp_i on accède au pointeur de la ZP_i, pointeur qui correspond au $fief_k$ sinon le pointeur se trouve dans le deuxième mot de l'étiquette variable. La valeur du pointeur définit le nouveau PP ; il y a désempilage.
On procède ensuite au branchement comme au cas 1).

4) - Le Test de localité du saut -

Nous avons vu dans le chapitre 3 décrivant le sous-ensemble SPL/1 que les restrictions introduites relatives aux étiquettes variables ne permettent pas de déduire statiquement - c'est-à-dire sans une interprétation partielle du programme-source - si un saut à une étiquette variable STATIC, contenu dans un corps de procédure cp_i ayant l'attribut AD ne désactive pas l'activation courante du cp_i et ne retourne ainsi à une activation précédente du même cp_i. Donc, pour toute occurrence $goto_j$ - qui est un saut à une étiquette variable - contenue dans un corps de procédure ayant l'attribut AD il faudrait, pour qu'on ait un programme-objet sûr, tester que le saut est bien local à l'activation courante du cp_i. Ceci peut se faire en comparant la valeur de $PPILE_i$ courante (cf. 5.3.4) à la valeur du deuxième mot de l'étiquette variable référencée par le $goto_j$ en question. En effet, le cp_i étant AD la valeur de $PPILE_i$ doit être toujours inférieure à la valeur du deuxième mot de l'étiquette variable ; cette dernière valeur étant la valeur du pointeur p_j associé au $fief_j$ (du cp_i) déclarant l'étiquette constante qui est la valeur courante de l'étiquette variable.

5.3.8 - Tableaux -

Nous suivrons généralement l'implémentation décrite dans la brochure IBM [61]. Nous rappellerons dans ce chapitre les formules d'accès aux éléments des tableaux et nous décrirons l'allocation des vecteurs d'accès.

1) - Vecteurs d'accès ("dope vectors") -

Soit une déclaration de tableau :

$$\text{DCL}\quad T(bi_1 : bs_1,\ bi_2 : bs_2,\ \ldots,\ bi_n : bs_n)\ ;$$

Le vecteur d'accès pour le tableau T est implémenté de la façon contiguë suivante :

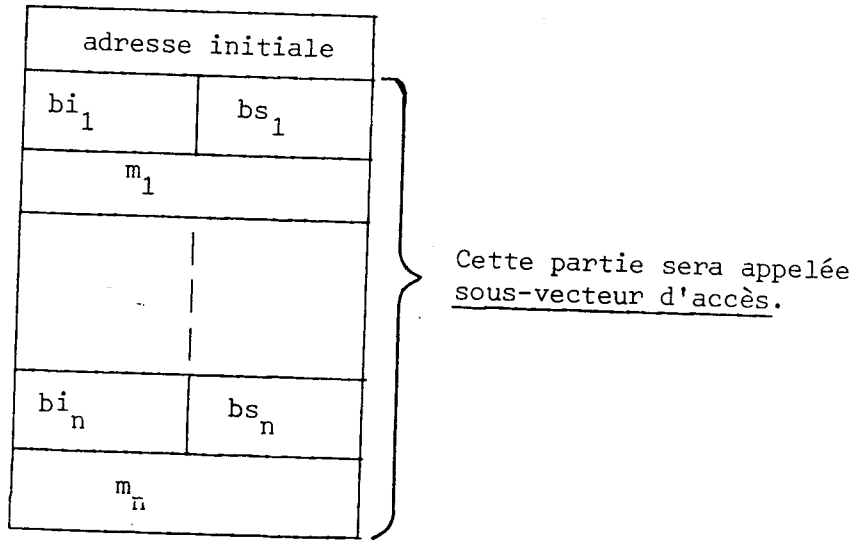


Figure 1.

Les bornes sont des entiers BIN FIXED d'un demi-mot et l'adresse initiale et les multiplicateurs sont des entiers BIN FIXED d'un mot.

Les formules : $m_n = 1$(mot)

$$m_{n-1} = (bs_n - bi_n + 1) * m_n$$

$$\vdots$$

$$m_2 = (bs_3 - bi_3 + 1) * m_3$$

$$m_1 = (bs_2 - bi_2 + 1) * m_2$$

$$adrinit = adr(T) - \sum_{j=1}^{n} m_j * bi_j$$

L'accès à un élément de tableau :

$$adr(T(ind_1, \ldots, ind_n)) = adrinit + \sum_{j=1}^{n} m_j * ind_j$$

Appelons le déplacement initial la quantité : $- \sum_{j=1}^{n} m_j * bi_j$

Appelons synonyme de tableau T(AUTO ou STATIC) tout tableau déclaré avec l'attribut DEFINED -dans le mode de correspondante directe (cf.[58])- sur le tableau T. Un synonyme de T spécifie le tableau T lui-même ou une partie de T. Pour l'accès aux éléments d'un synonyme de T on utilise le vecteur d'accès de T. Ainsi a-t-on besoin d'allouer un vecteur d'accès pour le synonyme S de T

seulement lorsque S est passé en argument. Les multiplicateurs et l'adresse initiale du vecteur d'accès de S sont ceux de T, les bornes étant celles de S. Appelons sous-tableau de T toute variable indéxée spécifiant le tableau T (ou l'un des synonymes de T) et ayant l'astérisque à la place d'au moins un indice. Un vecteur d'accès du sous-tableau doit être alloué si le sous-tableau est passé en argument.

2) - Tableaux AUTO et STATIC en tant qu'arguments -

Lorsqu'un tableau (un synonyme de tableau, un sous-tableau) est passé par référence, l'adresse de son vecteur d'accès est passée par valeur.

Lorsqu'un tableau (un synonyme de tableau, un sous-tableau ou une expression de tableaux) est passé par "dummy", un tableau intermédiaire est alloué en pile, un vecteur d'accès intermédiaire est alloué par recouvrement (cf. 5.4) dans la zone locale du corps de la procédure exécutant l'appel concerné et l'adresse du vecteur d'accès intermédiaire est passée par valeur. Un vecteur d'accès intermédiaire est nécessaire pour spécifier la nouvelle adresse initiale - celle correspondant au tableau intermédiaire empilé.

Notons qu'un tableau intermédiaire dont la taille est connue à la compilation pourrait être alloué par recouvrement comme son vecteur d'accès intermédiaire. Notre choix de l'empilage du tableau intermédiaire est, dans ce cas, tout-à-fait arbitraire.

3) - Allocation des vecteurs d'accès des tableaux AUTO et STATIC -

Un tableau STATIC est alloué dans la ZS2. Un tableau AUTO à bornes constantes est alloué dans la ZL du corps de la procédure déclarant le tableau. Un tableau AUTO à bornes variables est empilé.

a) - Les tableaux STATIC, les sous-tableaux des tableaux STATIC, les synonymes à bornes constantes des tableaux STATIC et les sous-tableaux de ces synonymes.

Les vecteurs d'accès de tous ces tableaux (synonymes et sous-tableaux) sont constants c'est-à-dire les bornes étant constantes les multiplicateurs le sont aussi et l'adresse initiale est connue au chargement. Les vecteurs d'accès sont donc allouables dans la ZS2 de la façon contiguë indiquée par la figure 1. L'allocation contiguë est nécessaire seulement dans le cas où ces tableaux sont passés en argument ; dans le cas contraire, on peut considérer les bornes,

les multiplicateurs et les adresses initiales comme des constantes indépendantes c'est-à-dire allouées dans la ZS2 mais ne formant pas nécessairement des vecteurs d'accès contigus. Ceci permet d'optimiser l'allocation dans la ZS2 dans les cas où existent plusieurs occurrences d'une même constante.

Si les tableaux cités dans le sous-titre sont passés en argument par "dummy", les adresses initiales des tableaux intermédiaires correspondants ne sont pas connues au chargement. Dans ce cas on alloue leurs sous-vecteurs d'accès et leurs déplacements initiaux dans la ZS2 parce qu'ils sont constants ; ils servent à établir - à l'exécution - des vecteurs d'accès intermédiaires alloués par recouvrement. Rappelons ici que le vecteur d'accès d'un synonyme de tableau T a les mêmes multiplicateurs et la même adresse initiale que le vecteur d'accès de T.

Les tableaux AUTO à bornes constantes déclarés dans des corps de procédures à l'attribut AS, leur sous-tableaux, leurs synonymes à bornes constantes et les sous-tableaux de ces synonymes sont traités de la même façon que les tableaux cités dans le sous-titre, les adresses initiales des vecteurs d'accès de tous ces tableaux étant connues au chargement.

b) - Les tableaux AUTO à bornes constantes déclarés dans des corps de procédures à l'attribut AD, leurs sous-tableaux, leurs synonymes, les sous-tableaux de ces synonymes et les tableaux AUTO à bornes variables, leurs sous-tableaux, leurs synonymes et les sous-tableaux de ces synonymes.

En ce qui concerne les tableaux AUTO cités dans le sous-titre, on doit allouer leurs vecteurs d'accès dans les zones locales des corps des procédures déclarant ces tableaux.

On agit ainsi pour les synonymes cités dans le sous-titre uniquement si ces synonymes sont passés en argument.

5.4 - Allocation par recouvrements : un exemple d'optimisation de l'espace des données -

Nous décrirons ici les allocations dans les deux zones statiques (ZS1 et ZS2, cf. 5.3.1). Nous définirons d'abord l'allocation par recouvrements des zones locales à l'attribut AS (cf. 5.2.1) déduite du graphe GPA (cf. 4.1 et 4.2.1) et ensuite les allocations à l'intérieur des zones locales.

5.4.1 - Recouvrement des zones locales -

Nous considérerons ici l'allocation statique des zones locales des procédures à l'attribut AS. En tenant compte du graphe potentiel d'appels (GPA) nous définirons un recouvrement de ces zones locales.

Rappelons que le GPA est un graphe (X, U) où les sommets de X représentent les corps des procédures d'un programme-source et les arcs de U les appels directs possibles. Les sommets de X représenteront aussi les zones locales des procédures et seront partitionnés, selon l'attribut d'allocation (AD ou AS) d'une zone locale, en deux ensembles :

$$X = XAD \cup XAS.$$

Soit ensuite la partition XAS = A∪B où

$$B = \{x \mid x \in XAS \text{ et } \exists\, y \in XAD \text{ tel que } (x, y) \text{ ou } (y, x) \in U\}$$

Afin de déduire du GPA un graphe potentiel d'appels GPAS = (XAS, V) nous pouvons procéder de la façon suivante :

soit G1 = (XAD∪B, W1) le sous-graphe du GPA engendré par XAD∪B⊂X et
soit $G1^+$ = (B, W2) le sous-graphe -engendré par B- du graphe correspondant à la fermeture transitive stricte de la matrice de connectivité de G1 ;
soit G2 = (XAS, W3) le sous-graphe du GPA engendré par XAS⊂X ;
le GPAS est alors un graphe (XAS, V = W2∪W3) obtenu par la "superposition" des deux graphes $G1^+$ et G2.
Notons que le GPAS, théoriquement, peut comporter des circuits qui sont des circuits apparents c'est-à-dire que, quelque soit l'occurrence d'exécution du programme, il ne peut exister de chemin effectif d'activation des procédures, dans le GPAS, qui puisse comporter des circuits. Bien que, dans la plupart des cas, le GPAS soit sans circuit il faut cependant tenir compte de cette possibilité théorique d'existence de circuits apparents dans le GPAS.
Notons d'autre part que le graphe de recouvrement GR déduit du GPAS ne doit pas comporter de circuits ; il définira ainsi un recouvrement par la relation successeur qui sera alors un ordre partiel sur les sommets du GR. Il faut donc supprimer les circuits apparents du GPAS. La méthode la plus simple sera la suivante :

- supprimer d'abord toutes les boucles (c'est-à-dire les arcs (x, x)) du GPAS ;

- chercher les composantes fortement connexes* (cfc) maximales du GPAS (elle sont disjointes). Pour toute cfc maximale (C, W) définissons les prédécesseurs PcXAS et des successeurs ScXAS :

$$P = \{x \mid x \notin cfc \wedge \exists y \in cfc \text{ tel que } (x, y) \in V\}$$

$$S = \{x \mid x \notin cfc \wedge \exists y \in cfc \text{ tel que } (y, x) \in V\}$$

Notons que toute cfc maximale a au moins un prédécesseur étant donné que le sommet correspondant au corps de la procédure MAIN, par hypothèse, ne peut faire partie d'une cfc.

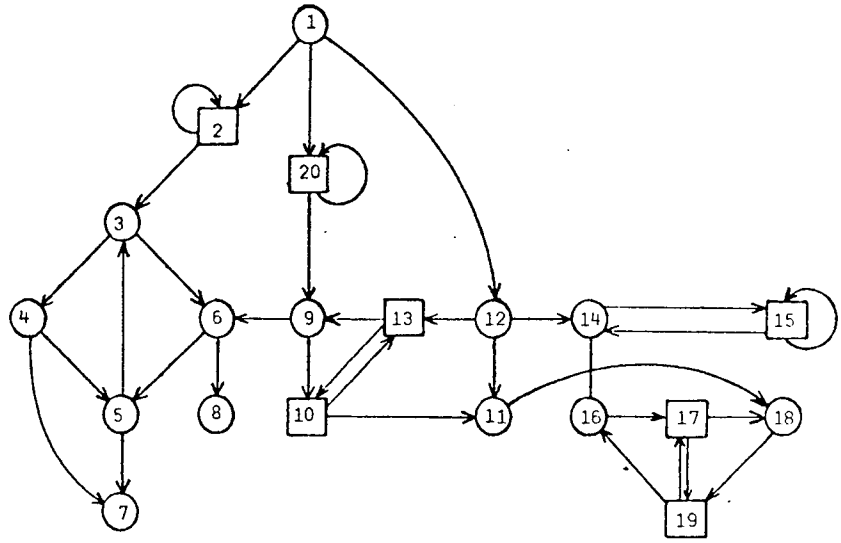
Si $C = \{c_1, c_2, \ldots, c_k\}$ on composera un chemin $c = c_1 \rightarrow c_2 \rightarrow \ldots \rightarrow c_k$ et on remplacera la cfc du GPAS par c de façon à ce que l'ensemble des prédécesseurs (immédaits) de c_1 soit P et l'ensemble de successeurs (immédiats) de c_k soit S.

Le graphe GR1 ainsi obtenu est sans circuit et définit correctement un recouvrement des zones locales correspondant à ses sommets.

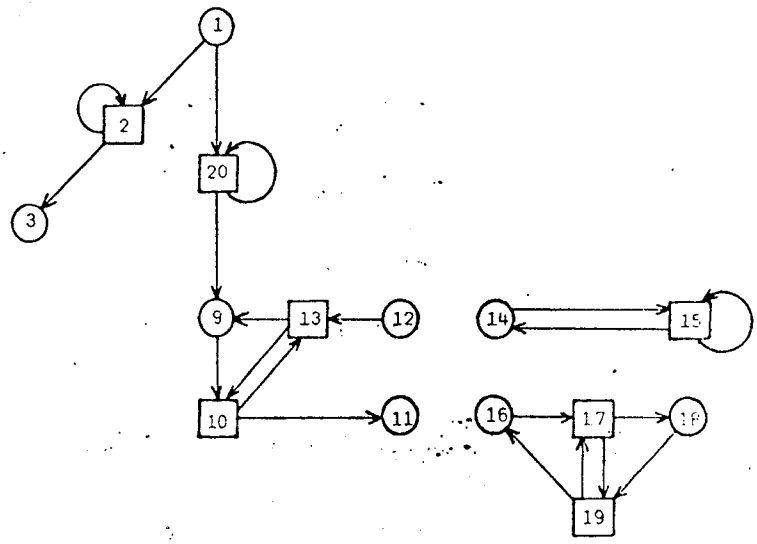
Notons que, du point de vue du recouvrement, un arc (x, y) du GR1 est une information redondante s'il existe un chemin (comportant 2 arcs ou plus) de x à y. De tels arcs peuvent donc être supprimés. D'une façon plus précise on cherche une relation (successeur) minimale P et telle que, si R est la relation successeur (immédiat) de GR1 alors $P^+ = R^+$ ($^+$désigne la fermeture transitive stricte). On peut facilement montrer qu'une telle relation minimale $P = R^-$ existe d'une façon unique si le graphe correspondant est sans circuit et que - en termes matriciels - $R^- = R \wedge \neg(R \times R^+)$.
Le graphe final GR sera donc celui de la relation R^- si R est la relation successeur de GR1.
Par la suite nous donnons un exemple du processus d'obtention du GR à partir d'un GPA.
Les sommets AS du GPA sont notés ○ et les sommets AD sont notés □ .

* Note : une composante fortement connexe cfc = (A, V) est un sous-graphe d'un graphe G tel que $\forall x, \forall y \in A$, il existe un chemin, dans cfc, de x à y et de y à x. .

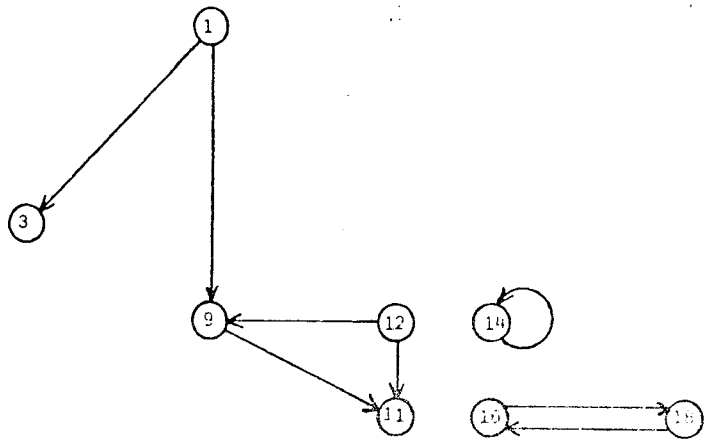
1) - Soit le GPA :



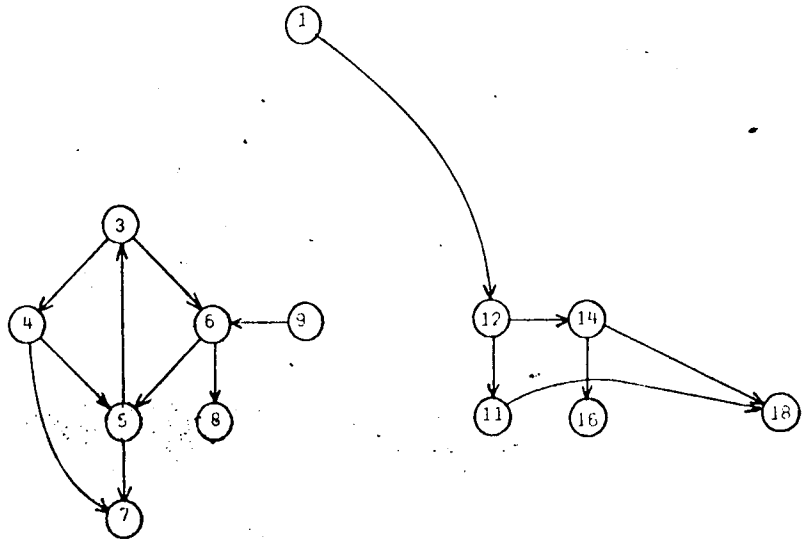
2) - Le G1 est le graphe :



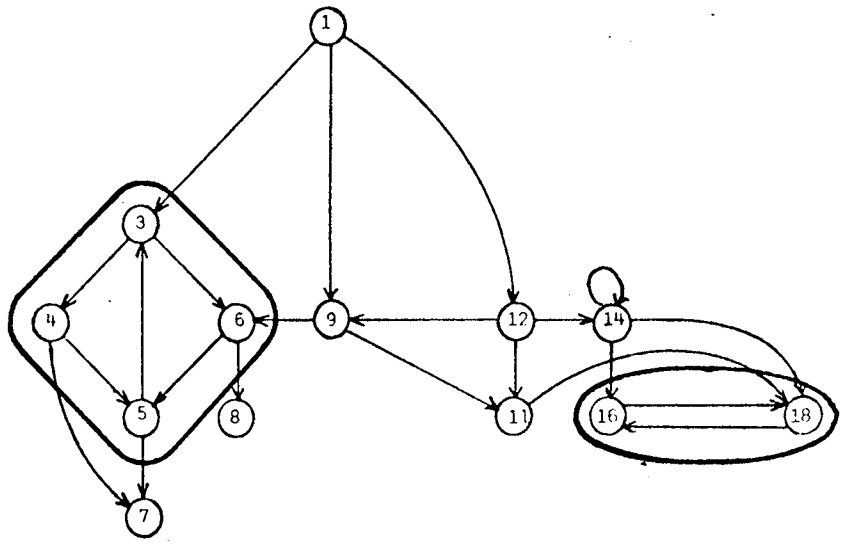
Le $G1^{+}$ déduit :



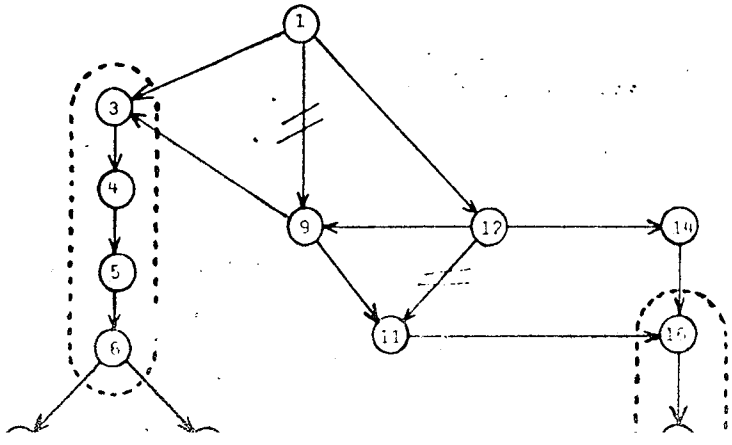
3) - Le G2 est le graphe :



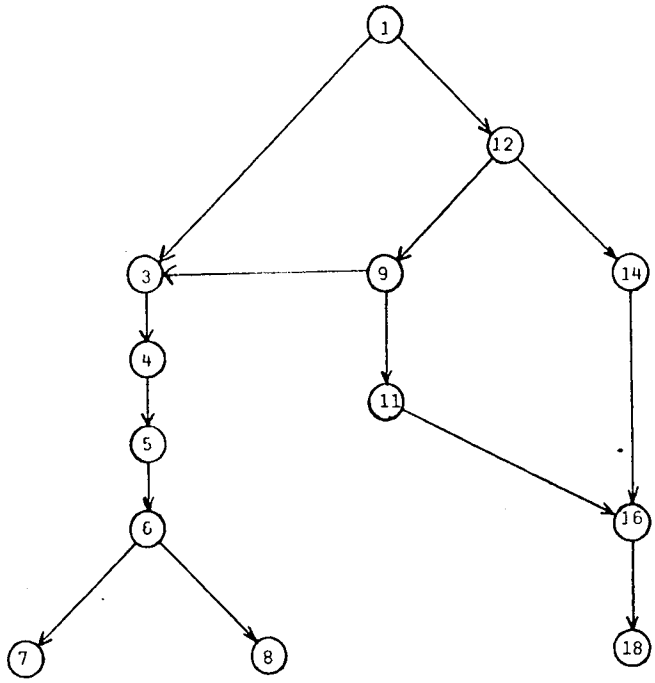
4) - Le GPAS = $G1^{+}$ et G2 ; les cfc (après la suppression des boucles) sont {3, 4, 5, 6} et {16, 18}.



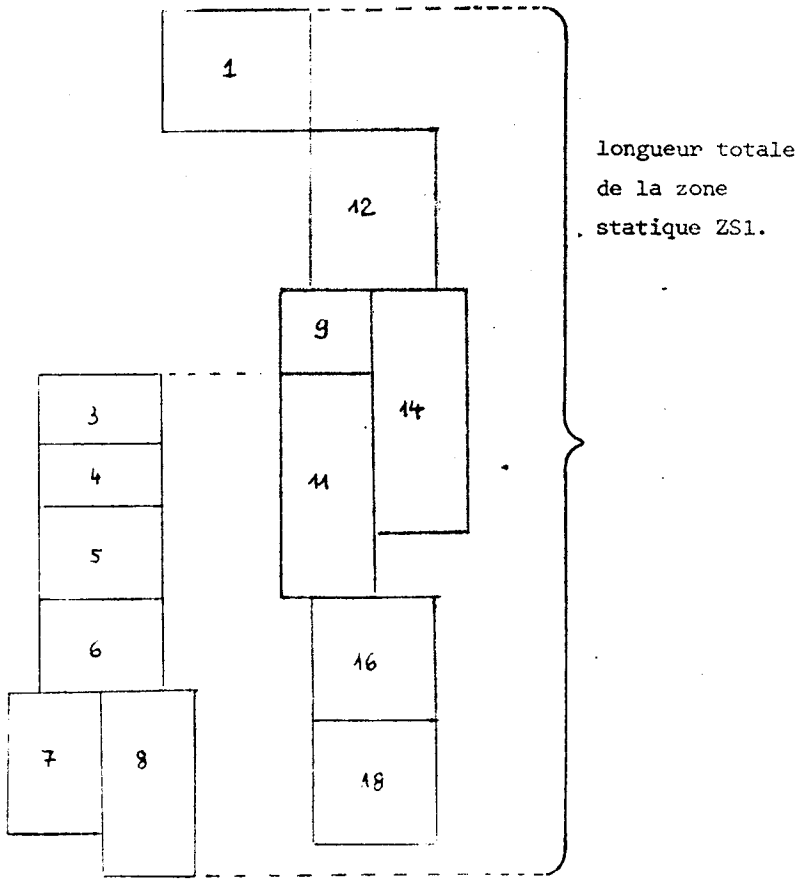
5) - Le graphe GR1 : les arcs supprimés par l'étape suivante (6) y sont notés.



6) - Le graphe final GR :



7) - L'allocation des zones locales correspondant aux sommets du GR peut alors être schématisée ainsi :



5.4.2 - Allocation par recouvrements à l'intérieur d'une zone locale -

Rappelons qu'une zone locale correspondant à un corps de procédure a une taille statiquement connue ; elle contient les données-source de taille fixe locales au corps de procédure associé ainsi que certaines données résultant de l'implémentation. Plus exactement on distinguera dans une ZL deux parties principales allouées sériellement :

- l'espace initial (ou la partie initiale, voir 5.3.4) contenant entre autres la zone de sauvegarde des registres, la zone des pointeurs, le vecteur local réduit d'environnement et la zone des paramètres ;
- l'espace des fiefs où les zones des fiefs (une zone de fief comprend les données-source locales au fief) sont allouées par recouvrement. Ce recouvreme étant donné l'activabilité "in situ" des fiefs-début d'un corps de procédure, est défini par l'arbre ASFP (cf. 4.1) dont la racine est le fief-procédure du corps de procédure considéré.

Dans ce chapitre nous émettrons quelques considérations concernant l'allocati des temporaires (introduits par l'évaluation des expressions) et l'allocatior des listes d'arguments et des arguments intermédiaires ("dummies"). Nous nommerons les données énumérées ci-dessus les données générées, par oppositic aux données-source.

Remarquons tout d'abord que, ordinairement, l'allocation des données-source n'est pas traitée de la même façon que l'allocation des données générées : le premières sont toujours allouées en mémoire - indépendamment de l'allocation des registres - et à l'intérieur du schéma des recouvrements défini par ASFP alors que l'allocation en mémoire des secondes peut dépendre des méthodes d'analyse d'une part et de l'allocation des registres d'autre part (nous pensons ici aux temporaires).

La distinction entre les temporaires et les données-source est établie implicitement par le programmeur ; il est raisonnable de la maintenir tant que la compilation est faite d'une façon peu sophistiquée : les temporaires sont évalués une seule fois (dans l'espace d'un texte intermédiaire ou du texte-objet) et utilisés une seule fois alors que ce n'est pas vrai a priori pour les données-source. Mais dès qu'on essaye de trouver des temporaires équivalents et qu'on aboutit ainsi à des mises en commun des temporaires et à des évaluations et utilisations multiples d'un même temporaire cette distinction

première se perd. Aussi, l'allocation en registres des certaines données-source (les compteurs des boucles DO, par exemple) peut être plus efficace en termes du temps d'exécution et peut rendre inutile leur allocation en mémoire.

Reprenons : la distinction de traitement entre un temporaire et une donnée-source consiste, pour le premier, à l'allouer d'abord dans un registre et ensuite - si nécessaire - en mémoire alors que c'est le contraire pour la seconde ; l'hypothèse sous-jacente consiste à supposer, d'une part, que "la portée d'utilisation" d'un temporaire reste "petite" par rapport à la portée d'utilisation d'une donnée-source (qu'on confond avec sa portée tout court) et, d'autre part, qu'il est plus profitable d'allouer les données à "petite" portée d'utilisation dans des registres. Comme on l'a vu ci-dessus, la première partie de cette hypothèse peut se révéler fausse dans les cas de mises en commun des temporaires et la seconde partie semble arbitraire.
Ajoutons que, en ce qui concerne l'allocation des registres, aucune hypothèse n'a probablement été formulée explicitement quant à la profitabilité - en termes de diminution du temps d'exécution - d'une méthode d'allocation des registres particulière ; ainsi les méthodes locales (c'est-à-dire au deça du niveau d'une instruction-source ou d'une instruction en un texte intermédiaire ou un texte-objet) d'allocation des registres se sont imposées par leur simplicité c'est-à-dire que le critère de leur choix a été une compilation rapide.

Par la suite nous examinerons comment on pourra inclure l'allocation en mémoire des données générées à taille fixe dans le schéma des recouvrements défini par ASFP. Nous distinguerons les temporaires communs (c'est-à-dire mis en commun) des autres temporaires et nous considérerons tous les temporaires alloués en mémoire a priori (ils pourront être, dans la phase d'allocation des registres, alloués dans des registres).

5.4.2.1 - Temporaires -

En supposant les noms des données (du programme-source ou d'un texte intermédiaire) uniques, nous appellerons un temporaire une occurrence de (sous) expression du texte (source ou intermédiaire) d'un fief donné.

Deux temporaires seront dits <u>équivalents</u> s'ils correspondent à deux expressions littéralement égales et dont les occurrences devront faire partie du texte d'un même corps de procédure.

Nous dirons que le point d'allocation d'un temporaire est le fief de son occurrence. Les temporaires faisant partie d'une classe d'équivalence détermineront un temporaire commun dont le point d'allocation sera, au sens de fiefs et de ASFP, le prédominateur immédiat* dans ASFP des points d'allocation (fiefs) des éléments de sa classe.

Tous les temporaires communs ayant pour point d'allocation un fief donné f seront alloués dans la zone des communs de f. Tous les autres temporaires ayant f pour point d'allocation seront alloués dans la zone des temporaires de f. Il est évident que ces deux zones, lorsqu'elles sont relatives à un même fief, ne doivent pas se recouvrir.

Un exemple simple illustrera l'incorporation de ces deux zones dans le schéma des recouvrements défini par ASFP.
Soit le texte source (les noms d'identificateurs étant uniques) :

```
B1 : BEGIN ; DCL  A, B, I ;
     B2 : BEGIN ; DCL M ;
     .
     .
     D  : = C/(A+B) ; K := I+M ;
     .
     .
     B3 : BEGIN ;
          .
          .
          K1 := I+M -12 ;
          .
          .
          END B3 ;
          END B2 ;
     .
     .
     B4 : BEGIN ;
          .
          .
          D1 := C1*(A+B) ;
          .
          .
          F := G/(A+B) ;
          .
          .
          END B4 ;
     .
     .
     END B1 ;
```

* Note : le prédominateur immédiat d'un ensemble de sommets d'un arbre est la racine du plus petit sousarbre contenant tous les sommets de l'ensemble considéré.

Les occurrences de A+B des fiefs-début B2 et B4 sont équivalentes et le temporaire commun T1 qu'elles définissent a pour point d'allocation le fief-début B1 ; les occurrences de I+M des fiefs-début B2, B3 sont équivalentes et définissent un temporaire T2 dont le point d'allocation est le fief-début B2. Le schéma de ASFP initial (comprenant les zones des fiefs-début B1, B2, B3 et B4), augmenté par des zones des temporaires pour les quatre fiefs-début et par des zones des communs relatifs aux fiefs B1 et B2 est donné par la figure 1.

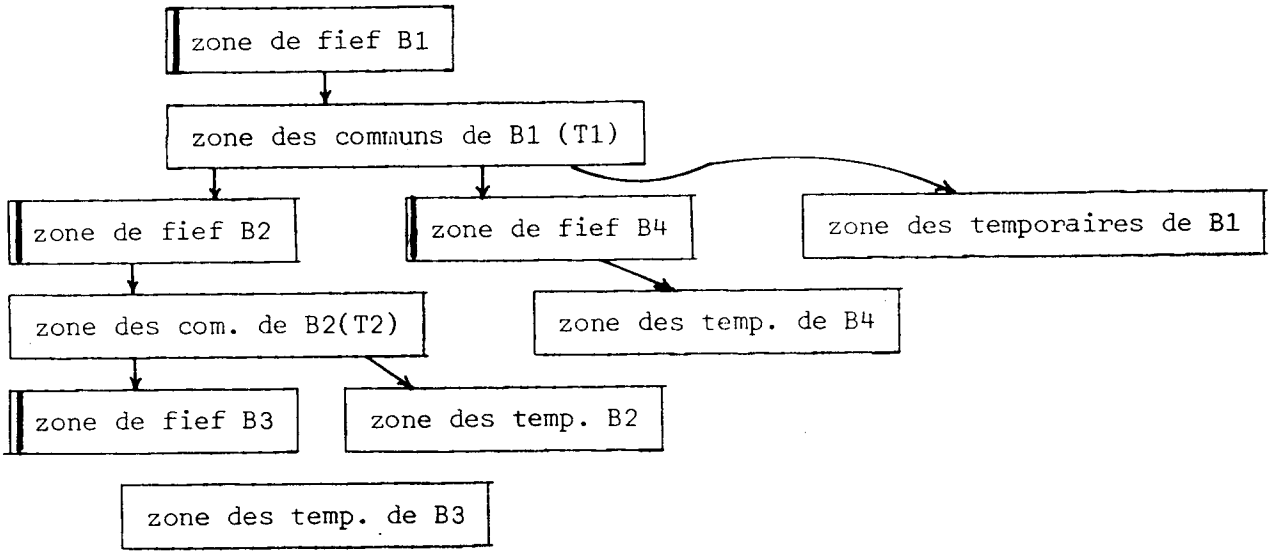


Figure 1.

On voit qu'il s'agit au fait de l'élargissement d'une zone de fief par la zone des communs du même fief. Nous ne nous occuperons pas ici de l'allocation à l'intérieur de ces zones, ni de l'efficacité, en termes du temps d'exécution, de la mise en commun des temporaires.

5.4.2.2 - Listes d'arguments et les arguments intermédiaires à taille fixe -

Nous allouerons, conformément à l'optique énoncée dans le chapitre 5.3.1, les arguments intermédiaires à taille fixe statiquement exception faite des arguments intermédiaires qui sont des tableaux (voir 5.3.8) et nous incorporerons l'allocation statique des listes d'arguments et des arguments intermédiaires à taille fixe (c'est-à-dire des arguments intermédiaires simples et des vecteurs d'accès intermédiaires) dans le schéma des recouvrements défini par ASFP.

Nous supposerons que l'ordre d'évaluation des arguments d'un appel n'est pas défini par le langage et qu'il peut donc être librement changé.

Une occurrence d'appel définira un arbre d'appel : la racine sera le nom (unique) de la procédure (sous-programme ou fonction) appelée et les sommets-fils représenteront les arguments de la liste d'appel ; au sommet représentant un argument qui est une occurrence d'appel (un argument d'appel) d'une procédure-fonction sera substitué l'arbre d'appel correspondant. Un arbre d'appel sera dit maximal s'il n'est pas sousarbre d'un arbre d'appel.

Exemple : l'instruction source

```
CALL P (A + 3, B, F1 (C * 2, D-F6(J), F2 (E, F), G(2, *)),
     F3 (H/2, (I), F4 (K * L), F5), M * N) ;
```

définit deux arbres d'appel maximaux donnés par la figure 2.

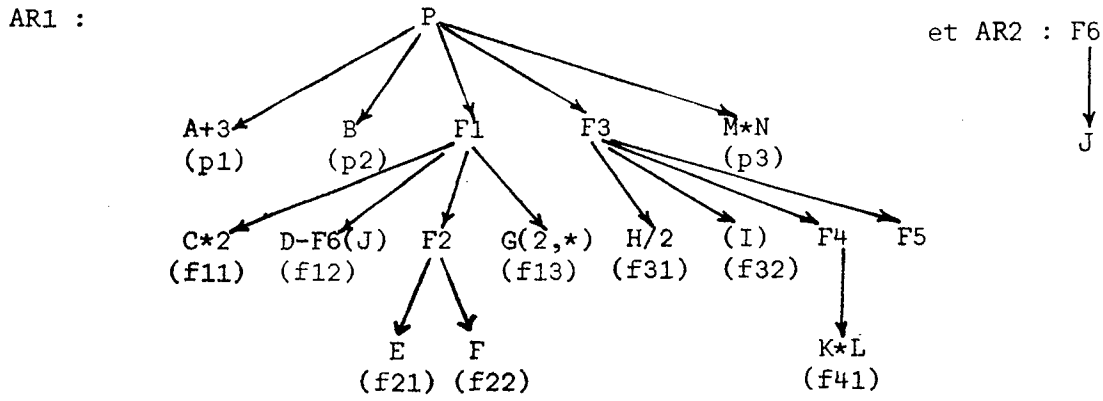


Figure 2.

A chaque sommet on associera l'information concernant la taille de l'espace à allouer et le genre de l'argument associé :

- pour la racine : la taille de la liste des arguments et une indication spécifiant procédure - sousprogramme ou procédure-fonction ;
- pour tout autre sommet :
  - si c'est un argument d'appel : une indication spécifiant qu'il s'agit d'un argument d'appel, la taille de la liste des arguments et le mode de passage (par "dummy" ou par valeur) ;
  - sinon : si l'argument est passé par référence ou si l'argument (simple) est passé par valeur : la taille à allouer est nulle ;

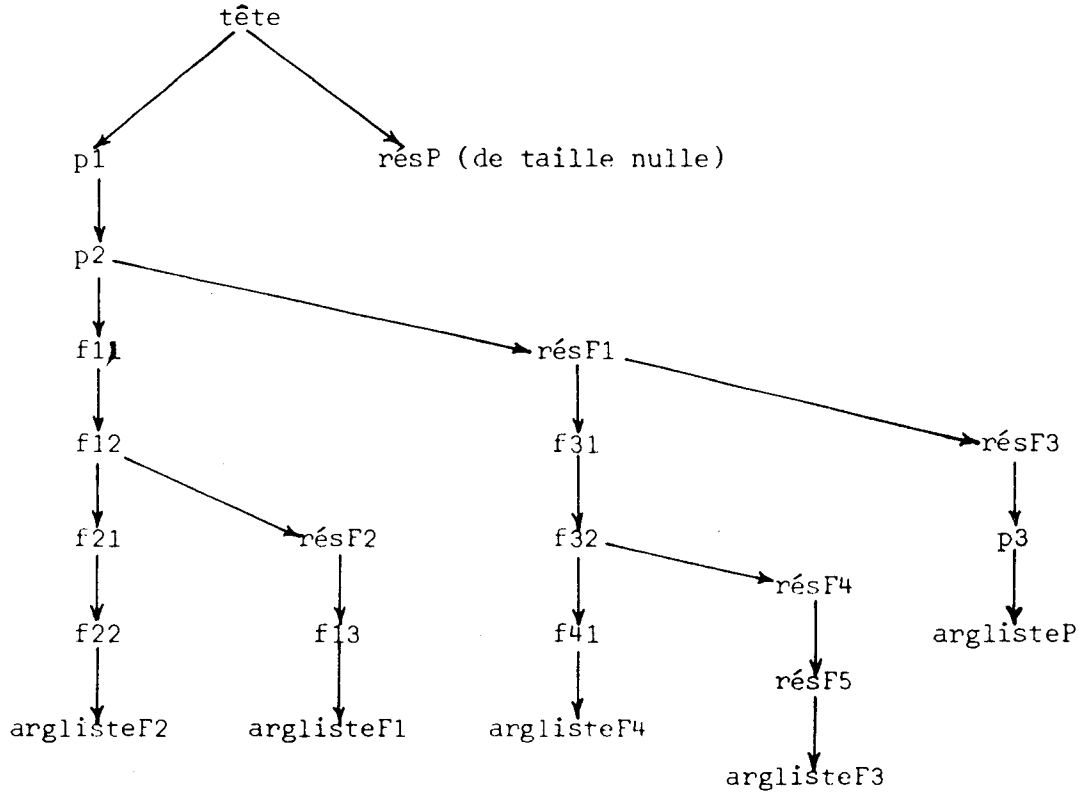
sinon : si l'argument est simple (et passé par "dummy") la taille à allouer est d'un mot sinon (c'est-à-dire l'argument est composé) la taille à allouer est celle du vecteur d'accès intermédiaire.

D'un arbre d'appel nous déduirons deux familles d'arbres d'allocation possibles définissant, à la façon de ASFP, une allocation des listes d'arguments et des arguments intermédiaires. Les sommets d'un arbre d'allocation représentent des données générées (listes d'arguments, arguments intermédiaires) et indiquent la taille de l'espace qu'elles occupent.

Les deux familles d'arbres d'allocation correspondront aux deux parcours par ordres postfixé et préfixé [57] des sommets d'un arbre d'appel.

Un sommet-noté Fi - représentant la racine ou un argument d'appel d'un arbre d'appel dénotera, dans l'arbre d'allocation correspondant, le résultat de la procédure Fi (la taille associée sera nulle si Fi représente la racine d'un arbre d'appel maximal) et sera alors noté résFi et fournira un nouveau sommet - noté arglisteFi - dans l'arbre d'allocation correspondant si, toutefois, le nombre d'arguments de la procédure Fi est non nul. Un sommet-tête (spécifiant une taille nulle) sera ajouté à l'arbre d'allocation.

Ainsi, pour reprendre l'exemple précédent, le parcours postfixé ("postorder traversal") de l'arbre AR1 fournira l'arbre d'allocation de la figure 3



alors que le parcours préfixe ("preorder traversal") fournira l'arbre d'allc cation de la figure 4.

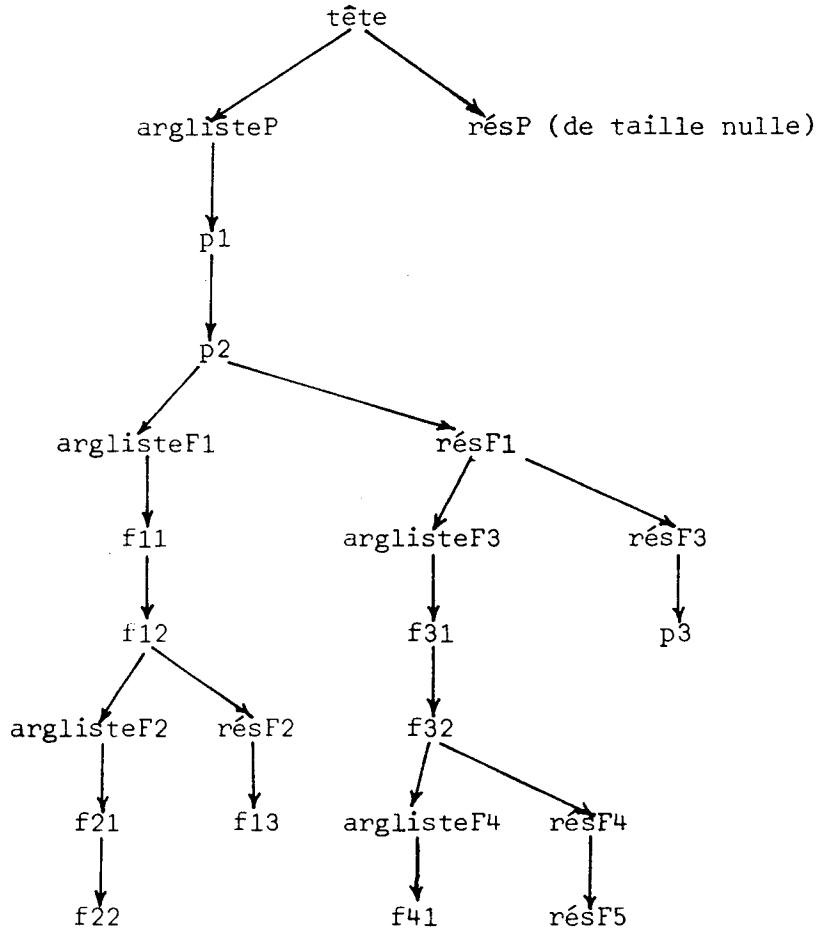


Figure 4.

Par la suite nous présenterons deux algorithmes construisant les deux arbres d'allocation possibles à partir d'un arbre d'appel et nous comparerons les deux solutions.

Nous montrerons finalement comment faire l'allocation statique

- à l'intérieur du schéma des recouvrements défini par ASFP - d'une donnée générée (composée des listes d'arguments et des arguments intermédiaires) qu représente un arbre d'allocation déduit d'un arbre d'appel maximal.

a) - Algorithme A2 (le parcours postfixé) -

L'algorithme A2 transforme un arbre binaire [57] AB, correspondant naturel [57] - augmenté d'un sommet-tête - d'un arbre d'appel maximal A en un arbre binaire R, correspondant naturel de l'arbre d'allocation AL de A. Nous supposons que tout sommet de AB possède trois champs :

- pointeur gauche (vers le fils gauche)
- pointeur droit (vers le fils droit)
- information :
  - . spécificateur indiquant s'il s'agit d'un sommet représentant un nom de procédure ou un argument d'appel
  - . spécificateur indiquant - au cas où le sommet représente un nom de procédure ou un argument d'appel de fonction - la nullité ou non du nombre d'arguments de la procédure ou de la fonction
  - . taille à allouer pour le sommet (un sommet-tête a une taille nulle).

Le pointeur gauche du sommet-tête de AB pointe vers la racine de A et le pointeur droit de la racine de A pointe vers le sommet-tête.

L'algorithme modifie directement AB, en ajoutant éventuellement de nouveaux sommets (représentant des listes d'arguments).

Pour simplifier la description de l'algorithme nous supposons disponibles les procédures et les fonctions suivantes :

- psommet (P) : fonction booléenne qui est vrai si le sommet dont l'adresse est P représente un nom de procédure ou un argument d'appel et faux sinon ;
- args(P) : fonction booléenne s'appliquant à un sommet d'adresse P tel que psommet(P) est vrai et qui est vrai si la procédure correspondant au sommet d'adresse P a au moins un argument et faux sinon ;
- zéro(P) : procédure mettant à nil les deux pointeurs du sommet d'adresse P ;
- infixsucc(P) : procédure (dont l'argument implicite est l'arbre AB) fournissant l'adresse P du sommet qui est le successeur en ordre infixé (dans AB) du sommet fourni par l'appel précédent de infixsucc. Le premier appel fournit le premier sommet-en ordre infixé- de AB et le dernier appel fournit le sommet-tête. La procédure doit utiliser une pile étant donné que l'algorithme modifie les pointeurs des sommets déjà parcourus en ordre infixé.
- créer(Q, P) : procédure créant un nouveau sommet d'adresse Q à partir du sommet d'adresse P de AB ; on doit avoir psommet(P) = vrai et args(P) = vrai.

Le nouveau sommet représente la liste des arguments de la procédure corresp à P ; son champ-taille est déduit de celui de P et ses pointeurs sont mis à nil par zéro (Q).

Les deux pointeurs d'un sommet d'adresse P seront désignés par gauche(P) et droite(P).

Nous pouvons maintenant écrire l'algorithme A2 :

- la donnée à l'entrée : l'adresse H du sommet-tête d'un arbre binaire AB d'au moins deux sommets ;
- le résultat : un arbre binaire dont le sommet-tête est à l'adresse H ;
- l'algorithme :

```
début{A2}
PT := H ;
infixsucc(cour1) ; zéro(PT) ;
faire infixsucc(cour2) ;
     si psommet(cour1) ∧ args(cour1) alors
     créer(nouveau, cour1) ; gauche(PT) := nouveau ;
     droite (gauche(cour1)) := cour1
     sinon gauche(PT) := cour1 fsi ;
     PT := cour1 ; zéro(cour1) ; cour1 := cour2
tantque cour1 ≠ head fin
fin A2.
```

Exemple : l'arbre d'appel maximal A1 de la figure 5

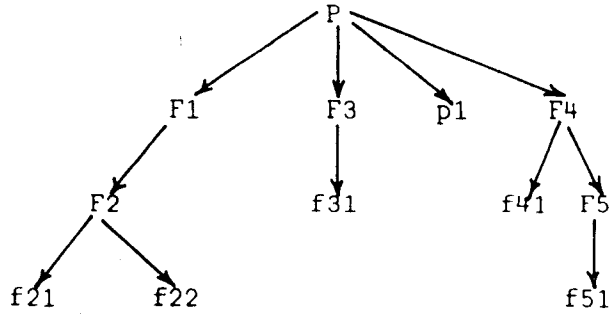


Figure 5. L'arbre A1

a pour correspondant naturel l'arbre AB1 de la figure 6

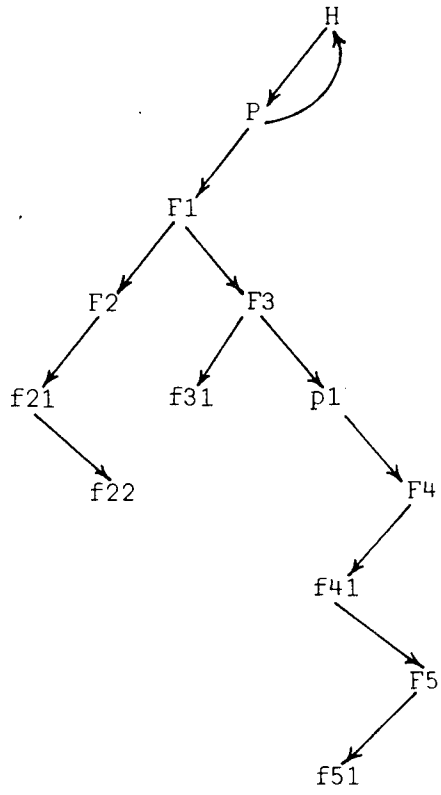


Figure 6. L'arbre AB1

que l'algorithme A2 transforme en l'arbre binaire R1 (figure 7)

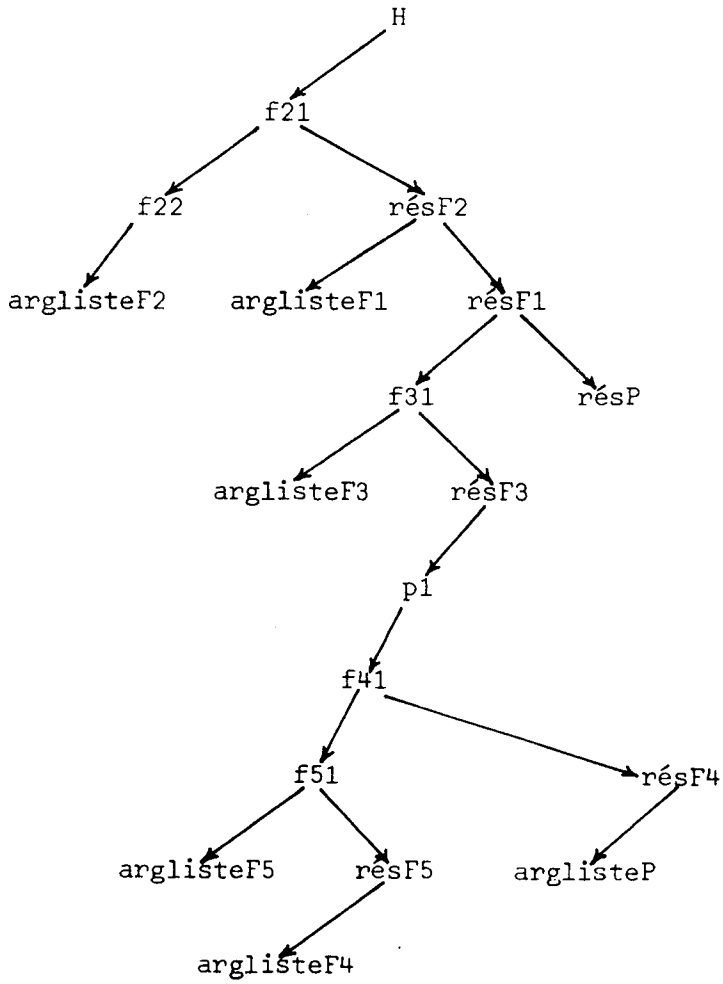
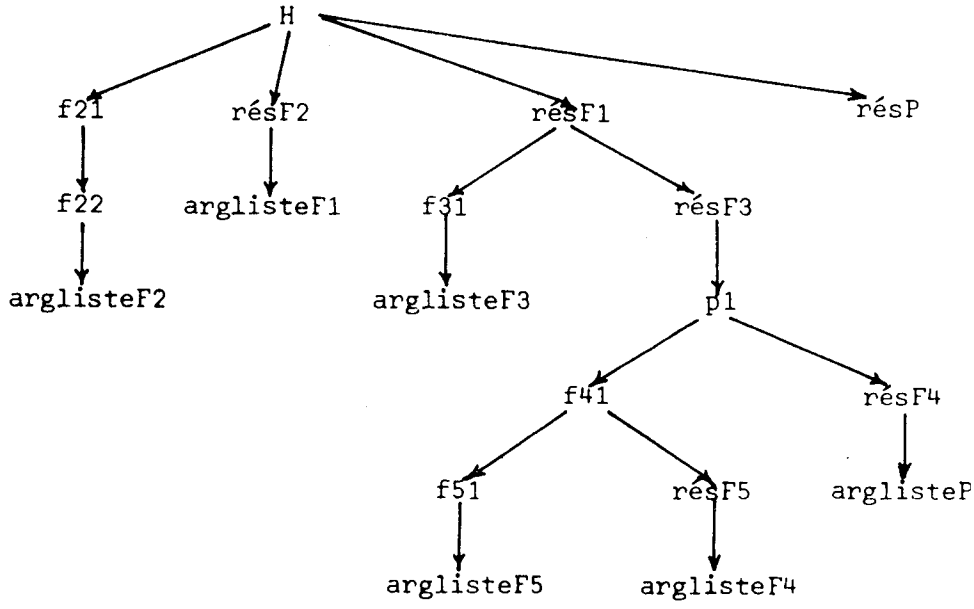


Figure 7. L'arbre R1

dont le correspondant naturel non binaire est l'arbre d'allocation AL1 de A1 (figure 8).



L'ordre de gauche à droite des sommet-fils d'un sommet d'un arbre d'appel A influe sur la taille définie par l'arbre d'allocation AL de A. Proposons-nous de déterminer l'ordre optimal c'est-à-dire celui qui fournira un arbre d'allocation de taille minimale.

Cherchons d'abord à étudier la forme d'un arbre d'allocation.

Les sommets Fi, résFi, arglisteFi, seront parfois dénotés avec les i en position d'indice ; tous les autres sommets seront dénotés par a_i. On dénotera par saFi ou saF_i l'arbre d'allocation-obtenu au moyen de l'algorithme A2 d'un arbre d'appel de racine F_i - privé du sommet $résF_i$.
Avec ces notations l'arbre d'allocation AL1 de l'arbre d'appel A1 (F0 = P, a_1 = p1) de l'exemple précédent sera schématisé de la façon indiquée dans la figure 9.

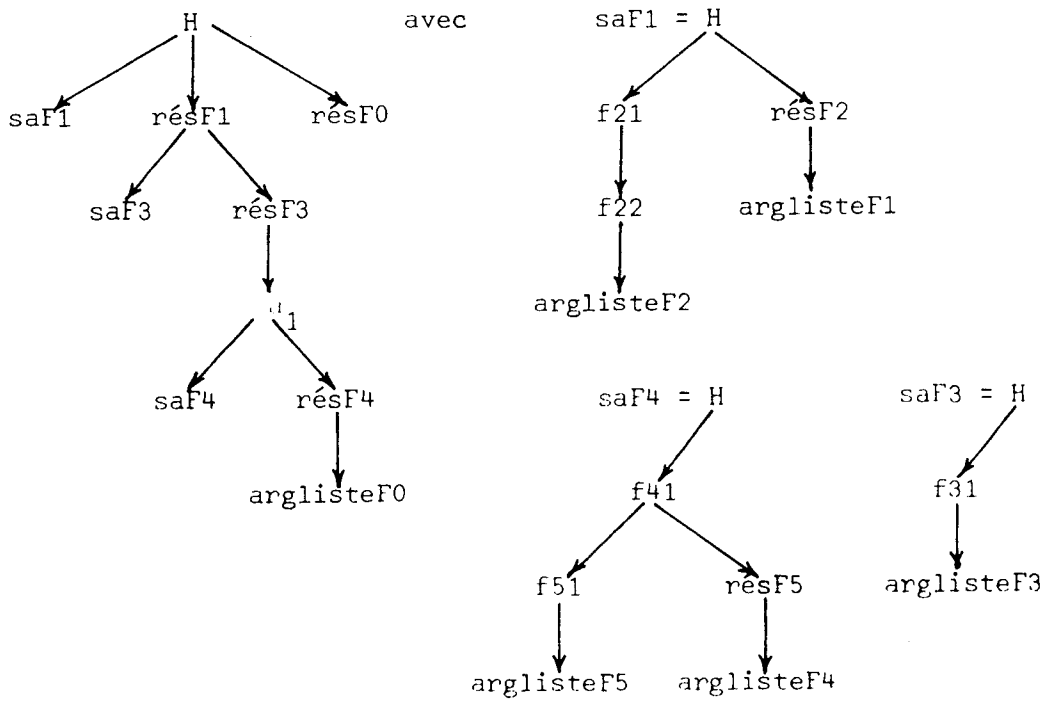


Figure 9.

On peut vérifier, en généralisant, qu'à des séquences des fils de F_0 : (F_i, a_j), (F_i, F_j), (a_i, a_j) et (a_i, F_j) d'un arbre d'appel de racine F_0 correspondent - dans l'arbre d'allocation correspondant - les arbres schématisés dans la figure 10.

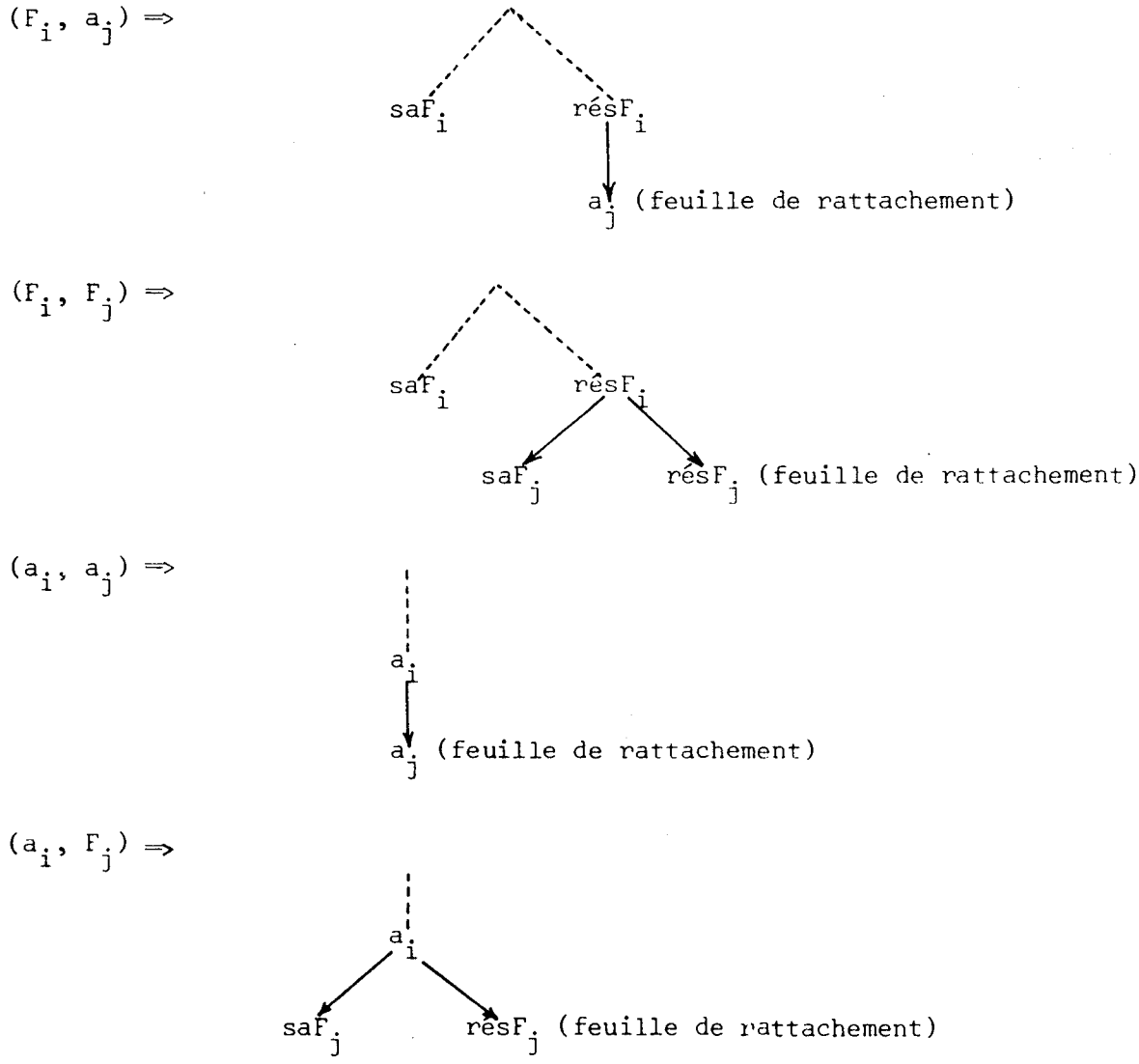


Figure 10.

Le sommet **arg**listeF_0 est l'unique fils de la feuille de rattachement de l'arbre correspondant à la séquence des deux derniers fils de F_0.

Dénotons un arbre d'appel par la notation en préordre parenthésée [57] ; l'arbre d'appel A1, par exemple, est noté :

$$F_0(F_1(F_2(f21, f22)),F_3(f31),a_1,F_4(f41,F_5(f51))).$$

On constate que la taille (en mots) de l'arbre d'allocation AL1 est définie (récursivement) par la formule suivante :

$$\begin{aligned}\text{taille(AL1)} = \max\ (&\text{taille}\ (\text{résF}_0),\ \text{taille}\ (\text{saF}_1),\\ &\text{taille}\ (\text{saF}_3) + 1,\\ &\text{taille}\ (a_1) + 2 + \text{taille}\ (\text{saF}_4)\ ,\\ &\text{taille}\ (a_1) + 3 + \text{taille}\ (\text{arglisteF}_0)).\end{aligned}$$

Proposition 1 : la taille définie par l'arbre d'allocation déduit au moyen de l'algorithme A2 d'un arbre d'appel A de racine F_0 ayant pour fils les deux ensembles de sommets $\{F_i\}$, $i \in I$ et $\{a_j\}$, $j \in J$ est minimale lorsqu'on ordonne ces fils de la façon suivante (l'ordre défini ci-dessous déterminera l'arbre d'appel - déduit de l'arbre A - optimal par rapport à la taille, relativement à A2) :

- $\forall i \in I, \forall j \in J$: F_i est à gauche de a_j (ce n'est toutefois pas nécessaire si F_i correspond à une procédure sans argument ; taille (saF_i) est alors nulle) et

- $\forall i_1, \forall i_2 \in I$: F_{i_1} est à gauche de $F_{i_2} \iff$

$$\text{taille}\ (\text{saF}_{i_1}) \geq \text{taille}\ (\text{saF}_{i_2})$$

- et les arbres d'appel de racines F_i, $i \in I$ sont optimaux par rapport à la taille, relativement à A2.

Démonstration : la taille correspondant à un arbre d'appel

$$F_0(F_1\ (\ldots),\ F_2\ (\ldots),\ \ldots\ F_n(\ldots),\ a_1,\ a_2,\ \ldots,\ a_m)$$

est définie récursivement par la formule suivante :

$$\begin{aligned}\text{T1} = \max\ (&\text{taille}\ (\text{résF}_0),\ \text{taille}\ (\text{saF}_1)\ ,\\ &\text{taille}\ (\text{saF}_2) + 1,\\ &\cdot\\ &\cdot\\ &\cdot\\ &\text{taille}\ (\text{saF}_n) + (n-1),\\ &\sum_{i=1}^{m} \text{taille}\ (a_i) + n + \text{taille}\ (\text{arglisteF}_0)).\end{aligned}$$

Si l'un des a_i, $1 \leq i \leq m$, est mis à gauche d'un F_j, $1 \leq j \leq n$, l'arbre d'appel résultant :

$F_0(F_1(...), ..., F_j(...), a_i, F_{j+1}(...), ..., F_n(...), a_1, ... a_{i-1}, a_{i+1}, ..., a_m)$

aura une taille :

T2 = max (taille ($résF_0$), taille (saF_1),
taille (saF_2) + 1,
.
.
.
taille (saF_j) + (j-1),
taille (saF_{j+1}) + j + taille(a_i),
.
.
.
taille (saF_n) + (n-1) + taille(a_i),
$\sum_{i=1}^{m}$ taille (a_i) + n + taille (argliste F_0)).

} termes augmentés

Il est évident que T2 ≥ T1.

Exemple : nous transformerons l'arbre A1 de l'exemple précédent (figure 5) - en supposant que : taille(saF_1)≥taille(saF_3)≥taille(saF_4)) - en l'arbre optimal A1' (figure 11)

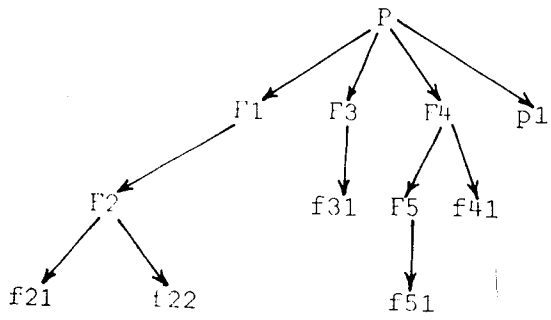


Figure 11. L'arbre A1'

et nous obtiendrons l'arbre d'allocation AL1' à taille minimale (voir la figure 12).

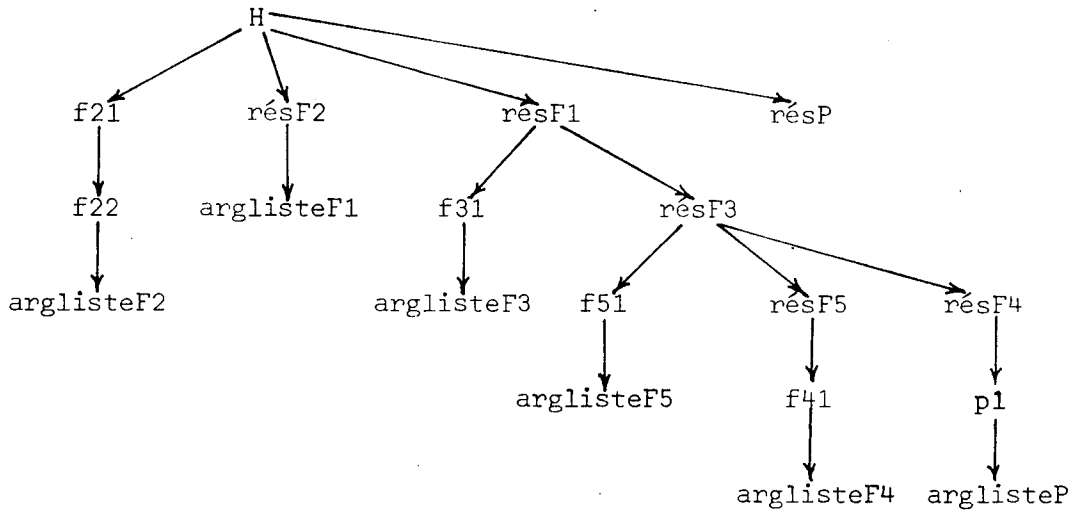


Figure 12 : L'arbre AL1'.

Notons, à la fin de ce souschapitre, que les considérations menant à la formule donnant la taille d'un arbre d'allocation nous suggèrent un algorithme récursif de parcours en préordre d'un arbre d'appel fournissant directement, sans passer par les correspondants naturels binaires, l'arbre d'allocation défini par l'algorithme A2.

b) - Algorithme A3 (le parcours préfixé) -

Les hypothèses de départ sont celles de l'algorithme A2. Nous supposerons disponibles, en plus de celles décrites dans le souschapitre b), les procédures et les fonctions suivantes :

- marquer (P) : procédure marquant le sommet d'adresse P ;
- marqué (P) : fonction booléenne vrai si le sommet d'adresse P est marqué et faux sinon ;
- empiler (P) et dépiler (P) : procédures utilisant une pile implicite et empilant l'adresse P sur la pile ou désempilant la pile dans la variable P ;
- préfixsucc (P) : procédure analogue à infixsucc ; le dernier appel fournit le sommet-tête de l'arbre binaire AB et le premier le fils gauche du sommet-tête de AB. Etant donné que l'algorithme peut changer les pointeurs des sommets déjà parcourus - dans l'ordre préfixé - la procédure préfixsucc doit utiliser une pile pour empiler les fils droits des sommets déjà parcourus.

L'algorithme A3 :

- la donnée à l'entrée : l'adresse H du sommet-tête d'un arbre binaire AB d'au moins deux sommets ;
- le résultat : un arbre binaire dont le sommet-tête est à l'adresse H ;
- L'algorithme :

```
début {tous les sommets sont non marqués}
PT := H ;
préfixsucc (cour1) ;
faire préfixsucc (cour2) ;
     si psommet (cour1) ^ args (cour1) alors
     créer (nouveau, cour1) ;
          si marqué (cour1) alors dépiler(top) ;
          gauche (top) := nouveau
          sinon gauche (PT) := nouveau fsi ;
          droite (nouveau) := cour1 ;
          si droite(cour1) ≠ nil alors marquer (droite(cour1)) ;
          empiler (cour1) fsi ; PT := nouveau
     sinon si marqué (cour1) alors dépiler (top) ; gauche (top) := cour1
          sinon gauche (PT) := cour1 fsi ;
          PT := cour1
     fsi
     zéro (cour1) ; cour1 := cour2
tantque cour1 ≠ H fin
fin A3.
```

Exemple : Reprenons l'arbre d'appel A1 et son correspondant naturel AB1 du souschapitre précédent ; AB1 est transformé par A3 en l'arbre binaire R2 (figure 13)

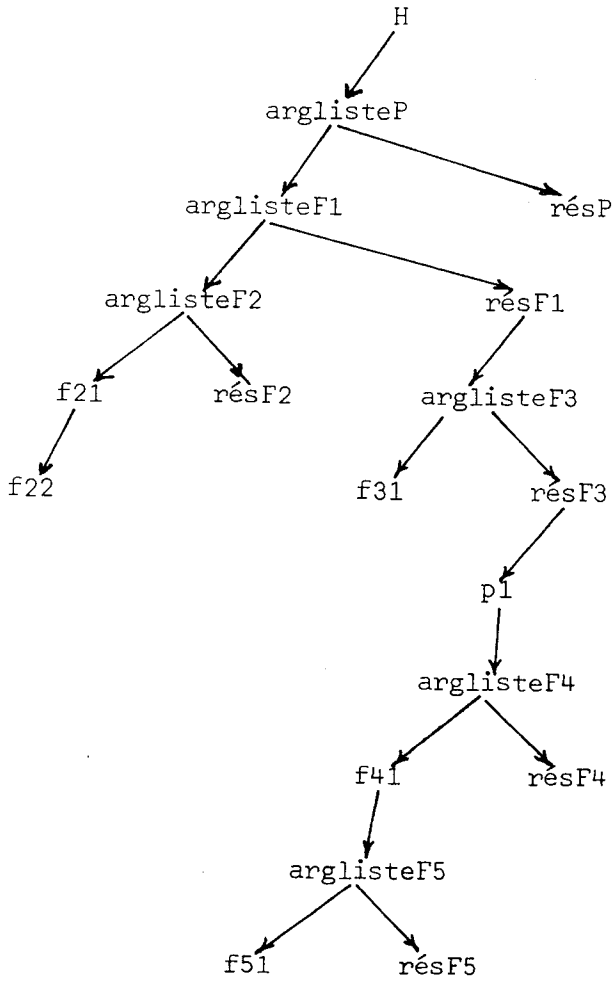


Figure 13 : L'arbre R2.

dont le correspondant naturel est l'arbre d'allocation AL2 de A1 (figure 14).

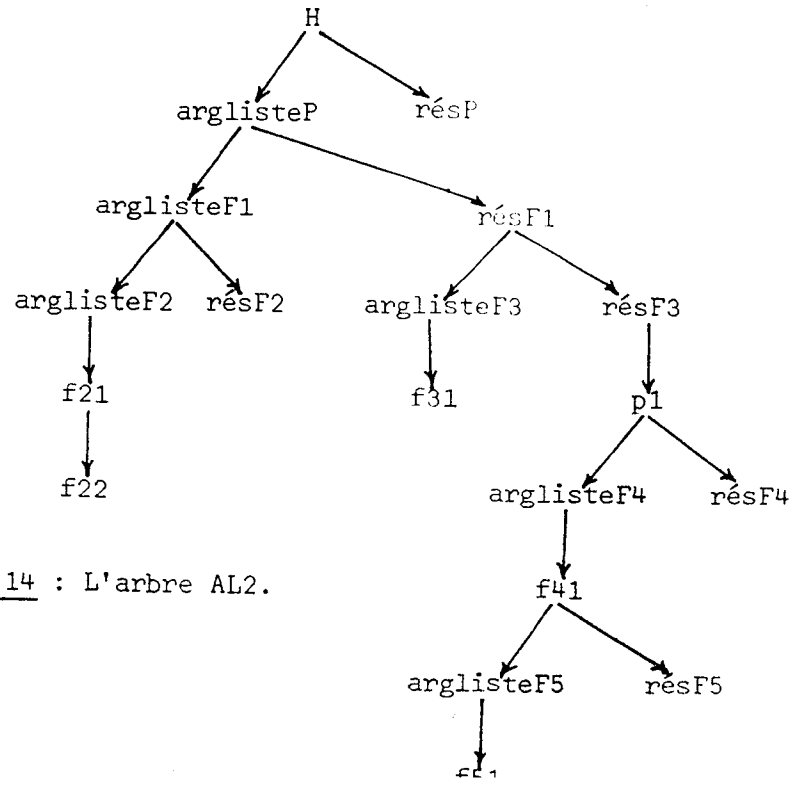


Figure 14 : L'arbre AL2.

On peut facilement vérifier qu'il est possible d'établir une version analogue-relative à l'algorithme A3 - de la Proposition 1 du souschapitre précédent.

L'arbre d'allocation - obtenu au moyen de l'algorithme A3 d'un arbre d'appel de racine F_i - privé du sommet résFi sera dénoté par sa'F_i.

Nous reprendrons l'exemple de l'arbre d'appel maximal A1 du souschapitre précédent : en supposant que la version optimale de A1 est toujours A1' (c'est-à-dire que taille (sa'F_1) ≥ taille (sa'F_3) ≥ taille (sa'F_4)) nous obtiendrons l'arbre d'allocation AL2' à taille minimale donné par la figure 15.

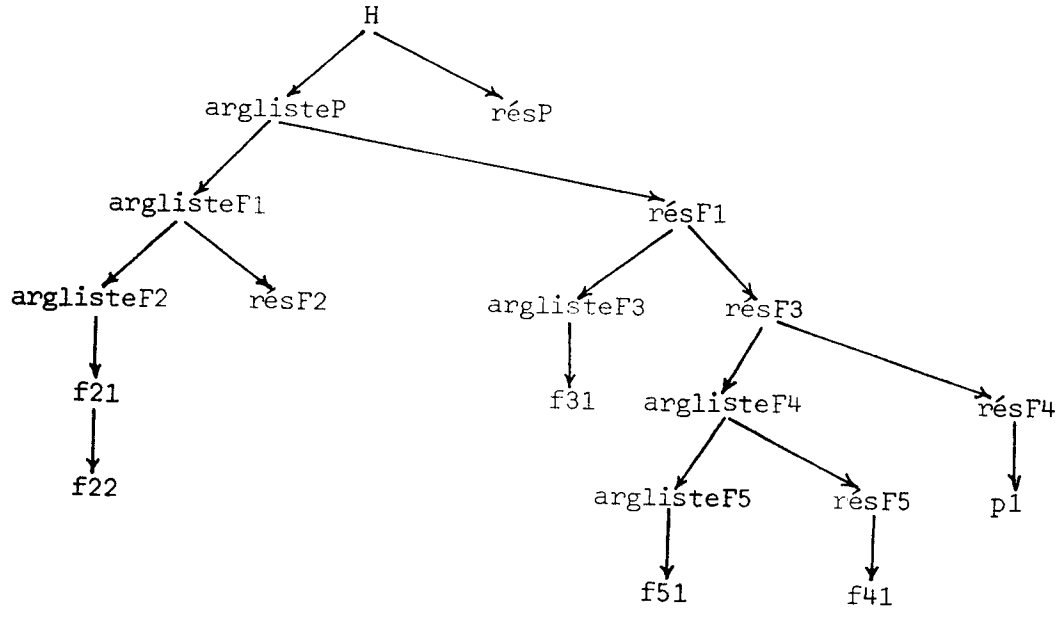


Figure 15 : L'arbre AL2'.

c) - Comparaison des solutions fournies par les algorithmes A2 et A3 -

Proposons-nous d'abord de comparer les tailles des arbres d'allocation AL2 - obtenu au moyen de l'algorithme A2 de l'arbre d'appel A02 optimal par rapport à la taille, relativement à A2 et où A02 est la version optimale d'un arbre d'appel maximal A - et AL3 - obtenu au moyen de l'algorithme A3 de l'arbre d'appel A03 optimal par rapport à la taille, relativement à A3 et où A03 est la version optimale de A.

Cas 1 : Profondeur = 1.

L'arbre A, donné par le schéma de la figure 16

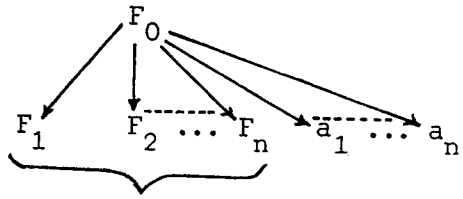


Figure 16.

produit, au moyen de A2 et A3, les deux arbres d'allocation donnés par la figure 17.

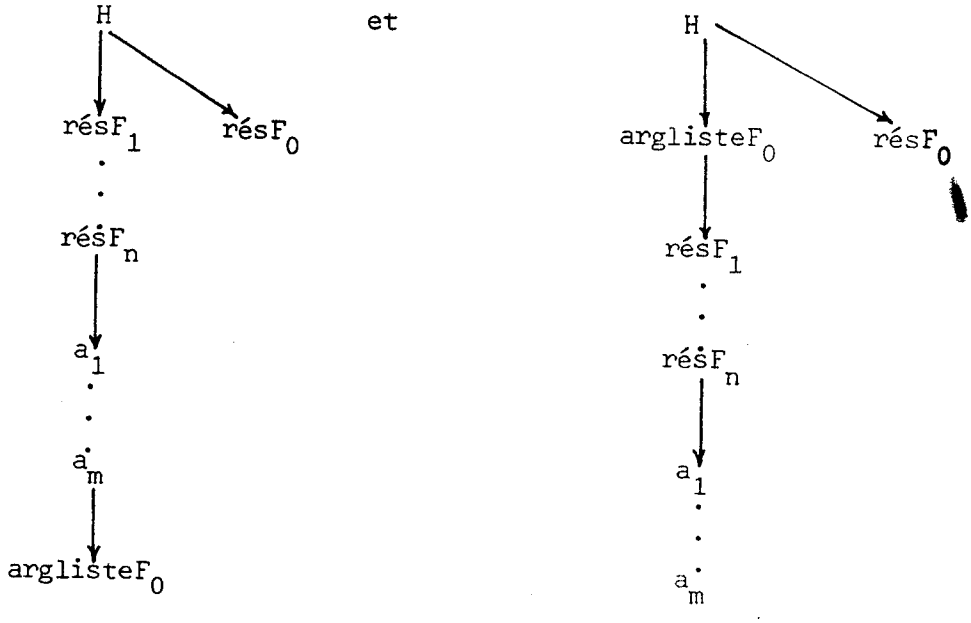


Figure 17.

Quel que soit l'ordre des fils de F_0, les deux arbres d'allocation sont de la même taille, d'où aussi :

$$\text{taille } (saF_0) = \text{taille } (sa'F_0).$$

Cas 2 : Profondeur = 2.

L'arbre A est donné par le schéma de la figure 18.

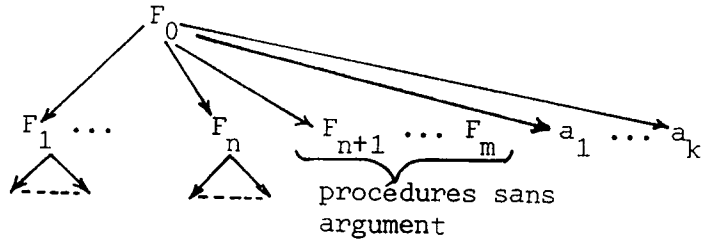
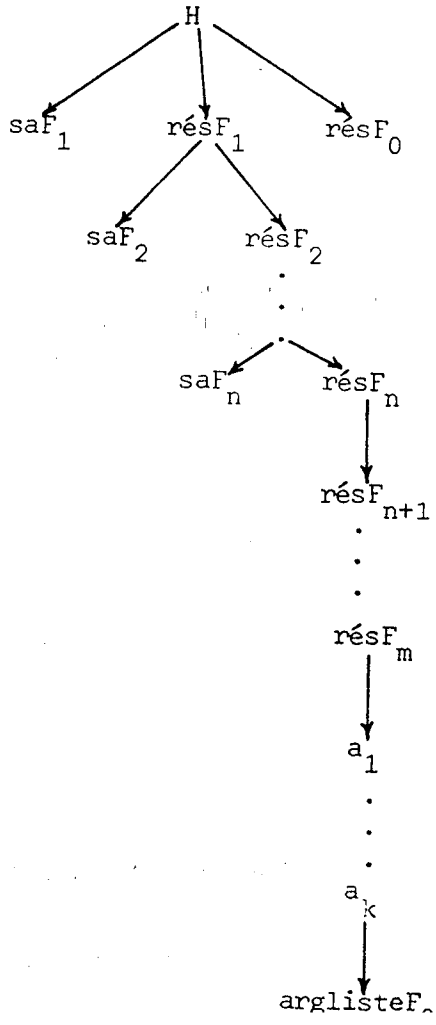


Figure 18 : L'arbre A.

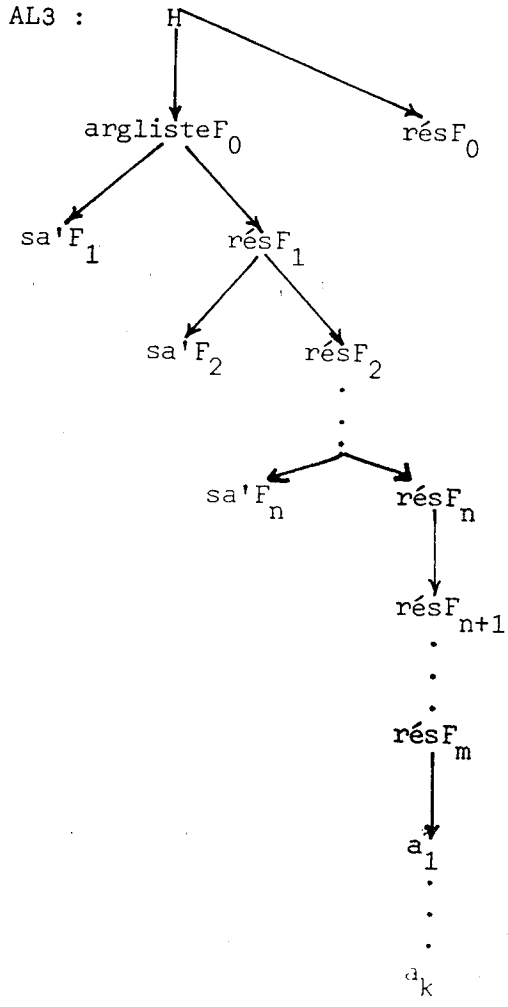
Les sousarbres de racines F_1, ..., F_n sont de profondeur = 1 donc
$\forall i$, i = 1, ... n : taille ($sa(F_i)$) = taille ($sa'(F_i)$).

Il en résulte que A02 = A03. En supposant que ces deux arbres optimaux sont représentés par le schéma de A de la figure 18 on déduit les schémas de AL2 et de AL3 donnés par la figure 19.

AL2 :



AL3 :



Il est facile de voir que la taille (AL2) ≤ taille (AL3) d'où également : taille (saF_0) ≤ taille ($sa'F_0$).

Cas 3 : Profondeur > 2.

Aucune comparaison des tailles définies par AL2 et AL3 n'est possible, les arbres A02 et A03 étant, en général, différents.

Cherchons maintenant à comparer les deux arbres d'allocation AL2 et AL3 à tailles minimales en termes du temps d'exécution relatif à l'évaluation de leurs arbres d'appel A02 et A03 optimaux.

Evaluer A0i (i=2, 3) c'est établir les valeurs des arguments, les listes d'arguments et effectuer des appels. On constate, au vu des schémas d'allocation de AL2 et AL3 que les sommets F_{i_1}, ... F_{i_n} d'un sousarbre d'appel SA de A0i, soit :

$$SA = F_{i_0}(F_{i_1}(\ldots), \ldots, F_{i_n}(\ldots), F_{i_{n+1}}, \ldots, F_{i_m}, a_{j_1}, \ldots, a_{j_k})$$

doivent être évalués dans cet ordre ; suit l'évaluation - dans un ordre a priori quelconque - des feuilles de SA et l'évaluation de F_{i_0}. Un parcours en ordre postfixé des sommets de A0i sera donc un ordre d'évaluation correct par rapport aux allocations AL2 et AL3. Toutefois une distinction entre les allocations définies par AL2 et AL3 apparaît lorsqu'on considère l'évaluation d'un sommet F_{i_0} (représentant un argument d'appel ou la racine de A0i) : cette évaluation consiste en l'établissement de la liste des arguments associée et en l'appel proprement dit ; nous pouvons constater, en effet, que l'allocation AL3 permet - alors que l'allocation AL2 peut ne pas le permettre - l'établissement d'une liste d'arguments au fur et à mesure que les arguments correspondants sont évalués.

L'établissement d'une liste d'arguments consiste en le rangement de l'adresse de l'argument (s'il est passé par référence) ou de l'adresse de l'argument intermédiaire correspondant (si l'argument est passé par "dummy") ou de la valeur de l'argument (s'il est passé par valeur) dans la liste. Il est préférable - pour éviter d'éventuelles sauvegardes des registres - de pouvoir effectuer le rangement de la valeur d'un argument (passé par valeur) au moment où cette valeur est disponible ; ceci n'est possible, en général, qu'avec l'adoption de l'allocation définie par AL3.

Il en résulte qu'en termes du temps d'exécution relatif à l'évaluation de l'arbre d'appel A l'allocation définie par AL3 est préférable à celle définie par AL2.

d) - Allocation des données générées par des arbres d'appel maximaux -

Dénotons par dga les données générées (arguments intermédiaires, listes d'arguments) par des arbres d'appel maximaux.

Nous considérerons d'abord l'allocation des dga au niveau d'une expression-source (englobant les appels, les opérations standard et l'indexation) lorsque celle-ci comporte plusieurs occurrences d'appels donnant lieu à plusieurs arbres d'appel maximaux.

Nous supposerons que les expressions-source informellement définies ci-dessus sont évaluées sériellement (les unes par rapport aux autres). L'allocation des dga résultant d'une même expression-source peut dépendre de la façon dont cette expression est évaluée. Il faut par conséquent faire des hypothèses quant aux ordres admissibles d'évaluation des sommets d'un arbre (syntaxique) d'une expression-source.

L'arbre d'une expression-source, considéré aussi comme un graphe orienté, est composé de deux genres de sous-graphes (qui sont des arbres) : ceux représentant les arbres d'appels maximaux (si l'expression comporte des occurrences d'appels) - dénotés apm - et ceux dont les racines sont des opérateurs standard ou l'opérateur select, expliqué par la suite, et dont les sommets Fi ne peuvent apparaître qu'en position de feuilles - dénotés sg. Les sg considérés sont maximaux c'est-à-dire disjoints deux à deux. La première hypothèse concerne les arbres apm : nous les supposerons optimaux par rapport à la taille, relativement à l'algorithme A3 ; leurs arbres d'allocation sont donc déduits par l'algorithme A3. L'évaluation des apm - en ordre postfixé - sera celle esquissée au souschapitre précédent. La seconde hypothèse concernera l'ordre d'évaluation des arbres sg.

Définition : l'évaluation des sommets d'un arbre sg est faite dans un ordre raisonnable si, ayant évalué l'un des sommets-fils d'un sommet s_i (un opérateur) de sg_j on ne peut passer à l'évaluation d'un sommet-fils d'un sommet s_k de sg_j - $s_k \neq s_i$ et il n'existe pas de chemin de s_i à s_k - qu'après l'évaluation de tous les autres sommets-fils de s_i.

Considérons l'exemple de la figure 20 :

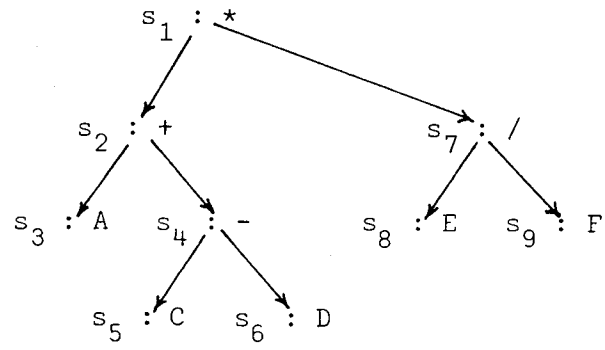


Figure 20.

si la première feuille évaluée est s_5 les ordres raisonnables possibles d'évaluations des sommets sont :

$s_5, s_6, s_4, s_3, s_2, s_8, s_9, s_7, s_1$ et

$s_5, s_6, s_4, s_3, s_2, s_9, s_8, s_7, s_1$.

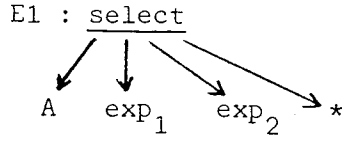
Un ordre non raisonnable est par exemple :

$s_5, s_3, s_8, s_6, s_4, s_9, s_7, s_2, s_1$.

Nous supposerons que, pour évaluer un sg, on choisira une évaluation des sommets de sg dans un ordre raisonnable (notons, par exemple, que l'évaluation postfixée est dans un ordre raisonnable).

Nous supposerons en outre que l'ordre raisonnable effectivement choisi sera basé sur des considérations relatives à la disponibilité des registres, en vue d'une diminution du temps d'exécution, et non pas sur des considérations de minimisation de l'espace nécessaire pour l'allocation des dga résultant d'une expression-source.

Le choix exposé ci-dessus nous dictera la façon dont devront être allouées les dga résultant de l'arbre E d'une expression-source. Leur allocation sera décrite par un arbre d'allocation - noté al(E) - que nous définirons par la suite. Nous supposons, dans le cadre de ce chapitre, que l'arbre d'une expression-source est de la forme illustrée par les exemples de la figure 21.



représente un soustableau A(exp$_1$, exp$_2$,

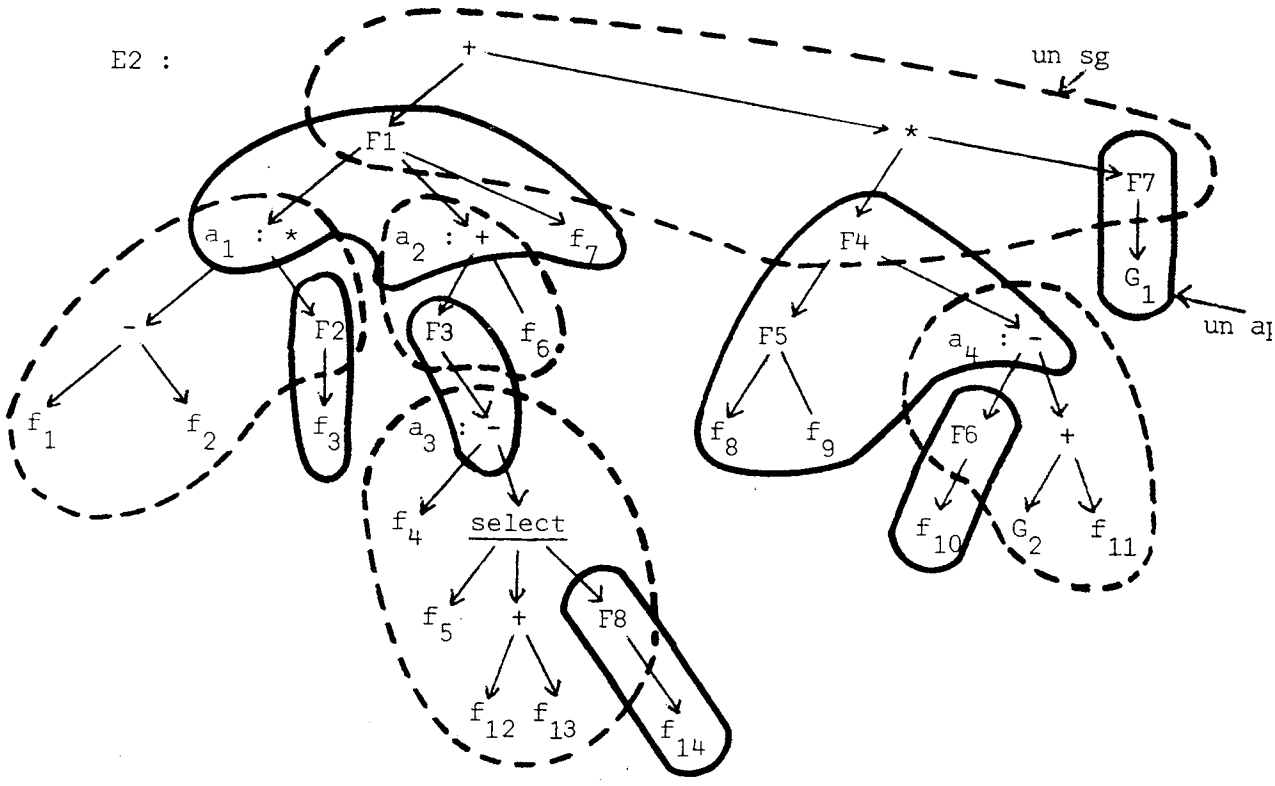


Figure 21.

Notons que l'arbre d'une expression peut avoir pour racine la racine d'un apm.

Soit E l'arbre d'une expression-source. L'ensemble des sommets de E sera également dénoté par E.

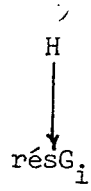
Nous dénoterons certains sommets de E de la façon suivante :

- F_i dénote un nom de procédure ayant au moins un argument ;
- les feuilles de E sont dénotés soit par G_i - qui dénote un nom de procédure sans argument - soit par f_i.

En outre, toute feuille s d'un apm de E qui n'est pas du type G_i ou f_i sera dénotée par :

- a_i si le sousarbre - de E - de racine s comporte au moins un sommet F_i et
- b_i sinon.

Notons que les apm d'un seul sommet (du type G_i) ne sont pas considérés : l'arbre d'allocation d'un tel apm est :



et la donnée générée $résG_i$ est considérée comme un temporaire, non une dga.

Dénotons par :

- ar(s) : le sousarbre (de E) de racine s ;
- $apmF_i$, $F_i \in E$: l'arbre apm de racine F_i ;
- $ar(apmF_i) = ar(F_i)$, $F_i \in E$;
- al(ar(s)), $s \in E$ et s n'est pas du type F_i : l'arbre d'allocation (qu'on définira) représentant l'espace alloué pour les dga correspondant aux apm qui sont des sousgraphes de ar(s) ;
- $al(ar(apmF_i))$, $F_i \in E$: l'arbre d'allocation - qu'on définira - représentant l'espace alloué pour les dga correspondant aux apm de $ar(apmF_i)$;
- $al(apmF_i)$, $F_i \in E$: l'arbre d'allocation de $apmF_i$;
- F(s), $s \in E$ et s n'est pas du type F_i : l'ensemble de sommets du type F_i de E tel que :

  $\forall F_j \in F(s)$, $\exists$ un chemin de s à F_j dans E, ne passant par aucun autre sommet du type F_i.

Définissons maintenant, d'une façon interdépendante, les arbres al(ar(s)) et $al(ar(apmF_i))$:

1) - Si un $apmF_i$, $F_i \in E$, ne comporte pas de feuilles du type a_i alors :

$$al(ar(apmF_i)) = al(apmF_i)$$

2) - Un arbre al(ar(s)) est donné par le schéma de la figure 22.

$(F(s) = \{F_{j_1}, \ldots, F_{j_n}\})$:

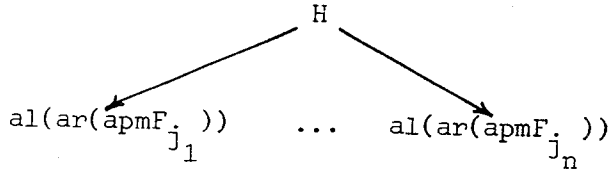


Figure 22.

3) - Si un $apmF_i$, $F_i \in E$ comporte des feuilles du type a_i l'arbre al(ar($apmF_i$ sera obtenu par une modification de l'arbre al($apmF_i$). A tout sommet F_{j_0} de $apmF_i$ correspond, dans al($apmF_i$), le schéma (voir le souschapitre c) de la figure 23

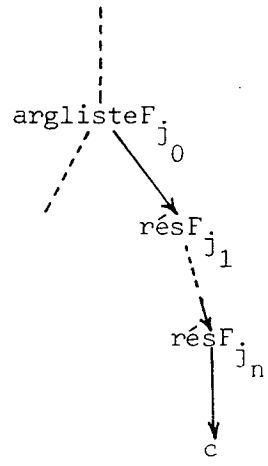


Figure 23.

où c désigne un chemin quelconque, composé des sommets du type G_i, a_i, b_i et f_i qui sont des sommets-fils, dans E, de F_{j_0}. Nous ordonnerons les sommets de c de la façon suivante :

- tous les sommets du type a_i de c précèdent tous les autres sommets de c et

- $\forall a_i \in c, \forall a_j \in c$; a_i précède $a_j \iff$

taille (al(ar(a_i))) $\geq$ taille (al(ar(a_j))).

L'arbre al(ar($apmF_i$)) est alors obtenu en remplaçant tout schéma ci-dessus (voir la figure 23), correspondant à un F_{j_0} de $apmF_i$, par le schéma de la figure 24.

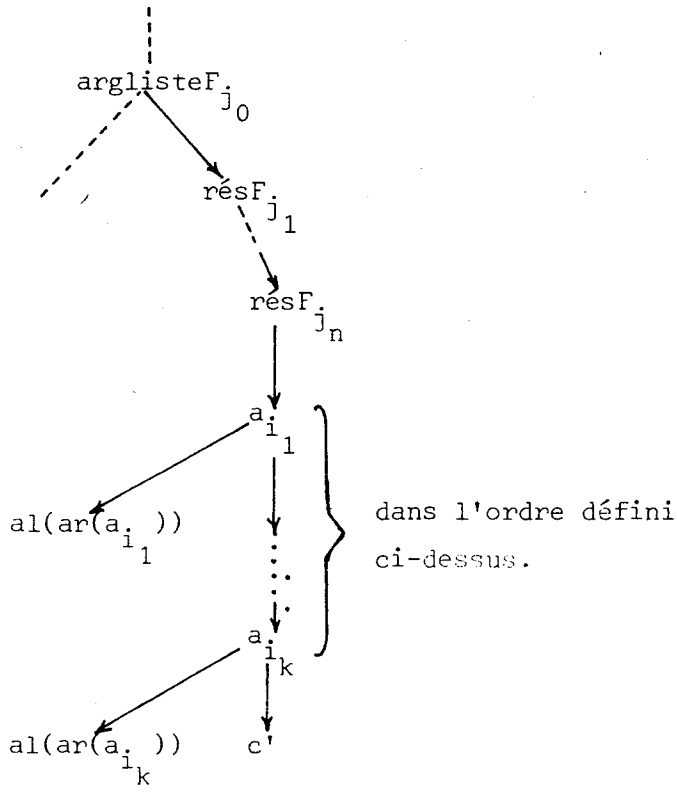


Figure 24.

On peut finalement définir al(E) :

- si la racine R de E est du type F_i : al(E) = al(ar(apmR))
- sinon : al(E) = al(ar(R)).

On dénotera par dge la donnée générée dont l'allocation est définie par un arbre d'allocation al(E) relatif à une expression-source d'arbre E.

Exemple : reprenons l'arbre d'expression E2 donné plus haut (figure 21) ; son arbre d'allocation al(E2)-en supposant que taille ($a\ell(ar(a_1))$) ≥ taille ($a\ell(ar(a_2))$)- est donné par la figure 25.

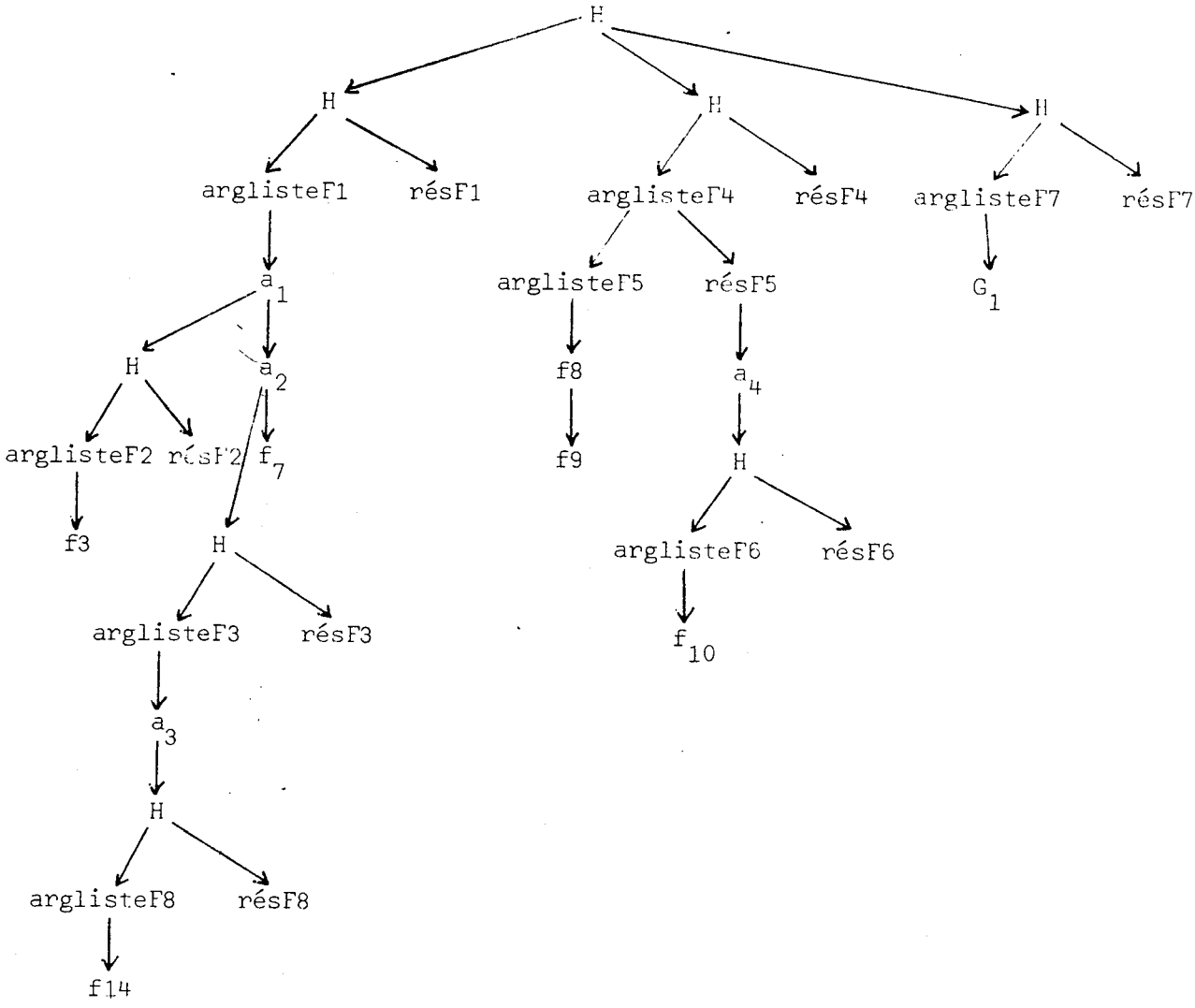


Figure 25.

Les dge correspondant aux occurrences des expressions-source d'un fief f seront allouées parallèlement - les expressions-source étant évaluées sériellement - et constitueront la zone des arguments de f. Le schéma de l'arbre d'allocation d'une zone des arguments est donné par la figure 26.

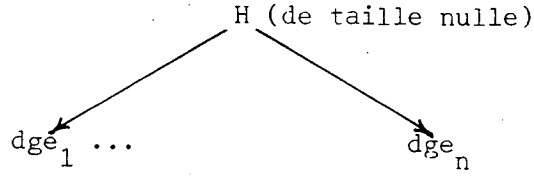


Figure 26.

L'allocation des zones des arguments des fiefs d'un corps de procédure peut se faire de plusieurs façons. Nous choisirons la solution qui semble la plus simple et qui incorpore les zones des arguments dans le schéma de recouvrement défini par ASFP : la zone des arguments du fief f sera allouée à la suite de la zone des temporaires de f. Cette solution est certainement supérieure (du point de vue de l'espace occupé) à celle qui consiste à allouer parallèlement toutes les zones des arguments d'un corps de procédure cp et ce à la suite de l'espace initial de la zone locale correspondant à cp.

L'espace des fiefs pourra ainsi être représenté par le schéma de l'arbre d'allocation de la figure 27 (à comparer à la figure 1 de 5.4.2.1).

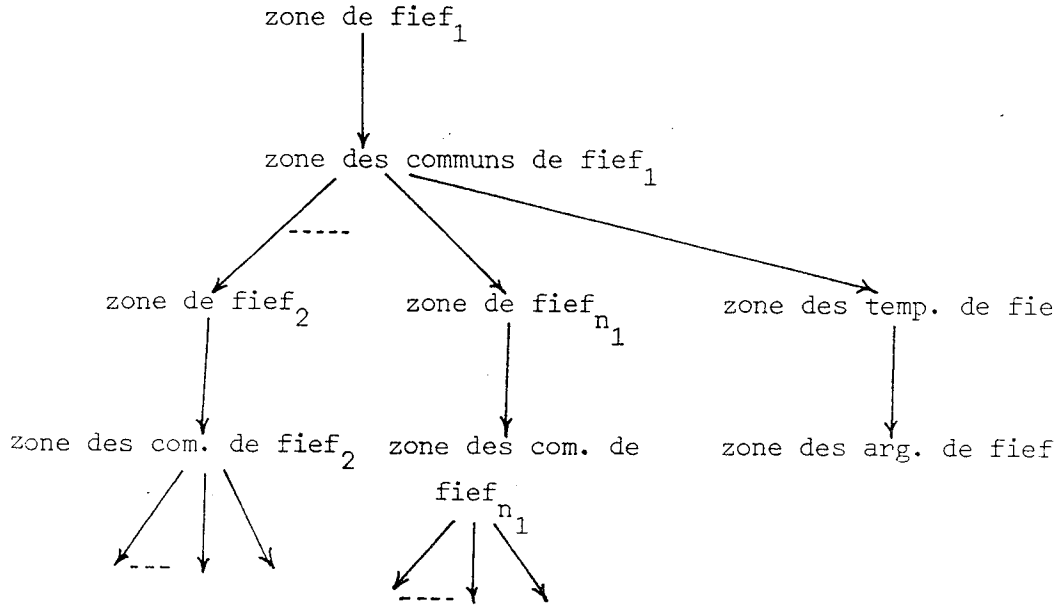


Figure 27.

6 - LANGAGE INTERMEDIAIRE -

6.1 - Présentation -

Nous introduirons dans ce chapitre un langage intermédiaire qui servira de base aux algorithmes d'optimisation.

Ce langage intermédiaire (noté LI) devra par conséquent avoir les caractéristiques suivantes :

- il devra pouvoir expliciter les opérations implicites au niveau du langage-source et susceptibles d'être optimisées ;
- il devra être simple pour que les programmes intermédiaires soient facileme manipulables par des algorithmes d'optimisation ;
- il devra bien refléter la structure des programmes-source.

Les deux premières caractéristiques impliquent un langage de bas niveau : les instructions du langage comporteront donc un seul opérateur ; les opérandes, en règle générale, corresponderont aux données simples ; les seules instructions spécifiant explicitement des transferts de contrôle seront des sauts conditionnels et inconditionnels. Les opérations "administratives", telles la gestion de la mémoire et la gestion de l'environnement ne seront pas explicitées au niveau d'un programme en LI.

Nous supposerons que toutes les informations relatives à la sémantique statiq d'un programme-source (cf. informations déduites, ch. 4 et 5) sont extraites c calculées au cours de la phase d'analyse syntaxique d'un programme-source ; par conséquent un programme en LI ne comportera pas d'instructions déclaratives.

La troisième caractéristique semble incompatible avec la seconde. Elle n'est pas indispensable : on peut, en effet, reconstruire la structure relative aux transferts de contrôle du programme-source à partir des instructions de saut du programme intermédiaire. Toutefois ceci exige un travail supplémentaire qu'il est possible d'éviter, du moins partiellement, si l'on garde des informations sur la structure du programme-source : il s'agit ici de la structure des blocs et des boucles DO (la structure des blocs est utile en ce que nous savons qu'il est impossible d'entrer dans un bloc sans passer par sa tête). Cette structure sera donc exprimée dans un programme intermédiaire à l'aide des instructions spéciales.

L'hypothèse qui consiste à supposer que les programmes-source sont bien structurés (au sens de Dijkstra) nous conduit également à garder au niveau du programme intermédiaire la structure du programme-source. Ceci permet à la limite - pour un langage-source sans sauts explicites comme Bliss, par exemple - de connaître, au niveau de l'analyse, toute la structure relative aux transferts de contrôle.

Les considérations qui précèdent nous ont mené à une décomposition du programme-source, décomposition qui reflète l'inclusion statique des blocs PROCEDURE et BEGIN et des boucles DO. En reprenant la terminologie introduite dans le chapitre 4 nous décomposerons un programme-source selon ses fiefs de portée et ses fiefs des boucles. Au texte d'un fief f correspondra un texte intermédiaire (c'est-à-dire une partie du programme intermédiaire) dans lequel les fiefs successeurs de f - relativement à l'arbre ASF d'un corps de procédure - seront représentés par des instructions spéciales (appelées instructions de structure) qui permettrons de reconstruire la structure des blocs et des boucles DO d'un corps de procédure.

Comme nous supposerons avoir déterminé les informations déduites décrites dans le chapitre 4 (le GPA, etc...) il sera inutile de représenter, au niveau d'un programme intermédiaire, la structure qu'exprime l'arbre ASC relatif à l'inclusion statique des corps des procédures d'un programmme-source. Au niveau d'un programme intermédiaire les ensembles de textes intermédiaires, ensembles correspondant aux différents corps des procédures du programme, seront ainsi traités indépendamment.

Dans les sous-chapitres suivants nous décrirons le langage intermédiaire LI et nous exposerons la structure des programmes intermédiaires.

La traduction de SPL/1 en LI n'a pas été réalisée.

6.2 - Description du langage intermédiaire LI -

Nous décrirons ici le langage intermédiaire LI en tâchant d'éviter un formalisme trop lourd.

Nous indiquerons en même temps la correspondance entre les instructions de SPL/1 et celles de LI.

6.2.1 - Eléments du langage LI -

Un programme intermédiaire est un ensemble de textes intermédiaires (cf. 6.1). Un texte intermédiaire est une suite d'instructions ; une instruction comporte une étiquette en option, un mot-clé, plusieurs opérandes-source (en option) et un opérande-résultat (en option).

Nous distinguerons les instructions suivantes :

- instructions arithmétiques, logiques et de conversion ;
- instruction de rangement ;
- instructions d'appel ;
- instructions de transfert ;
- instructions de structure ;
- instructions d'entrée-sortie ;
- instructions auxiliaires.

Les instructions des trois premières classes peuvent avoir la forme des instructions d'affectation ; une instruction d'affectation comporte le symoble terminal → ; l'opérande à droite de → est la destination de l'affectation.

Les opérandes des instructions correspondent, en règle générale, aux données simples. Les noms des opérandes qui correspondent aux identificateurs-source d'un programme-source sont directement déduits de ces derniers si l'on suppose que les identificateurs-source sont uniques.

Nous distinguerons les opérandes suivants :

- opérandes simples : ils correspondent aux identificateurs-source identifiant des données simples et aux données simples générées par l'implémentation de SPL/1 (cf. ch. 5) (bornes et multiplicateurs des tableaux, constantes) et explicitement manipulées au niveau de LI. La plupart des données générées par l'implémentation ne sont pas explicitement manipulées : déplacements initiaux des tableaux, éléments des vecteurs d'environnement et des zones de pointeurs, différents pointeurs utilisés dans la gestion de l'environnement et de la mémoire, les deux composants adresse-code et environnement d'une variable LABEL ou d'un paramètre ou argument ENTRY, etc.... Les paramètres et les arguments ENTRY sont par conséquent considérés comme des opérandes simples.

Dans la syntaxe ou dans les exemples les opérandes simples seront notés par : a, b, c, ...

- temporaires (cf. 5.4.2.) : ils correspondent aux résultats intermédiaires simples introduits par l'évaluation des expressions et aux valeurs délivrées par les procédures-fonctions. Un temporaire ne peut apparaître, en tant que destination, que dans une seule occurrence d'une instruction d'affectation d'un programme intermédiaire.

Notation : t_1, t_2, t_3, ...

- étiquettes : les opérandes-étiquettes correspondent aux étiquettes constantes et variables du programme-source et aux étiquettes générées par le compilateur. Notation : ℓ, ℓ_1, ...

- tableaux : les opérandes-tableaux apparaissent dans les instructions d'appel pour désigner des arguments qui sont des tableaux (tableaux-source ou intermédiaires, cf. 5.3.8) ; ils apparaissent également en tant que "bases" dans les opérandes indéxés et dans les sous-tableaux.

Notation : - ta, tb, tc,... ;

- ta ($\begin{Bmatrix} \text{opd} \\ * \end{Bmatrix}$, $\begin{Bmatrix} * \\ \text{opd} \end{Bmatrix}$) ;

La deuxième forme correspond aux sous-tableaux et comporte au moins un astérisque ; opd - appelé indice - y désigne un opérande simple ou un temporaire et ta est la base du sous-tableau.

Les tableaux intermédiaires (cf. 5.3.8) sont des tableaux temporaires ; notation : tt, tt_1, ...

- opérandes indéxés : ils correspondent aux éléments des tableaux. Notation : ta(opd), ..., tt(opd), ... où opd - appelé indice - est un opérande simple ou un temporaire et où ta et tt sont des bases des opérandes indéxés. Un indice qui est un opérande simple peut correspondre à une constante évaluée à la compilation.

Les bases des opérandes indéxés désignent les adresses initiales des tableaux correspondants et les indices désignent les "déplacements" par rapport aux adresses initiales ; un indice correspond donc à la valeur de l'expression $\sum_j m_j * ind_j$ (cf. 5.3.8).

Appelons opérande élémentaire un opérande simple ou indéxé ou un temporaire ; appelons opérande non-indéxé un opérande simple ou un temporaire.

6.2.2 - Instructions arithmétiques, logiques et de conversion -

Ces instructions correspondent aux opérations standard et sont des instructions d'affectation : la partie-source - qu'on appellera expression - est une opération unaire ou binaire et la destination est un temporaire. On dira que l'expression qui est la partie-source est associée au temporaire qui est la destination.

Syntaxe : <mot-clé> <opd.élém.> → t |
<mot-clé> <opd.élém.>, <opd.élém.> → t.

Les mots-clés désignent les opérations standard :

- opérations arithmétiques : plus, moins, div, divent (division entière), mod (avec mod a, b = moins a, (mult (divent a, b) b)), mult, exp, neg (le moins unaire) ;
- opérations logiques : non, et, ou ;
- opérations de comparaison : pp (plus petit), npp (¬<), ppe (<=), égal, négal (¬=), pg (>), npg (¬>), pge (>=) ;
- opérations de conversion : entier (unaire : conversion d'un opérande FLOAT DEC(6) en un résultat BIN FIXED (31)), flot (unaire : conversion de BIN FIXED (31) en FLOAT DEC (6)).

Au niveau de LI les conversions sont explicitées.

6.2.3 - Instruction de rangement -

Syntaxe : <opd élém.> → <opd. simple ou indéxé>.
L'instruction de rangement correspond à l'instruction d'affectation de SPL/1. De ce qui précède (définition d'un opérande temporaire, les instructions de 6.2.2., l'instruction de rangement) s'ensuit une séparation entre les évaluations des expressions et les mouvements des données dans la mémoire. Cette séparation simplifiera par la suite la suppression des expressions redondantes.

6.2.4 - Instructions d'entrée-sortie -

Syntaxe : lire {<opd. simple ou indéxé>,}* <opd. simple ou indéxé> |
écrire {<opd. élém.>,}* <opd. élém>.

Ces instructions correspondent à GET LIST et PUT LIST de SPL/1 ; elles n'opèrent toutefois que sur les données simples.

6.2.5 - Instructions de transfert -

Syntaxe : <instr. de transfert>::=<instr. de saut>|stop|retour|retour<opd. élém.>.
<instr.de saut>::=<instr.de saut conditionnel>|saut<étiquette>.
<instr.de saut conditionnel>::=sautcond<condition>, <étiquette>|
boucler<condition>.

Les instructions de saut correspondent aux instructions GOTO, IF et aux boucles DO (c'est-à-dire instructions DO répétitives) de SPL/1. La condition des sauts conditionnels est une expression comportant un opérateur standard logique ou de comparaison.

Si la condition d'une instruction sautcond ou d'une instruction boucler est fausse l'exécution se poursuit en séquence ; sinon (si la condition est vraie) l'exécution se poursuit à l'étiquette indiquée dans le cas d'une instruction sautcond et à la première instruction du texte intermédiaire courant dans le cas d'une instruction boucler (voir aussi 3.2.7 et les exemples de 3.2.9).

Notons qu'une instruction de saut peut entraîner implicitement un désempilage (cf. ch. 5).

L'instruction stop correspond aux instructions suivantes de SPL/1 : STOP, RETURN s'il s'agit d'un retour du corps de la procédure MAIN, et END s'il s'agit de la fin de la procédure MAIN.

L'instruction retour correspond aux instructions RETURN et END relatives à des procédures-sousprogrammes.

L'instruction retour <opd.élém.> correspond à une instruction RETURN (expression). Notons (cf. ch. 5) que la valeur du résultat d'une procédure-fonction est retournée dans l'un des deux registres réservés à cet effet.

Notons également (cf. ch. 5) que les instructions stop et retour [<opd.élém.>] peuvent implicitement entraîner des actions en épilogue ; l'instruction END relative à une procédure-fonction n'est pas traduite.

6.2.6 - Instructions d'appel -

Syntaxe : <nom de fonction> <liste des arguments>→t|
<nom de sousprogramme> <liste des arguments>.

La première forme correspond à un appel de procédure-fonction (c'est une instruction d'affectation) et la seconde à un appel de procédure-sousprogramme (CALL de SPL/1). La partie-source de la première forme est une expression qui est associée (cf. 6.2.2) à la destination t.

<liste des arguments>::=<arg>, <liste des arguments>|<arg>|∅.
<arg>::=<opd.élém.>|val <opd.élém.>|
dum <opd.simple ou indéxé>|
<opérande-tableau>|
dum <opd.-tableau non temporaire>.

<nom de fonction> et <nom de sousprogramme> sont des mots-clés secondair et désignent ainsi des opérations non-standard.
Lorsqu'un argument d'un appel - au niveau de SPL/1 - est une expression comportant des opérateurs nous explicitons l'évaluation de cette expression.

A un argument qui est une donnée simple peut correspondre (au niveau de LI) :

- un opérande simple ou indéxé si l'argument est passé par référence ;
- un temporaire si l'argument, au niveau de LI, correspond à une expression comportant un opérateur standard ou non-standard (l'argument est passé par "dummy") ;
- un opérande simple ou indéxé précédé du spécificateur dum si l'argument est passé par "dummy" et est, au niveau de SPL/1, une constante ou un identificateur d'une donnée simple entre parenthèses ou un élément de tableau entre parenthèses (ex. : 2, (A), (B (1, 2)), ...) ;
- un opérande élémentaire précédé du spécificateur val si l'argument est passé par valeur (cf. 5.3.3).

Un problème de degré d'explicitation se pose pour les arguments qui sont des données composées. Le passage de ces arguments peut entraîner des opérations complexes : pour les synonymes des tableaux et les sous-tableaux (cf. 5.3.8) il est nécessaire de construire des vecteurs d'accès s'ils sont passés par référence ; de plus, pour les données composées passées par "dummy" il faut allouer en pile les tableaux intermédiaires correspondants.

Nous avons choisi de ne pas expliciter le calcul de la taille à allouer et l'allocation en pile d'un tableau intermédiaire ni l'établissement des vecteurs d'accès. Ainsi, à un argument qui est une donnée composée peut correspondre :

- un opérande-tableau ou un opérande-sous-tableau (cf. 6.2.1) si l'argument est passé par référence ;
- un opérande de la même forme qui ci-dessus précédé du spécificateur dum si l'argument est passé par "dummy" et est, au niveau de SPL/1, un identificateur de tableau ou un sous-tableau entre parenthèses
(ex. : (A), (A (*, 2)), ...) ;
- un opérande-tableau intermédiaire si l'argument, au niveau de SPL/1, est une expression de tableau comportant des opérateurs (ex. : A(1, *) + 2, ...) ou si l'argument nécessite une conversion.

Dans le cas d'un argument-tableau intermédiaire l'instruction d'appel de LI est précédée d'une instruction auxiliaire préparer tt et de l'évaluation de l'expression de tableau dont la valeur est affectée au tableau temporaire tt. L'instruction préparer tt correspond au calcul de la taille à allouer pour le tableau temporaire tt, à l'allocation en pile de tt et à l'établissement du vecteur d'accès intermédiaire de tt.

6.2.7 - Instructions de structure -

Rappelons qu'un programme intermédiaire est un ensemble de textes intermédiaires (cf. 6.1). Les textes intermédiaires correspondant aux fiefs d'un même corps de procédure seront reliés entre eux par des instructions de structure : la structure ainsi représentée sera celle de l'arbre ASF d'un corps de procédure.

Syntaxe : bloc <constante> |
boucle <constante> |
fief <constante> | sortie | fin

Une instruction bloc n d'un texte intermédiaire d'un fief f_k renvoie au texte intermédiaire du fief-début - texte intermédiaire identifié par fief n - qui est, en tant que sommet de ASF, le successeur immédiat de f_k. De même, une instruction boucle n d'un texte intermédiaire d'un fief f_k renvoie au texte intermédiaire d'un fief de boucle f_n, successeur immédiat de f_k dans ASF.

Une instruction bloc n ou boucle n, renvoyant au fief f_n représente par conséquent les textes intermédiaires des fiefs qui sont les sommets du sous-arbre de racine f_n de ASF. Appelons texte représenté l'ensemble des textes représentés par une instruction bloc ou boucle.

La traduction d'une boucle DO est expliquée dans les figures 1 et 2. La figure 1 donne le schéma du texte-source équivalent à une boucle DO [58]. La figure 2 donne la traduction en LI du texte-source de la figure 1 : le fief de boucle ne comporte pas les initialisations et les tests initiaux ; les tests relatifs à la répétition de la boucle sont dupliqués et les sauts correspondants à ces tests sont traduits par des instructions boucler (cf. 6.2.5). Nous exposerons les raisons de cette duplication des tests dans le chapitre suivant.

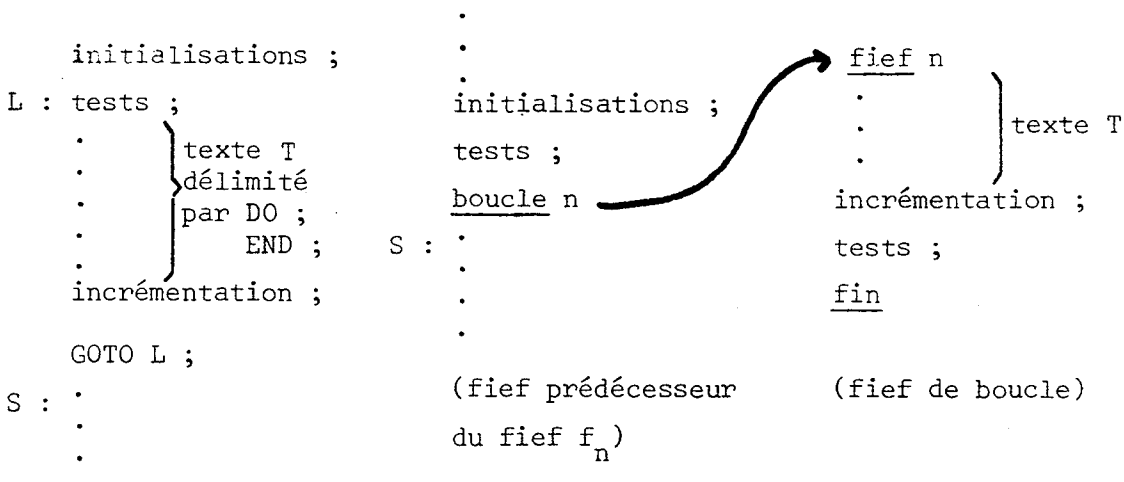


Figure 1.

Figure 2.

Des exemples de la traduction des boucles DO seront donnés en 6.2.9.

Plusieurs instructions sortie peuvent succéder à une instruction bloc ou boucle. A une instruction sortie succédant à une instruction bloc n ou boucle n d'un texte intermédiaire d'un fief f_k seront associées les informations déduites relatives à une partie du texte représenté par bloc n ou boucle n, partie qui se termine par un saut provoquant un transfert de contrôle en dehors du fief f_k ou à une étiquette préfixant une instruction du fief f_k.

Les instructions sortie seront expliquées plus amplement dans le sous-chapitre 6.3 et dans le chapitre suivant.

L'instruction fin termine le texte intermédiaire d'un fief et renvoie vers le texte intermédiaire du fief prédécesseur relativement à l'arbre ASF.

L'instruction fief n commence le texte intermédiaire d'un fief-début ou d'un fief de boucle ; la constante n identifie le fief en question.

Les instructions de structure sont des pseudoinstructions : elles ne génèrent pas de code à la compilation des programmes intermédiaires.

6.2.8 - Instructions auxiliaires -

Les instructions auxiliaires peuvent être conçues comme des macroinstructions représentant des séquences d'opérations "administratives" (gestion de l'environnement et de la mémoire, établissement des vecteurs d'accès, etc...), opérations qui ne sont pas explicitées au niveau de LI. Ces opérations correspondent essentiellement aux actions en prologue ou en épilogue (cf. ch. 5) et, en particulier, aux initialisations (cf. l'attribut INITIAL de SPL/1).

Syntaxe : prologue { <nom de fonction> | <nom de sousprogramme> | ø } |
épilogue |
préparer <tableau temporaire>.

Le texte intermédiaire d'un fief-procédure commence par une instruction prologue { <nom de fonction> | <nom de sousprogramme> } ; le texte intermédiaire d'un fief-début ou d'un fief de boucle commence par une instruction fief <constante> suivie éventuellement d'une instruction prologue s'il s'agit d'un fief-début.

Les instructions prologue correspondent aux instructions BEGIN et PROCEDURE de SPL/1. Les actions en prologue associées sont expliquées dans le ch. 5. Une action en prologue d'un fief-début peut être vide ; dans ce cas le texte intermédiaire associé ne comporte pas d'instruction prologue.

L'instruction épilogue correspond à l'instruction END fermant un bloc BEGIN de SPL/1 (voir 6.2.5 pour les autres cas de l'instruction END) ; elle doit correspondre à une action en épilogue non vide (cf. ch. 5.3.6).

L'instruction préparer a été expliquée en 6.2.6.

6.2.9 - Exemples -

Nous présenterons ici trois courts exemples de la traduction de SPL/1 en LI.

Le premier exemple (figures 3 et 4) comporte un seul texte intermédiaire et le second (figures 5 et 6) comporte deux textes intermédiaires correspondant à un même corps de procédure.

```
BC : PROCEDURE(N, M) RECURSIVE RETURNS (FIXED BIN(31)) ;
     DCL(N, M) FIXED BIN(31) ;
     IF M>0 THEN RETURN (BC (N-1, M-1)* N/M) ;
            ELSE RETURN(1) ;
     END BC ;
```

Figure 3.

```
     prologue bc
     sautcond npg m, k0, ℓ
     moins n, k1 → t1
     moins m, k1 → t2
     bc t1, t2 → t3
     mult t3, n → t4
     divent t4, m → t5
     retour t5
ℓ :  retour k1
     fin
```

Figure 4. : Le texte intermédiaire correspondant à la procédure-fonction de la figure 3.

```
POLY : PROCEDURE (X, A, N) ; DCL (X, VAL) FLOAT DEC(6),
       (N, I) FIXED BIN (31), A (*) FLOAT DEC(6) ;
       VAL = A(0) ;
       DO I = 1 BY 1 TO N ;
       VAL = VAL*X + A(I) ;
       END ;
       RETURN (VAL) ;
       END POLY ;
```

Figure 5. : Au corps de la procédure POLY correspondent deux fiefs : le fief de la procédure POLY et un fief de boucle.

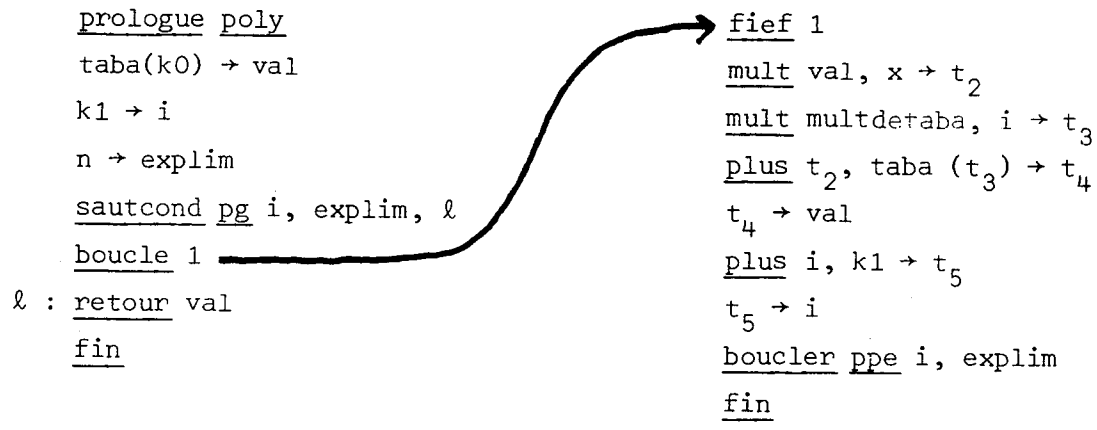


Figure 6. : Les deux textes intermédiaires correspondant à la procédure-fonction de la figure 5.

Les figures 7 et 8 donnent le schéma de la traduction des boucles DO imbriquées (voir les figures 1 et 2 de 6.2.7).

```
      DCL B(N) FIXED BIN (31) ;
      .
      .
      .
B1 :  DO I = 1 BY 1 TO N ;
B2 :  DO B(I) = 2 BY 1 TO M WHILE A>= D+C ;
      .
      .
      . (texte T)
      .
      .
        END B2 ;
        END B1 ;
      .
      .
```

Figure 7.

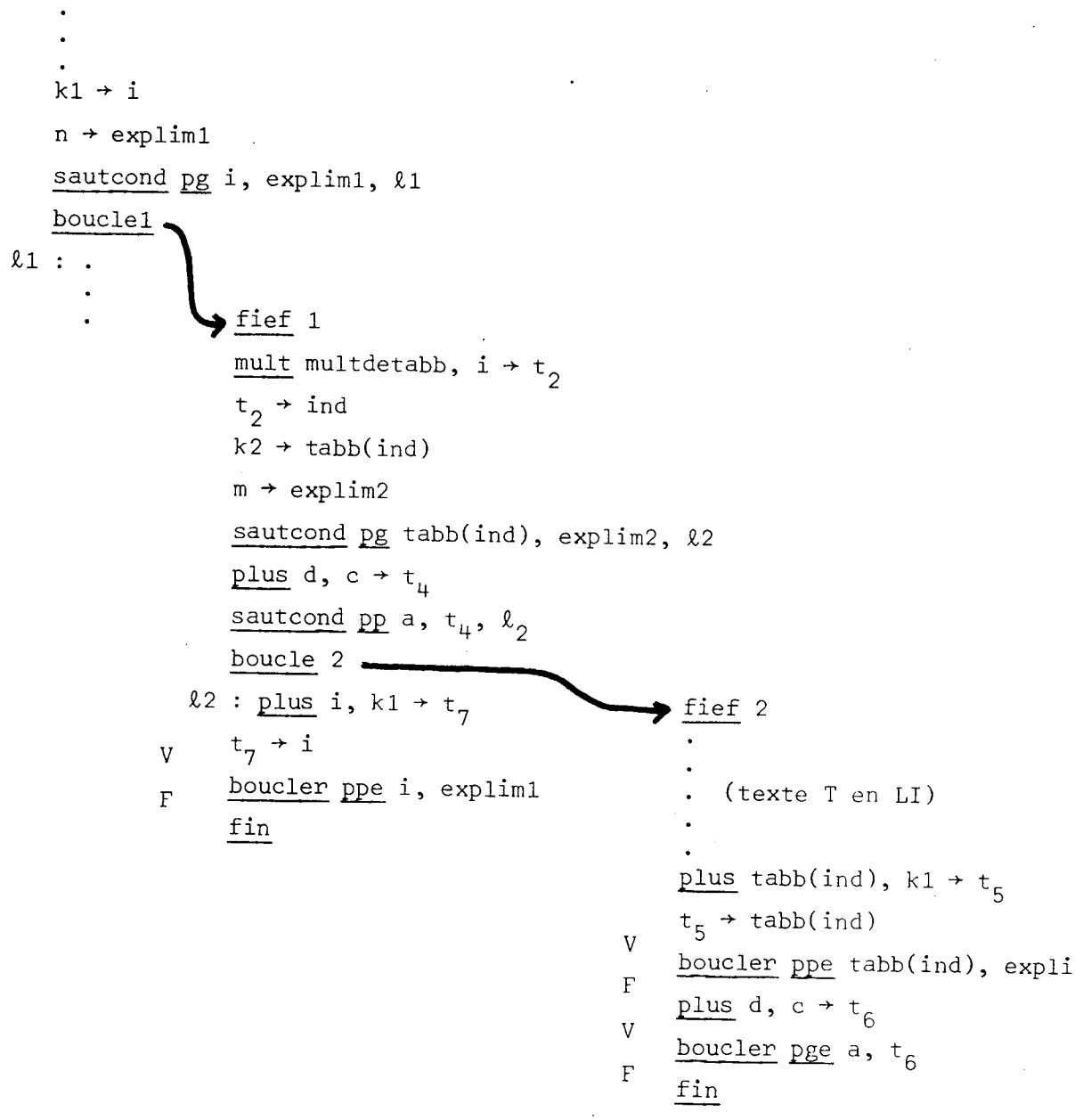


Figure 8. : La traduction en LI du texte de la figure 7. Les flèches indiquent les sauts relatifs aux instructions boucler.

6.3 - Sauts représentés par les instructions sortie -

Dans ce sous-chapitre nous expliquerons la signification des instructions sortie du langage intermédiaire et nous donnerons un algorithme (qui sera illustré par un exemple) effectuant l'insertion des instructions sortie dans un programme intermédiaire. Les instructions sortie ainsi insérées simplifieront, par la suite, l'établissement des graphes des programmes (cf. ch. 7).

On appellera étiquette locale à un texte intermédiaire (ou au fief correspondant) une étiquette qui préfixe une instruction de ce texte intermédiaire. Notons à ce propos qu'une étiquette-source préfixant une instruction BEGIN préfixe, au niveau de LI, l'instruction bloc (et non l'instruction fief) correspondante. Une étiquette qui n'est pas locale à un fief est globale à ce fief.

On appellera instruction de "saut" toute instruction de transfert saut, sautcond, stop, retour (d'un sousprogramme) et toute instruction d'appel d'une procédure-sousprogramme (notée nomdesspg).

A toute instruction de "saut" d'un texte intermédiaire T nous associerons deux listes : la liste des étiquettes locales à T (la liste locale) et la liste des étiquettes globales à T (la liste globale). Les éléments de ces deux listes représenteront les destinations possibles (locales et globales) des instructions de "saut". Dans le cas où l'opérande étiquette d'une instruction de saut est une étiquette constante étiq l'une des deux listes-associées à l'instruction de saut en question- est vide et l'autre contient un seul élément qui est étiq ; dans le cas où l'opérande étiquette est une étiquette variable etvar les éléments des deux listes donnent les valeurs (c'est-à-dire étiquettes constantes) possibles de etvar, valeurs qui sont fournies par le programme-source (cf. ch. 3).

La destination des instructions de transfert stop et retour (d'un sous-programme) sera dite - par définition - globale.

Une instruction d'appel d'une procédure-sousprogramme peut avoir une destination globale lorsque l'exécution du sousprogramme en question peut être abandonnée directement ou indirectement par l'exécution d'une instruction stop. La détermination des instructions nomdesspg qui peuvent avoir une destination globale n'est pas considérée ici ; il faudrait, pour effectuer cette détermination, examiner le graphe GPA (cf. ch. 4).

On dira qu'une instruction de "saut" s d'un fief f est un "saut" interne à f si la liste globale de s est vide ; sinon le "saut" sera dit externe à f ou encore sortie de fief.

L'"étiquette" qui est la destination globale d'une instruction stop, retour (d'un sousprogramme) ou d'une instruction nomdesspg qui est un "saut" externe sera représentée par une valeur spéciale dans l'algorithme d'insertion exposé par la suite (soit "étiqexit" cette valeur spéciale).

Au niveau du texte intermédiaire d'un fief f_1 il importera de connaître les destinations locales des instructions de "saut" de f_1 et les instructions de "saut" qui sont des sorties de fief. Les destinations globales d'un "saut" de f_1 seront représentées par une instruction sortie placée à la suite de l'instruction bloc ou boucle dont f_1 est le texte représenté (cf. 6.2.7). Il y aura par conséquent - au niveau du texte intermédiaire du fief f_2, prédécesseur immédiat de f_1 par rapport à l'arbre ASF - une instruction sortie pour tout "saut" externe de f_1.

Les instructions sortie introduites devront ensuite être traitées - par l'algorithme d'insertion des instructions sortie - comme des instructions de "saut" c'est-à-dire avec des listes locales et globales associées.

Nous présentons par la suite un programme en Algol68 qui effectue l'insertion des instructions sortie dans les textes intermédiaires des fiefs composant un corps de procédure. Les textes intermédiaires sont des listes d'instructions ; les instructions de structure bloc et boucle de ces textes servent à représenter l'arbre ASF du corps de procédure ; aux instructions de "saut" sont attachées les listes locales et globales décrites précédemment.

```
début co l'algorithme d'insertion des sorties co
union instr = (instra, instrb, co autres instructions co) ;
struct instra = (entier codeinstr, opds opérandes,
                 rep info inf, rep instr suivante) ;
union  opds = co à définir co ;
mode rin = rep instr, rf = rep info ;
struct instrb = (entier codeinstr, nofief, rf inf,
                 rin suivante, fiefreprésenté) ;
co les instructions du mode instrb sont : bloc n et boucle n ; nofief = n ;
   suivante : instruction qui suit textuellement ; fiefreprésenté : pointeur
   sur l'instruction fief n correspondante co
```

```
union info = (sautinfo,  co autres formes d'informations co) ;
struct étiq = (chaîne étiqval, entier fiefdedcl co : fief auquel étiqval
               est locale co) ;
struct sautinfo = ([1 : 0 flex] étiq destloc, destglob) ;
co sautinfo est le mode des informations associées aux instructions de
saut et à l'instruction sortie ; l'entier fiefdedcl est 0 s'il s'agit du
fief-procédure, fiefdedcl de l'étiquette "étiqexit" est-1, fiefdedcl>0
dans tous les autres cas co
entier codesaut = co le code de l'instruction saut co,
       codesautcond = co le code de sautcond co,
       codestop     = co le code de stop co,
       coderetour   = co le code de retour d'un sous-programme co,
       codeappelsspg= co un code unique associé - pour les besoins de cet
                      algorithme - à toute instruction nomdesspg qui
                      est un "saut" externe co,
       codesortie   = co le code de sortie co,
       codefin      = co le code de fin co ;
rin corpstraité  ; co pointeur sur la première instruction (= prologue) du
                   texte du fief-procédure du corps traité co
co lecture des textes des fiefs du corps traité ; le pointeur corpstraité
   est mis à jour co
rin courante := suivante de corpstraité
procédure insertion = (rep rin arrivée) :
début entier fiefcourant ; rin lien, départ ;
     si arrivée :=: nil co premier appel co
     alors fiefcourant := 0
     sinon co arrivée pointe sur une instruction bloc ou boucle co
          courante := suivante de fiefreprésenté de arrivée ;
          fiefcourant := nofief de arrivée ;
          départ := arrivée ;
          lien := suivante de arrivée
     fsi ;
     tantque codeinstr de courante ≠ codefin faire
          début
          si instrb :: courante co traiter le texte du fief représenté co
          alors insertion (courante)
          sinon
```

```
            si entier c = codeinstr de courante ;
            ((c = codesaut) v (c = codesautcond) v (c = codestop)
            v(c = coderetour) v (c = codeappelsspg) v (c = codesortie))
            ^(fiefcourant ≠ 0)
            alors co traiter le saut co
            [1:0 flex] étiq tabloc, tabglob ; entier i, j, k ;
            j := k := 0 ;
            pour i jusqua bs destglob de inf de courante
            faire (fiefdedcl de destglob [i] de inf de courante
                  = fiefcourant|tabloc[j+:=1]|
                    tabglob[k + := 1]) :=
                    destglob[i] de inf de courante ;
            tas instr sortieinsérée ;
            codeinstr de sortieinsérée := codesortie ;
            (rf : inf de sortieinsérée) := sautinfo : (tabloc, tabglob) ;
            (rep rin : suivante de départ) := sortieinsérée ;
            suivante de sortieinsérée := lien ;
            départ := sortieinsérée
            fsi co saut traité co
            fsi co instruction traitée co
            courante := suivante de courante
            fin co boucle co
fin co procédure insertion co ;
insertion (nil) ; co courante est un pointeur global à la procédure
                    insertion co
fin co de l'algorithme d'insertion co
```

L'algorithme est illustré par un exemple qui est exposé dans les figures 9 et 10. Le schéma de la figure 9 représente un corps de procédure composé de trois fiefs ; les destinations locales et globales des sauts sont indiquées par des flèches. La figure 10 donne le schéma des trois textes intermédiaires correspondants avec les instructions sortie insérées par l'algorithme ; les flèches indiquent les destinations locales des sauts ; les flèches étiquetées par S sont attachées aux instructions de saut qui sont des sorties des fiefs (saut externes).

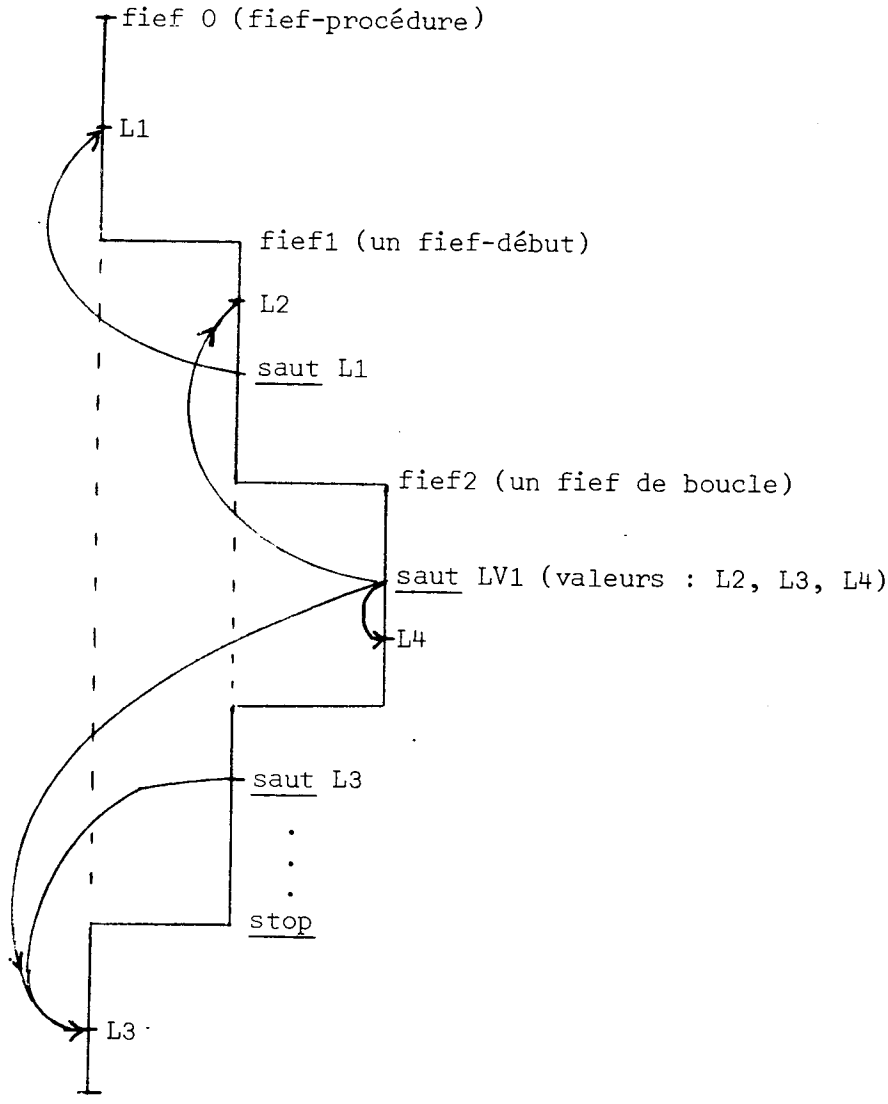

fief 0 (fief-procédure)
L1
fief1 (un fief-début)
L2
saut L1
fief2 (un fief de boucle)
saut LV1 (valeurs : L2, L3, L4)
L4
saut L3
stop
L3

Figure 9.

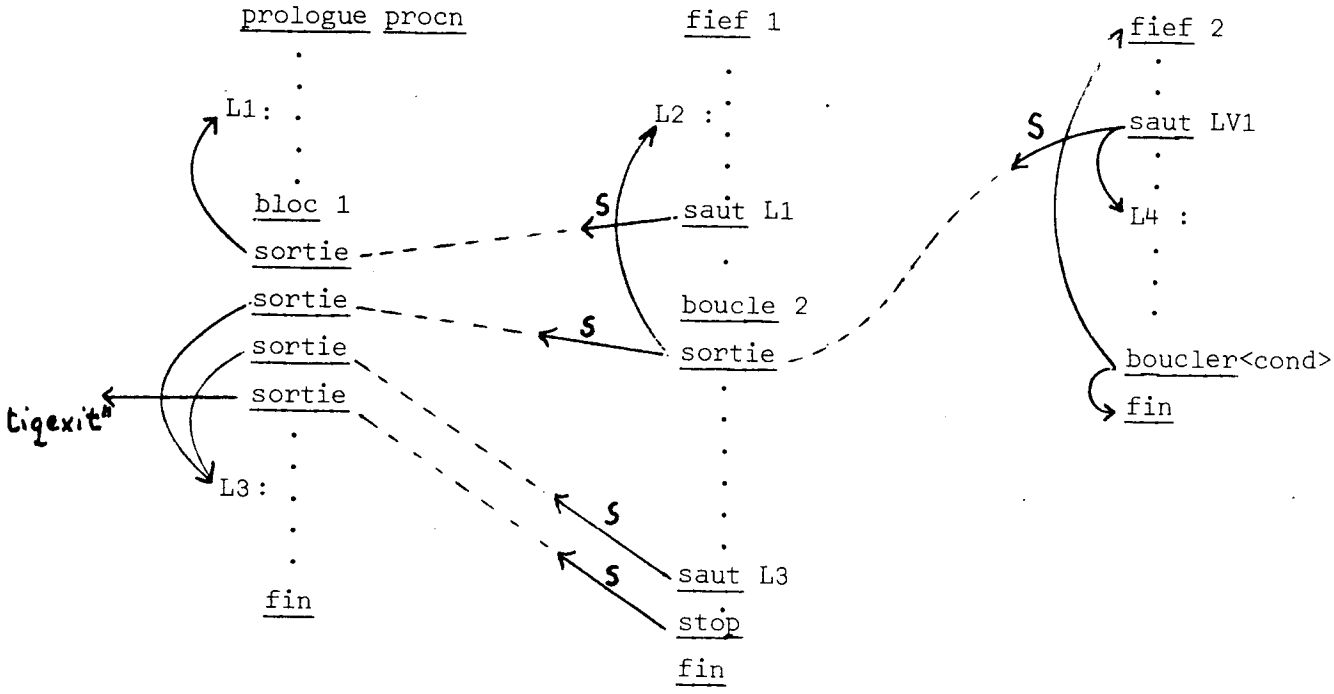


Figure 10.

6.4 - Opérandes d'un programme intermédiaire -

Nous expliquerons dans ce sous-chapitre la correspondance qui existe entre les variables-source de l'ensemble VS (cf. 4.2.2) et les opérandes des instructions d'un programme intermédiaire. L'établissement de cette correspondance permettra, d'une part, de rattacher les informations déduites relatives aux modifications des variables-source de l'ensemble VSMcVS (cf. 4.2.2) aux opérandes correspondants et, d'autre part, de déterminer les opérandes qui peuvent être considérés comme non-modifiables.

Nous définirons les termes suivants : opérande modifiable, classe d'allocation des opérandes et portée d'une opérande.

6.4.1 - Opérandes modifiables -

Les opérandes d'un programme intermédiaire seront partagés en deux ensembles disjoints : en l'ensemble d'opérandes non-modifiables et en l'ensemble d'opérandes modifiables.

Les opérandes non-modifiables correspondront, au niveau de SPL/1 et de l'implémentation, aux deux classes de données : aux données constantes au

niveau de la compilation (ou du chargement) et aux données constantes au niveau d'une activation d'un corps de procédure du programme-source considéré.

La première classe de données comprendra les éléments suivants :

- constantes (arithmétiques, étiquettes ou ENTRY) du programme-source ;
- variables-source de l'ensemble VS-VSM (cf. 4.2.2) et déclarées avec l'attribut STATIC et (éventuellement) initialisées par l'attribut INIT ; ce sont des variables-source qui ne sont plus modifiées par la suite ;
- bornes et multiplicateurs des tableaux qui sont des constantes au niveau de la compilation.

La deuxième classe de données comprendra :

- variables-source de VS-VSM déclarées avec l'attribut AUTO et (éventuellement) initialisées ; elles ne sont plus modifiées par la suite ;
- bornes et multiplicateurs des tableaux qui ne sont pas des constantes au niveau de la compilation (c'est-à-dire qu'on ne peut évaluer à la compilation).

D'autres opérandes seront considérés comme des opérandes non-modifiables ; ce sera le cas des opérandes particuliers suivants :

- tableaux intermédiaires (ils sont alloués dynamiquement au cours des appels, cf. ch. 5) : les éléments de ces tableaux sont initialisés à l'appel et ne sont plus modifiés par la suite au niveau du corps de la procédure appelante ;
- opérandes simples qui correspondent aux données simples générées par l'implémentation des boucles DO ; il s'agit des données simples recevant, à l'initialisation d'une boucle DO la valeur du pas, la valeur de l'expression limite ("expression-TO") et la valeur du déplacement de la variable "contrôlée" de la boucle lorsque cette variable est un élément de tableau (voir les exemples de 6.2.9). Les valeurs de ces données simples restent constantes pendant une exécution complète de la boucle DO correspondante.

Les opérandes modifiables sont les temporaires d'un programme intermédiaire et les opérandes qui correspondent, au niveau de SPL/1, aux variables-source de VSM.

Un opérande modifiable qui n'est pas un temporaire de LI doit pouvoir être un opérande d'une expression de LI (cf. 6.2.2) : les étiquettes variables ne seront donc pas incluses dans l'ensemble des opérandes modifiables.

La signification du partage en opérandes modifiables et non-modifiables apparaîtra pleinement dans le chapitre suivant.

6.4.2 - Classes de recouvrement des opérandes modifiables -

Soit $OM = \cup_i OM_i \cup T$ la partition suivante de l'ensemble OM des opérandes modifiables : l'ensemble T est l'ensemble des temporaires et un ensemble OM_i est tel qu'une modification d'un opérande de OM_i peut modifier l'un quelconque des autres opérandes de OM_i et ne peut modifier aucun opérande de $(OM-T)-OM_i$.

L'ensemble des opérandes d'un OM_i correspond aux opérandes simples et aux tableaux qui se recouvrent mutuellement et lorsque ce recouvrement est relatif à l'attribut-source DEFINED. Le recouvrement des paramètres et des arguments n'est pas considéré ici.

Un ensemble OM_i sera appelé une classe de recouvrement. Les opérandes indéxés et les sous-tableaux dont les bases (qui sont des opérandes-tableaux) font partie d'un OM_i ne sont pas explicitement inclus dans cet ensemble OM_i : ils sont des éléments implicites de la classe de recouvrement définie par l'ensemble OM_i en question.

6.4.3 - Portées des opérandes -

On appellera ici portée d'un opérande le fief de "déclaration" de l'opérande. Il s'agit donc de définir le fief de "déclaration" d'un opérande selon la nature de ce dernier :

- si l'opérande correspond à une variable-source (l'opérande peut être un opérande simple ou un opérande-tableau, cf. 6.2.1) la portée de l'opérande est le fief de déclaration (cf.4.2.2) de l'opérande ;
- si l'opérande correspond à une constante-source ou est une borne ou un multiplicateur de tableau constant(e) au niveau de la compilation alors le fief de "déclaration" de l'opérande sera le fief-procédure de la procédure MAIN du programme-source (c'est-à-dire la portée d'un tel opérande est tout le programme) ;
- si l'opérande correspond à une borne ou un multiplicateur de tableau non constant(e) au niveau de la compilation alors le fief de "déclaration" de l'opérande sera le fief de déclaration du tableau correspondant ;
- si l'opérande est composé d'une base b et des indices $i_1, \ldots, i_n$ $(n \geq 1)$ (cf. 6.2.1 : c'est le cas d'un opérande indéxé ou d'un opérande-sous-tableau) alors son fief de "déclaration" [illegible]

min (fief de "déclaration" de b,
fief de "déclaration" de i_1, ...
fief de "déclaration" de i_n).

L'ordre partiel des fiefs sera celui des sommets de l'arbre global de l'inclusion statique des fiefs d'un programme-source (noté AGF) : cet arbre s'obtient en remplaçant les sommets cp_i de l'arbre ASC (cf. 4.1) du programme par les arbres ASF_i associés (la définition de cet arbre global ne sera pas explicitée davantage). Un sommet s_i de AGF sera dit plus petit qu'un sommet s_j de AGF si s_j est un successeur de s_i dans l'arbre AGF (tout sommet de AGF est un successeur du sommet-racine de AGF). Notons que le minimum ci-dessus est bien défini car les fiefs qui sont ses arguments doivent se trouver sur un même chemin de AGF ;

- si l'opérande est un temporaire t apparaissant dans une instruction arithmétique, logique ou de conversion (cf. 6.2.2) :

$$\underline{\text{motclé}}\ opel_1, \ldots, opel_n \rightarrow t, \text{ avec } n = 1 \text{ ou } 2$$

alors le fief de "déclaration" de t sera le fief minimal suivant :

min(fief de "déclaration" de $opel_1$, ...
fief de "déclaration "de $opel_n$)

- si l'opérande est un temporaire apparaissant dans une instruction d'appel d'une procédure-fonction (cf. 6.2.6) :

$$\underline{\text{fonct}}\ arg_1, \ldots, arg_n \rightarrow t \quad , \quad n \geq 0$$

alors le fief de "déclaration" de t sera le fief minimal

min(fief de "déclaration" de fonct,
fief de "déclaration" de arg_1, ...
fief de "déclaration" de arg_n).

Le fief de "déclaration" de fonct - en tant que l'étiquette-source d'une instruction PROCEDURE (fonct est un <nom de fonction>, cf. 6.2.6) - est le fief de déclaration de cette étiquette-source.

Nous n'avons pas défini le terme de portée pour tous les opérandes d'un programme ; ainsi, en particulier, nous n'avons pas considéré les opérandes simples qui correspondent aux données simples générées par l'implémentation des boucles DO (cf. 6.4.1).

6.5 - Expressions d'un programme intermédiaire -

Rappelons (cf. 6.2.2 et 6.2.6) qu'une expression est par définition la partie-source d'une instruction arithmétique, logique ou de conversion ou la partie-source d'une instruction d'appel de procédure-fonction ; une expression est associée (cf. 6.2.2) à un temporaire : nous noterons par t_e le temporaire dont l'expression associée est e.

On appellera portée d'une expression e la portée de t_e.

Cette notion de portée d'une expression sera utilisée lors de la description des optimisations qui consistent en un déplacement (dans un programme intermédiaire) des instructions d'affectation dont les parties-source sont des expressions : il sera impossible de déplacer une instruction d'affectation dont la partie-source est une expression e dans un fief qui est plus petit (cf. 6.4.3) que le fief de "déclaration" de t_e. Cette limitation résulte du fait que l'allocation des variables-source est liée à la portée de ces dernières.

On appellera composants immédiats d'une expression les opérandes modifiables (cf. 6.4.1) qui composent cette expression. Ces opérandes peuvent être des opérandes simples, des tableaux et des temporaires.

Exemples : - les composants immédiats de l'expression mult ta (t_1), b sont les opérandes modifiables ta, t_1 et b (dans un opérande indéxé on discerne sa base et son indice) ;

- les composants immédiats de

fonct val tb(t_1), dum c, td, te (*, t_2, t_3),
dum tf (t_4, *), tt_1, k2

sont les opérandes modifiables tb, t_1, c, td, te, t_2, t_3, tf et t_4 (tt_1 est un tableau temporaire et k2 est une constante : ce sont donc des opérandes non-modifiables).

On appellera composants d'une expression e l'ensemble des composants immédiats de e, autres que des temporaires, uni aux ensembles des composants des expressions associées à ceux des composants immédiats de e qui sont des temporaires. Cette définition récursive spécifie tout simplement l'ensemble des feuilles - qui sont des opérandes simples modifiables ou des opérandes-tableaux modifiables - de l'arbre d'une expression e, arbre construit au niveau de LI (cet arbre peut éventuellement comporter des sommets qui représentent l'opération d'indéxation ou de sélection de sous-tableau : voir l'opérateur select de 5.4.2.2).

On notera par MOD(e), où e est une expression d'un programme intermédiaire, l'ensemble suivant :

$$\bigcup_{i=1}^{n} OM(c_i) \text{ où}$$

- $\{c_1, \ldots, c_n\}$ est l'ensemble des composants de e
- $OM(c_i)$ est la classe de recouvrement (cf. 6.4.2) contenant c_i.

L'ensemble MOD(e) est l'ensemble des opérandes modifiables (correspondant, au niveau de SPL/1, aux variables-source) dont la modification peut entraîner la modification de (la valeur de) l'expression e.

On notera par $MODR_i$ l'ensemble des opérandes modifiables qui peuvent être modifiés par l'instruction de rangement $rgmt_i$ dont la destination est un opérande simple modifiable ou un opérande indéxé à base modifiable. (Pour tout autre rangement $rgmt_j$ on a $MODR_j = \emptyset$).

Un ensemble $MODR_i$ associé à l'instruction de rangement (cf. 6.2.3) :

$opélem_i \rightarrow opmod_i$ ou

$opélem_i \rightarrow opmod_i\ (indice_i)$

est défini ainsi :

$$MODR_i = OM(opmod_i) \cup \{t_e \mid \forall\, opm \in OM(opmod_i), \forall e : opm \in MOD(e)\}.$$

7 - GRAPHES D'UN PROGRAMME EN LANGAGE INTERMEDIAIRE

Nous introduirons dans ce chapitre la terminologie relative aux graphes des programmes. Un graphe de programme (en anglais : control flow graph) correspond à la notion d'organigramme ; les chemins d'un graphe de programme indiquent les séquences d'exécution possibles des instructions d'un programme en langage intermédiaire.

En 7.1 et 7.2 seront décrits les graphes associés à certaines entités statiques introduites dans les chapitres précédents : fiefs, corps des boucles, corps des procédures.

En 7.3 nous présenterons l'ordre de base des sommets d'un graphe et en 7.4 sera donné un algorithme déterminant les articulations d'un graphe.

Certains rappels seront brièvement exposés dans le sous-chapitre 7.5.

7.1 - Blocs élémentaires -

Les instructions d'un texte intermédiaire (cf. chapitre 6) seront groupés en blocs élémentaires (ou blocs tout court) [3, 4]. Un bloc est une suite maximale d'instructions d'un texte intermédiaire dont l'exécution est séquentielle. Les blocs seront les sommets des graphes des programmes.

La détermination des blocs d'un texte intermédiaire s'effectuera par un parcours séquentiel des instructions de ce texte intermédiaire. La dernière instruction d'un bloc pourra être l'une des instructions suivantes :

- une instruction stop ou retour ;
- une instruction d'appel d'une procédure-sous-programme (on peut sortir directement ou indirectement d'une procédure-sous-programme par une instruction stop) ;
- une instruction saut, sautcond ou boucler ;
- une instruction bloc ou boucle ;
- une instruction sortie (elle constituera à elle seule un bloc) ;
- une instruction fin ;
- toute instruction suivie d'une instruction étiquetée.

Nous verrons par la suite que, dans certains cas, les blocs contenant une instruction sortie pourront être supprimée : ceci dépendra du genre du graphe de programme considéré.

7.2 - Graphes des programmes -

On appellera graphe de programme (ou graphe tout court) un graphe orienté (B, A) où B est l'ensemble des sommets qui sont des blocs élémentaires et où A est l'ensemble des arcs.

Tout graphe de programme comportera un seul sommet d'entrée (l'entrée du graphe) et éventuellement plusieurs sommets de sortie (les sorties du graphe); l'ensemble des sorties sera dénoté par BS, BS $\subset$ B.

Les sommets d'un graphe de programme g qui peuvent être les sorties de g sont les blocs suivants :

- blocs se terminant par une instruction stop, retour ou fin ;
- blocs se terminant par une instruction saut, sautcond ou sortie et
- blocs se terminant par une instruction d'appel d'une procédure-sous-programme lorsque cette instruction peut être un saut externe (cf.6.3).

On appellera sortie absolue d'un graphe de programme g tout bloc b qui est une sortie de g et tel que b se termine par une instruction de "saut" (cf.6.3) dont la liste globale contient l'étiquette "étiqexit" (indiquant que b peut être une sortie du graphe de procédure contenant b).

On supposera qu'il existe pour $\forall$ s $\in$ B un chemin dans le graphe allant de l'entrée du graphe au sommet s.

On pourrait associer un graphe de programme à toute entité statique d'un programme (fief, corps de boucle, corps de procédure, etc.) ; cependant nous utiliserons par la suite seulement les graphes suivants :

- graphe d'un fief,
- graphe d'un corps de procédure,
- graphe d'un corps de boucle propre et
- graphe d'un corps de procédure propre (ou encore graphe de procédure propre).

On appellera corps de boucle propre un corps de boucle (cf. 4.1) dont sont exclus les corps des boucles textuellement internes.

On appellera corps de procédure propre un corps de procédure (cf. 4.1) dont sont exclus les corps des boucles textuellement internes.

Nous présenterons par la suite, à l'aide des exemples, les trois genres des graphes des programmes.

7.2.1 - Graphe d'un fief (ou graphe de fief) -

Les sommets d'un graphe de fief gf sont les blocs élémentaires du texte intermédiaire correspondant au fief en question.

Dans le cas où gf ne correspond pas à un fief-procédure on appellera sortie principale le bloc qui se termine par l'instruction fin. Dans le cas où gf correspond à un fief-procédure le bloc qui se termine par l'instruction fin ne sera pas considéré comme un bloc de sortie de gf.

Les arcs de gf correspondent aux destinations locales (cf.6.3) des instructions saut et sautcond du fief en question ; certains arcs spéciaux peuvent être introduits par les instructions bloc et boucle du fief en question.

Deux exemples sont donnés par la suite (voir les figures 1 et 2) ; ils correspondent au fief de procédure et au fief-début des figures 9 et 10 de 6.3.

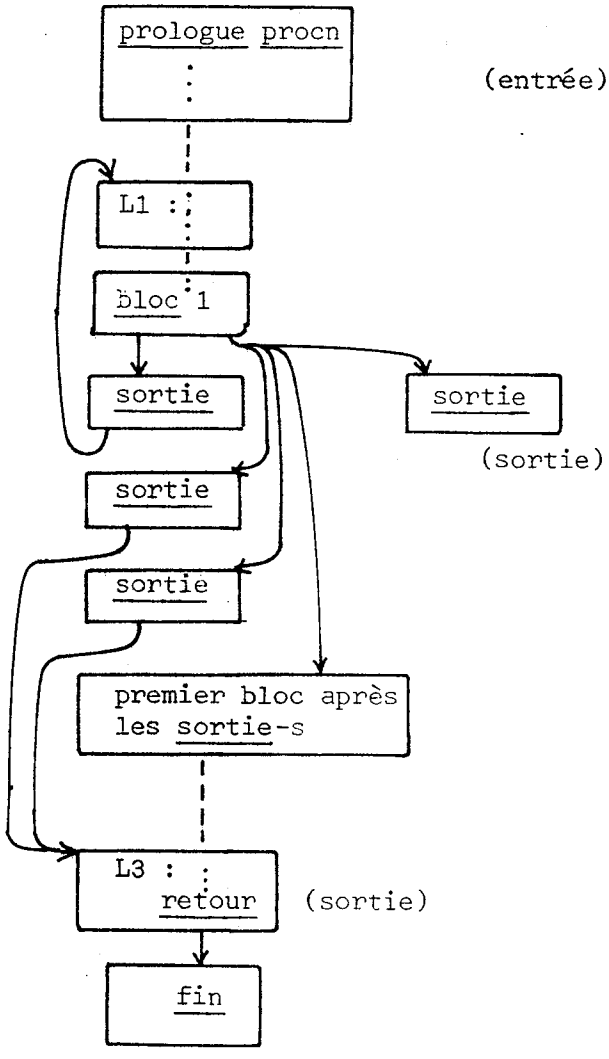


Figure 1. Le schéma du graphe de fief du fief de procédure de la figure 10, chapitre 6.3.

Notons qu'un bloc se terminant par une instruction bloc ou boucle a pour successeurs les blocs que constituent les instructions sortie qui suivent l'instruction bloc ou boucle en question et le bloc commençant par la première instruction qui suit la dernière instruction sortie.

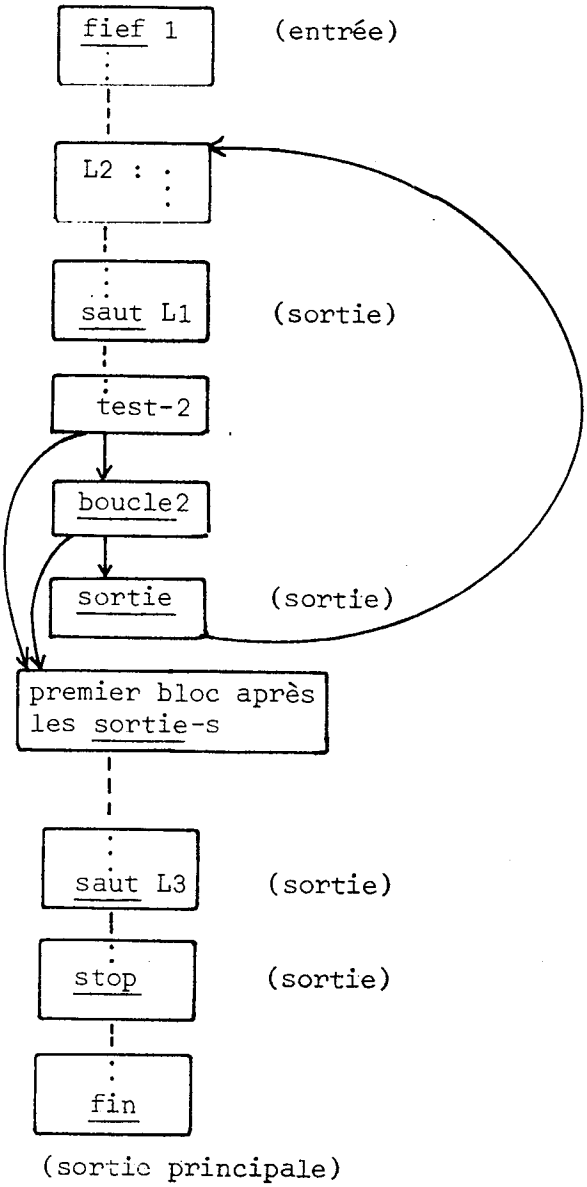


Figure 2. Le schéma du graphe de fief du fief-début de la figure 10. chapitre 6.3.

7.2.2 - Graphe d'un corps de procédure (ou graphe de procédure) -

Les sommets d'un graphe de procédure gp sont les blocs élémentaires des textes intermédiaires correspondant aux fiefs du corps de procédure cp_i considéré (il s'agit des fiefs qui sont les sommets de ASF_i, cf.4.1).

Les seuls sommets qui peuvent être des sorties de gp sont les blocs qui se terminent par une instruction stop, retour ou nom-de-sous-programme ;

en effet, les restrictions du ch.3 impliquent que aucune instruction de saut d'un corps de procédure ne peut avoir de destinations globales à ce corps de procédure. Ceci entraîne, d'autre part, l'inutilité des instructions sortie qui ont été insérées (cf.6.3) dans les textes intermédiaires du corps de procédure considéré ; le graphe gp associé ne comportera donc pas de blocs que constituent ces instructions sortie.

Nous reprendrons l'exemple des figures 9 et 10 de 6.3 ; le graphe de procédure correspondant est schématisé dans la figure 3. Nous pourrons remarquer que le seul successeur d'un bloc se terminant par bloc n ou boucle n est le bloc commençant par l'instruction fief n correspondante.

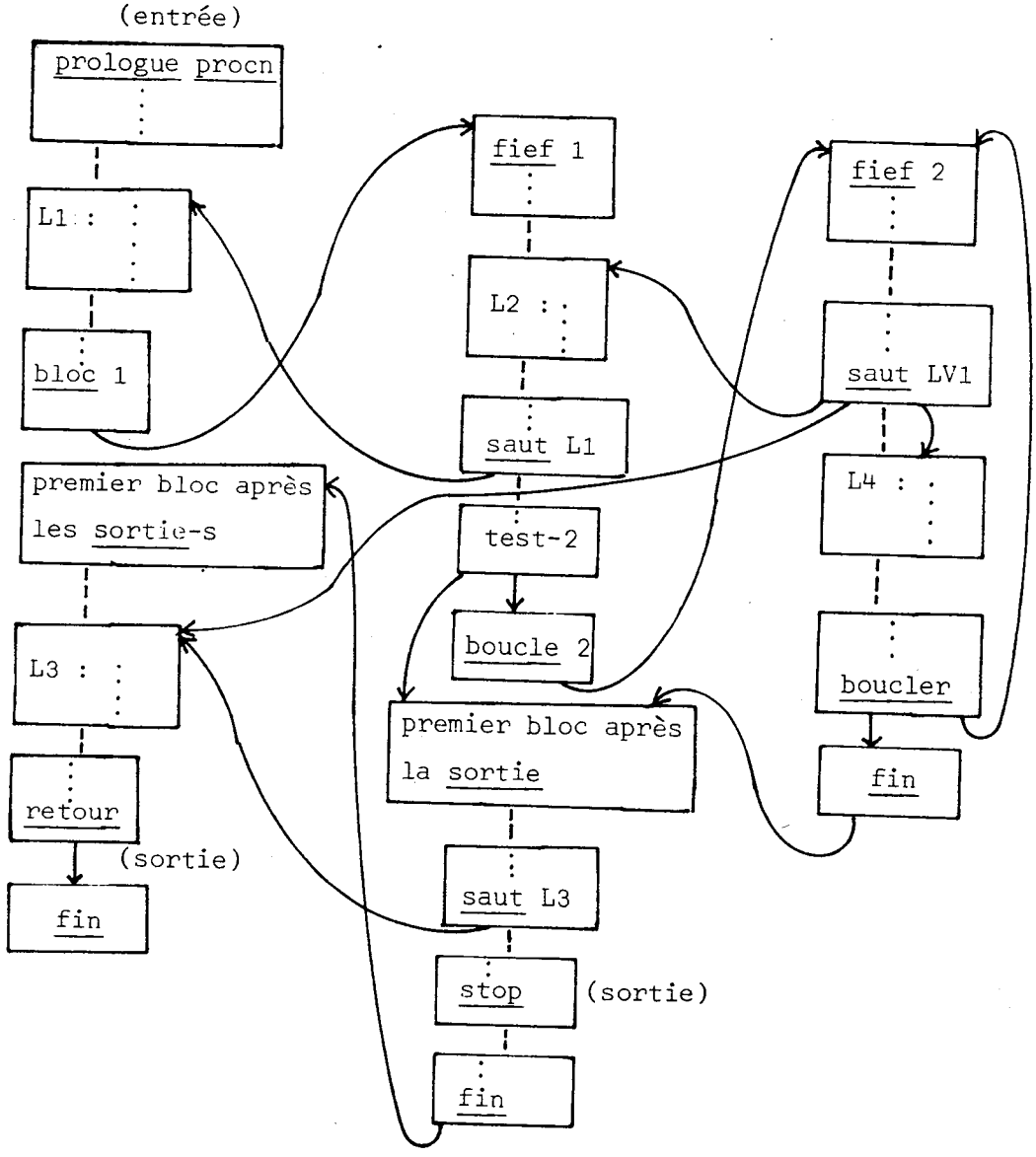


Figure 3. Le schéma du graphe de procédure qui correspond au corps de procédure de la figure 10, chapitre 6.3.

7.2.3 - Graphe d'un corps de boucle propre (ou graphe de boucle propre) -

Les sommets d'un graphe de boucle propre sont les blocs élémentaires des textes intermédiaires correspondant aux fiefs du corps de boucle propre considéré.

Les instructions sortie qui apparaissent à la suite d'une instruction bloc sont inutiles ; il faudra, par contre, garder les blocs que constituent les instructions sortie apparaissant à la suite d'une instruction boucle du corps de boucle propre considéré.

Un exemple est donné par les figures 4 et 5.

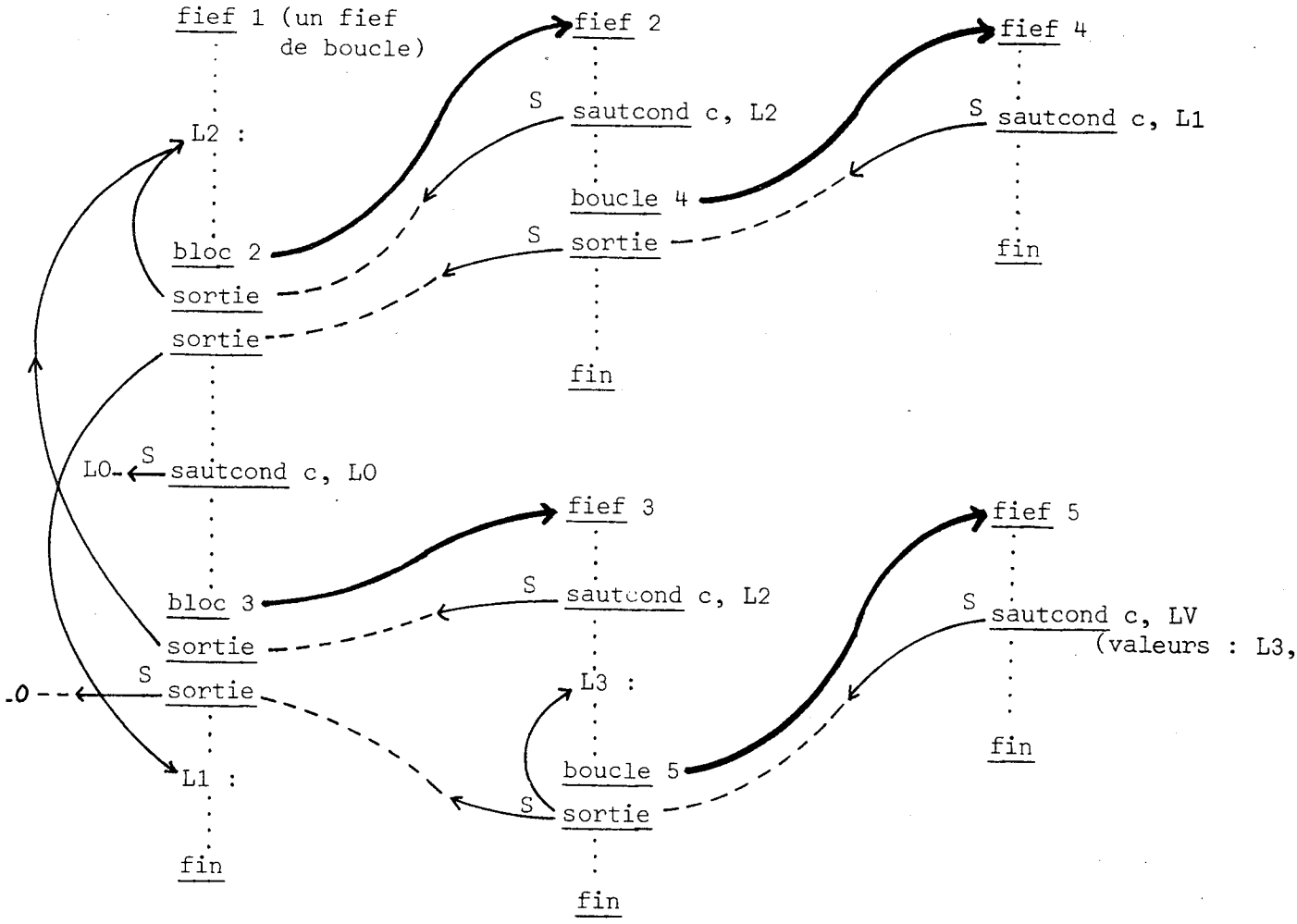


Figure 4. Le schéma des textes intermédiaires correspondant aux fiefs d'un corps de boucle (les notations sont celles de la figure 10, chapitre 6.3). Le corps de boucle propre considéré est constitué des fiefs numérotés 1, 2 et 3.

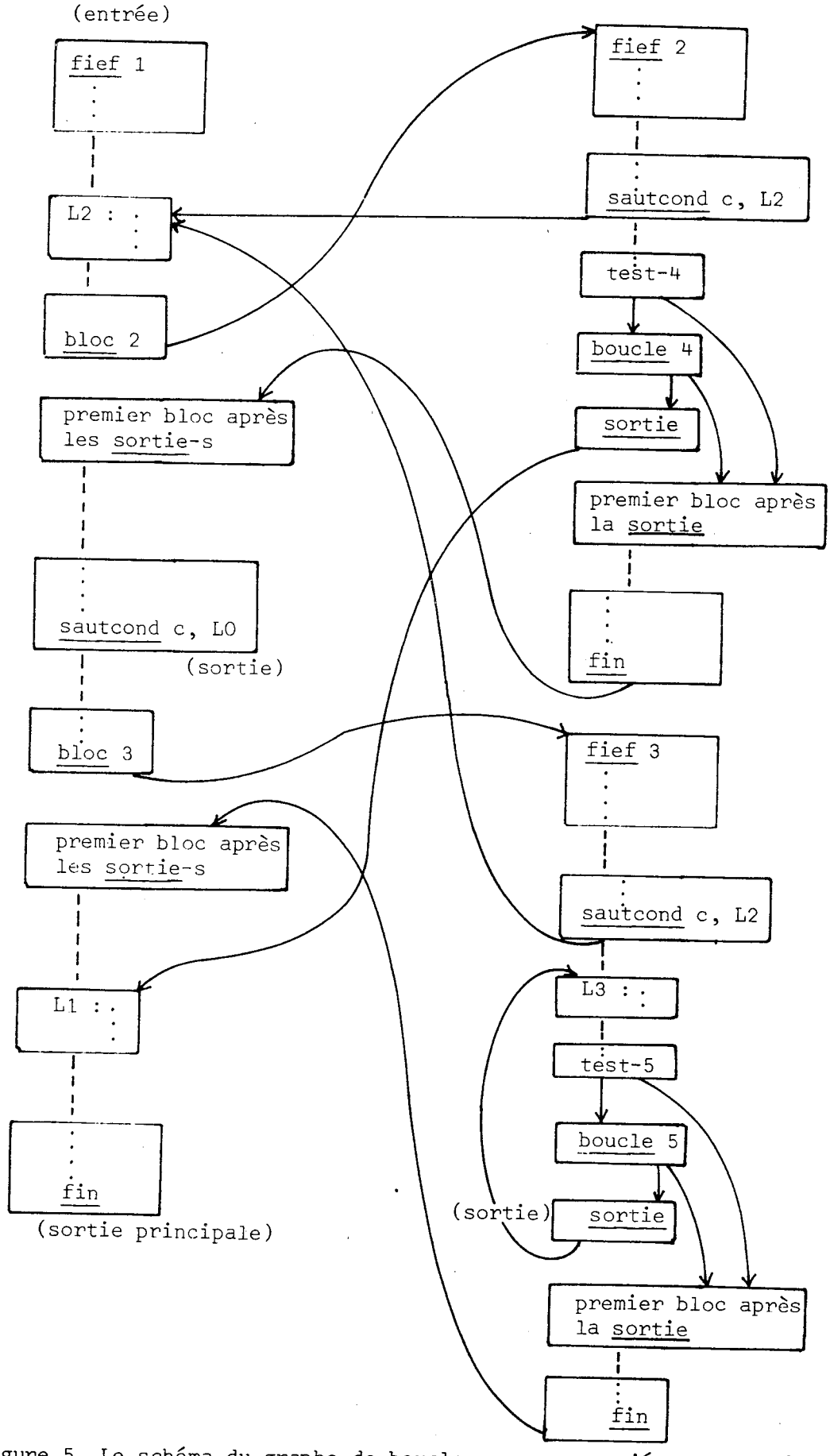


Figure 5. Le schéma du graphe de boucle propre associé au corps de boucle propre de la figure 4.

Le bloc se terminant par l'instruction fin qui termine le fief de boucle du corps de boucle considéré est la sortie principale du graphe de boucle propr

La définition d'un graphe de procédure propre est en tout point analogue à celle d'un graphe de boucle propre.

Note : Les graphes des programmes suivants n'ont pas de sortie principale (toutes leurs sorties sont non-principales) :

- graphe de procédure ;
- graphe de procédure propre ;
- graphe de fief-procédure.

7.2.4 - Décomposition d'un graphe de procédure -

Les optimisations qui seront examinées en 8.2 seront effectuées sur les graphes des programmes décrits précédemment.

Deux décompositions d'un graphe de procédure GP seront brièvement décrites par la suite ; ces décompositions seront utilisées en 8.2.2.

Dans la première décomposition les composants (les graphes-composants) de GP seront les graphes des fiefs qui correspondent aux fiefs du corps de proc dure associé au graphe GP ; la structure des composants sera par conséquent celle de l'arbre ASF du corps de procédure considéré.

Dans la deuxième décomposition les composants de GP seront : le graphe du corps de la procédure propre qui correspond au graphe GP et les graphes des boucles propres qui correspondent aux corps des boucles du corps de procédur associé au graphe GP ; dans cette décomposition la structure des composants est celle d'un arbre ARB des boucles associé au corps de procédure considéré et qui peut être obtenu à partir de ASF d'une façon évidente.

A tout graphe-composant gc de GP pourront correspondre plusieurs blocs de représentation du graphe gc dans le graphe-composant gc' qui est le prédécesseur de gc dans l'arbre ASF ou dans l'arbre ARB.

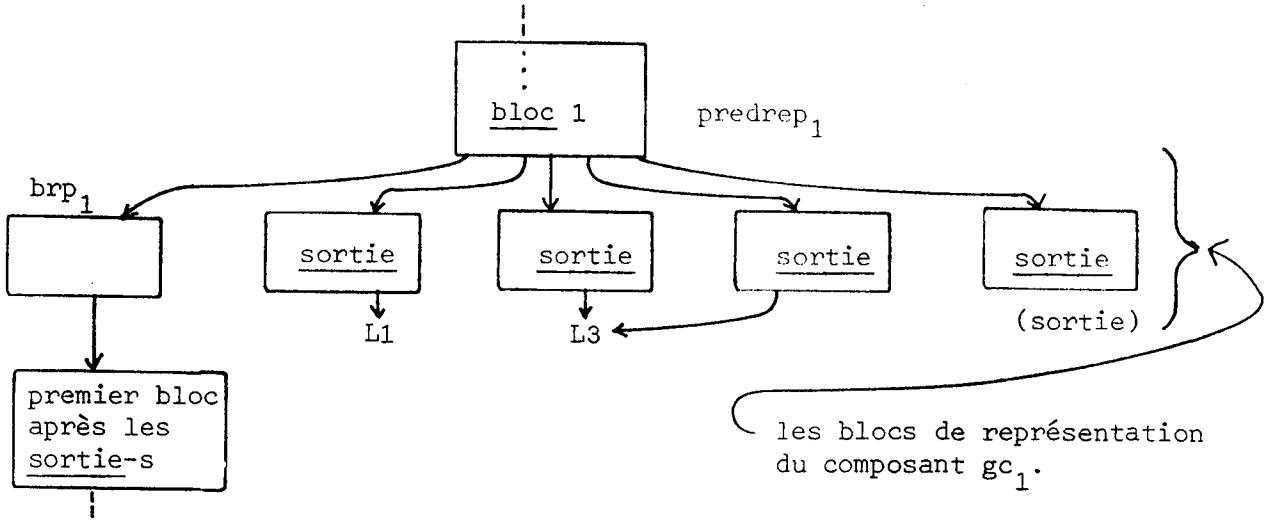
Chacun des blocs de représentation d'un composant gc correspondra à une sortie du graphe gc ; le bloc de représentation principal correspondra à la sortie principale du graphe gc. Un bloc de représentation principal est initialement vide.

Les blocs de représentation non-principaux correspondront aux sorties non-principales du graphe gc ; ces blocs sont constitués d'une instruction sortie (cf. 6.2.7 et 6.3).

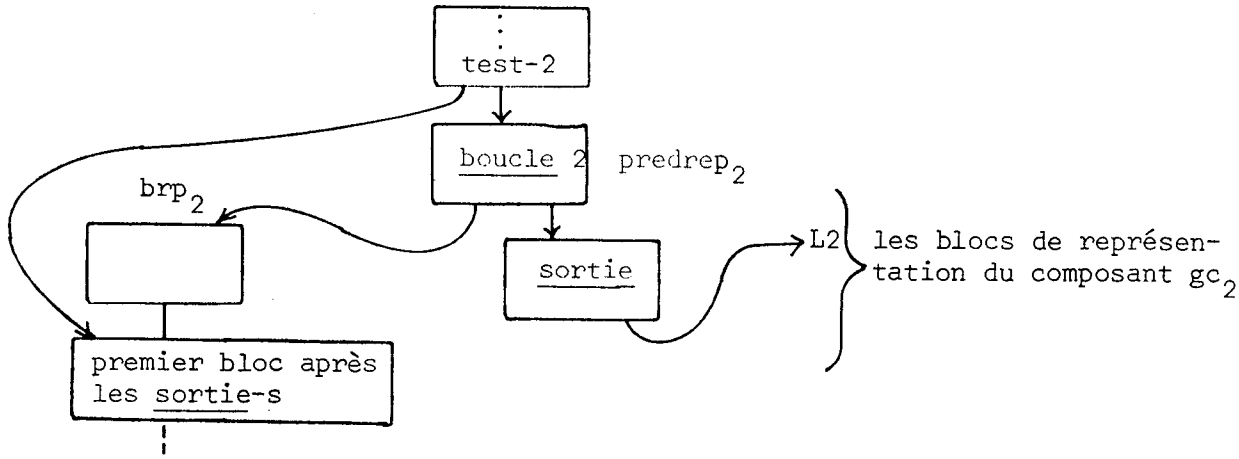
Reprenons l'exemple de 6.3 (figures 9 et 10) : le graphe de procédure GP est donné par la figure 3 de 7.2.2 ; GP a trois composants par la première décomposition : le graphe de fief gc_0 du fief de procédure (figure 1 de 7.2.1), le graphe de fief gc_1 du fief 1 (figure 2 de 7.2.1) et le graphe de fief gc_2 du fief 2 (ce dernier ne contient aucun bloc de représentation).

L'insertion des blocs de représentation principaux (en abrégé : brp) dans les graphes gc_0 et gc_1 est illustrée par la figure 5.1. Les graphes-composants (les composants gc_0 et gc_1 de l'exemple) ainsi modifiés seront utilisés lors de la résolution des systèmes d'équations booléennes, systèmes qui fournissent des informations nécessaires aux optimisations (cf. chapitre 8).

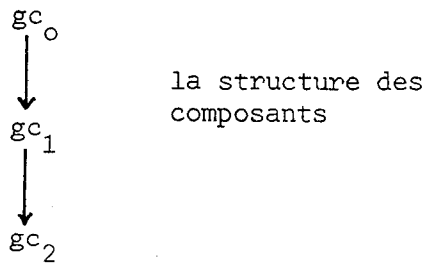
Si BR_i est l'ensemble des blocs de représentation d'un graphe-composant gc_i alors on pourra remarquer que $\text{card}(\Gamma^{-}(BR_i)) = 1$ où Γ est la correspondance du graphe-composant qui contient les blocs de représentation de gc_i. Les blocs de représentation de gc_i ont donc un même bloc prédécesseur qu'on notera par $predrep_i$ (cf. figure 5.1).



Le schéma du graphe gc_o, cf. figure 1 de 7.2.1.



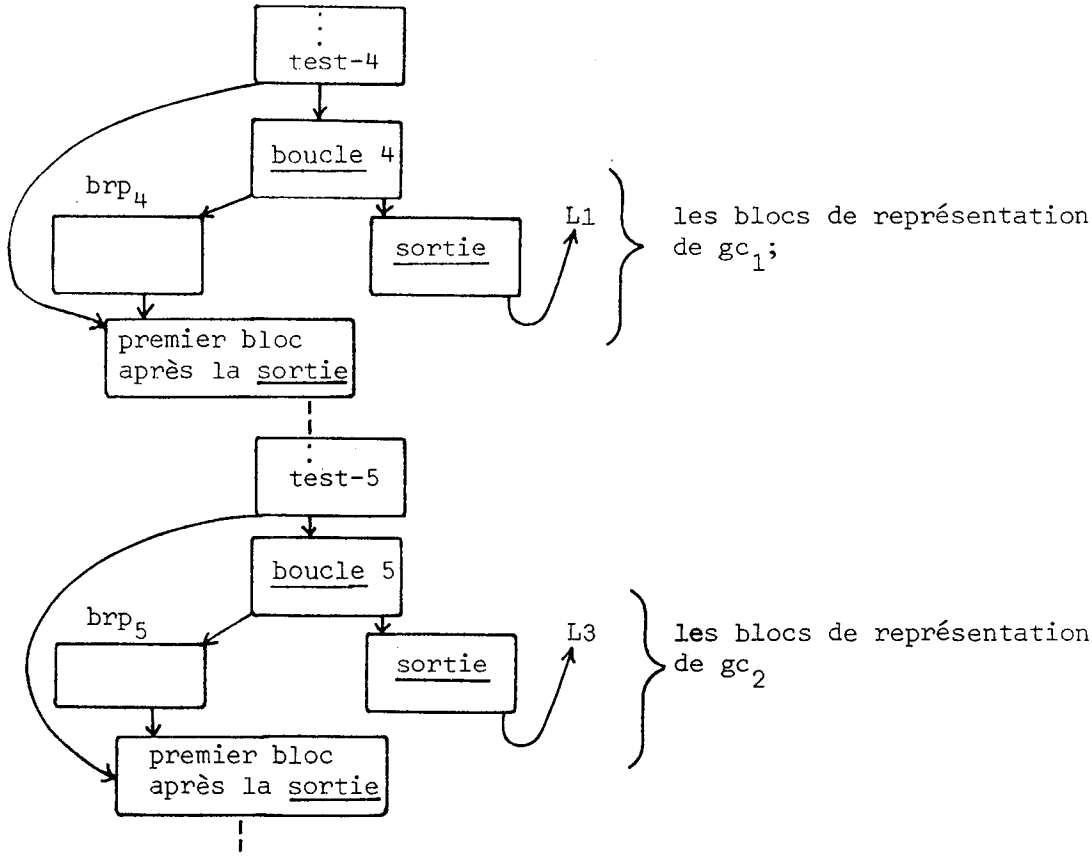
Le schéma du graphe gc_1, cf. figure 2 de 7.2.1.



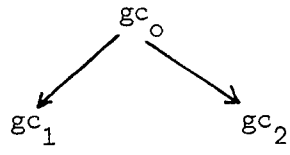
L'arbre ASF du corps de procédure de la figure 10, chapitre 6.3.

Figure 5.1. Les composants par la première décomposition d'un graphe de procédure.

La deuxième décomposition est illustrée dans la figure 5.2 : l'exemple considéré est celui des figures 4 et 5 de 7.2.3 où le fief 1 est remplacé par un fief - procédure (c'est-à-dire l'instruction fief 1 est remplacée par une instruction prologue ; l'instruction sautcond, c,L0 du fief 1 est supprimée ainsi que l'instruction sautcond c,LV du fief 5 et la dernière instruction sortie du fief 1 -voir en effet les restrictions du chapitre 3).



Le schéma du graphe gc_o, cf. figure 5, avec les blocs de représentation de gc_1 et gc_2.



L'arbre ARB du corps de procédure considéré.

Figure 5.2. Les composants par la deuxième décomposition d'un graphe de procédure.

Le schéma de la figure 4 représente alors un corps de procédure ; soit GP son graphe de procédure. GP a trois composants par la deuxième décomposition :

- le graphe de procédure propre gc_0 (figure 5) ;
- le graphe de boucle propre gc_1 (il correspond au fief 4 de la figure 4) et
- le graphe de boucle propre gc_2 (il correspond au fief 5 de la figure 4).

Les graphes gc_1 et gc_2 ne contiennent aucun bloc de représentation et le graphe gc_0 contient les blocs de représentation des graphes gc_1 et gc_2 (cf. figure 5.2).

7.3 - Numérotation des sommets d'un graphe de programme. Prédominance -

Les graphes des programmes seront utilisés dans les algorithmes de collection d'informations et seront traités par les algorithmes d'optimisation.

Il faudra à cet effet numéroter (c'est-à-dire ordonner totalement) les sommets des graphes. Les algorithmes de collection d'informations ne nécessitent pas une numérotation particulière : on pourrait utiliser une numérotation "naturelle" qui correspond à l'ordre textuel des blocs élémentaires à la sortie du compilateur. Il a été montré [cf. 33] que, dans la plupart des cas, une numérotation "naturelle" peut convenir, c'est-à-dire que le choix d'une autre numérotation n'augmente pas sensiblement la rapidité de convergence des algorithmes de collection d'informations.

Nous utiliserons cependant une numérotation particulière qui définit un ordre des sommets appelé ordre de base, cf. [17], [34].

Nous donnons par la suite un algorithme en Algol68 (l'algorithme est celui de [26]) qui fournit un ordre de base des sommets d'un graphe. L'ordre initial des sommets est quelconque ; le sommet d'entrée du graphe doit toutefois être le premier.

L'algorithme ordredebase :

```
proc ordredebase = (rep [1 :, 1 :] bool connect
    co connect est la matrice associée au graphe ; les indices de connect
       correspondent, à l'entrée, à une numérotation initiale des sommets ;
       le sommet d'entrée du graphe correspond à l'indice 1 de connect ;
       à la sortie les indices de connect correspondent à l'ordre de base
       des sommets trouvé par l'algorithme co                    ):
début  entier n = bs connect ;
       entier k, i := 0, m := 1 ;
       [1 : n] entier ordre ; struct pred = (entier pred) ;
       [1 : n] pred arbre ; [1 : n] bool visité ;
  pour k jusquà n faire visité [k] := faux ;
       proc next =:
        pour le jusquà n faire
          si connect [m,k] ∧ ¬ visité [k]
          alors pred de arbre [k] := m ; m := k ;
                visité [k] := vrai ; next ; allera exit
          fsi co fin de next co ;
   1 : next ;
exit : ordre [m] := (i +:= 1) ;
       si m ≠ 1 alors co remonter co
         m := pred de arbre [m] ; allera 1
       fsi ;
       ordre [1] := n ;
       co remise à jour de connect selon la nouvelle numérotation inversée :
          l'indice i de connect tel que ordre [i] = j devient l'indice
          n + 1 - j co
       début [1 : n, 1 : n]  bool temp ;
         pour i jusquà n faire
         pour j jusquà n faire
         temp [n + 1 - ordre [i], n + 1 - ordre [j]] := connect [i,j] ;
         connect := temp
       fin
fin co ordre de base co
```

Un exemple illustrant l'algorithme est présenté dans les figures 6 et 7. La numérotation initiale est donnée par le graphe de la figure 6. Les numéros à côté des sommets du graphe correspondent à l'ordre de base que fournit l'algorithme (cet ordre n'est pas l'unique ordre de base possible). La figure 7 donne le même graphe mais avec les sommets disposés de haut en bas dans l'ordre de base inversé.

Par la suite, nous noterons par 0 un ordre de base des sommets d'un graphe G = (B,A) et par $\overline{0}$ l'ordre qui est l'inverse de l'ordre 0 : si n = card (B) alors $\overline{0}$ (b) = n + 1 - 0(b), $\forall$ b $\in$ B.

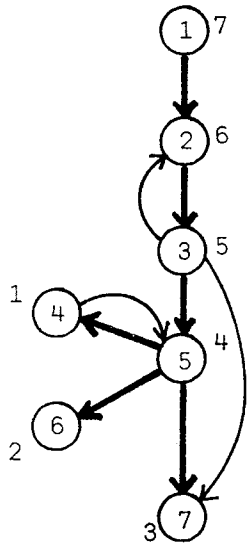


Figure 6. La numérotation initiale (les numéros encerclés) et un ordre de base possible fourni par l'algorithme.

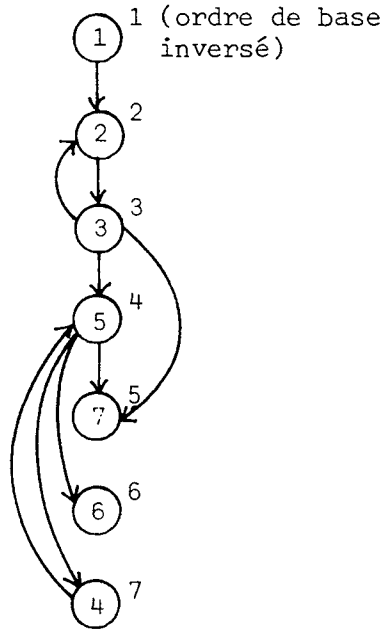


Figure 7. Le graphe de la figure 6 ordonné de haut en bas selon l'inverse $\overline{0}$ de l'ordre de base 0.

Rappelons la définition de la relation de prédominance, cf [32], définie sur les sommets d'un graphe G = (B,A) : la relation prédom (a,b), a $\in$ B, b $\in$ B est vraie uniquement si le sommet a se trouve sur tout chemin allant de l'entrée de G au sommet b.

Si 0 est un ordre de base des sommets de G alors on appelle arc descendant un arc $(b_1,b_2) \in A$ tel que $\overline{0}(b_1) < \overline{0}(b_2)$ (le sommet b_2 est un descendant du sommet b_1) ; dans le cas contraire l'arc est dit montant.
Un chemin descendant est un chemin composé d'arcs descendants.

Notons que l'algorithme ordredebase, appliqué à un graphe G construit un arbre maximal amax de G (cet arbre est représenté par le tableau arbre de l'algorithme) ; il est évident d'après la construction de amax (l'arbre maximal construit dans l'exemple de la figure 6 est représenté en traits gras) que tout chemin de amax est un chemin descendant. D'où la propriété suivante :

Propriété 1. Soit G = (B, A) un graphe et soit O un ordre de base de ses sommets. Pour tout sommet $s \in B$ il existe alors un chemin descendant allant de l'entrée de G au sommet s.

Rappelons encore la propriété suivante [cf. 26] :

Propriété 2. Soit G = (B, A) un graphe et soit O un ordre de base de ses sommets. Si b_1 prédomine b_2 (c'est-à-dire si prédom (b_1, b_2) est vraie) alors $\overline{O}(b_1) < \overline{O}(b_2)$.

7.4 - Sommets d'articulation d'un graphe de programme -

Rappelons qu'un sommet d'articulation (ou une articulation) d'un graphe G = (B, A) (soit s_e l'entrée de G et soit $BS \subset B$, $BS \neq 0$ les sorties de G) est un sommet qui se trouve sur tout chemin de l'ensemble de chemins $\{[s_e, \ldots, s]\}_{s \in BS}$.

Nous présentons ici un algorithme permettant de déterminer les articulations d'un graphe G = (B, A) dont les sommets sont numérotés selon un ordre de base O et auquel est ajouté un sommet s_x sans successeur et dont les prédécesseurs sont les sorties de G (soit $G' = (B \cup \{s_x\}, A')$ ce graphe). Si card(B) = n-1 alors $\overline{O}(s_x) = n$. La donnée à l'entrée de l'algorithme artics est la matrice associée au graphe G' ; les indices de cette matrice correspondent à l'ordre $\overline{O}$ des sommets de G'.

```
proc artics = (rep [1:, 1:] bool connect ; rep[1:bs connect-1]
               bool résultat
               co résultat[i] = vrai : i est une articulation co  ) :
début entier i, j, k, compte := 0 ; entier n = bs connect ;
co nous supposerons, pour simplifier, que n ≤ largeur de bits [cf. 53] co
     [1 : largeur de bits] bool unité = ([1 : largeur de bits] bool un ;
          pour i jusquà largeur de bits faire un [i] := vrai) ;
     [1 : largeur de bits] bool zéro = ([1 : largeur de bits] bool z ;
          pour i jusquà largeur de bits faire z[i] := faux) ;
     [1 : n] bits propag ;
pour i jusquà n-1 faire propag[i] := bab unité ;
propag[n] := bab zéro co valeurs à propager co ;
co propager en itérant : co
faire
(bits temp := bab unité ;
     pour i depuis n-1 pas -1 jusquà 1 faire
     (pour j depuis n pas -1 jusquà 1 faire
          co produit des successeurs de i co
          (connect [i, j] | temp := temp ∧ propag [j]) ;
          début [1 : largeur de bits] bool temp 1 := zéro ;
               bits propagligne ; temp 1 [i] := vrai ;
               propagligne := temp ∨ bab temp 1 ;
               si propagligne = propag[i] alors
               compte +:= 1 ; si compte = n-1 alors
                              allera fini fsi
               sinon co mise à jour co
               compte := 0 ;
               propag[i] := propagligne
               fsi
          fin
     )
) ;
fini : co fin d'itérations ; le résultat est donné par propag[1] co
pour i jusqua n-1 faire résultat[i] := i □ propag[1]
fin co artics co
```

L'algorithme artics est un cas particulier de l'algorithme général de collection d'informations [cf. 26] ; l'algorithme général sera rappelé au chapitre suivant.

L'algorithme artics consiste en la propagation de la valeur faux associée à un sommet i depuis le sommet de sortie unique de G' ; la propagation s'effectue en parcourant à rebours les chemins de G'. La signification de i □ propag[1] = faux en fin d'algorithme est la suivante : il existe un chemin à rebours de s_x jusquà l'entrée de G ne passant pas par le sommet i (car seul le sommet i peut arrêter la propagation de la valeur faux associée à i) ; la valeur faux a donc été propagée jusqu'au sommet d'entrée et, par conséquent, le sommet i n'est pas une articulation.

7.5 - Autres rappels -

Nous rappellerons ici d'autres définitions relatives à la structure des graphes des programmes [cf. 4].
On appelle intervalle I(s) le sous-graphe maximal d'un graphe tel que s est l'unique entrée de I(s) et tel que tous les circuits élémentaires de I(s) passent par le sommet s. Le sommet s de I(s) est appelé tête de l'intervalle.
Il est possible de partitionner d'une façon unique tout graphe de programme en un ensemble d'intervalles disjoints.
On appelle graphe dérivé $I(G_o) = G_1$ du graphe G_o le graphe dont les sommets sont les intervalles de G_o. Les sommets $I(s_1)$ et $I(s_2)$ de G_1 sont reliés par un arc s'il existe un arc dans G_o reliant un sommet $s \in I(s_1)$ de G_o à la tête s_2 de $I(s_2)$.

La suite dérivée est la suite $G = G_o, G_1, \ldots, G_k$ telle que $G_{i+1} = I(G_i)$, $G_{k-1} \neq G_k$ et $I(G_k) = G_k$. G_k est le graphe limite du graphe initial G_o.

On dit qu'un graphe de programme G est réductible si son graphe limite est constitué d'un seul sommet.

Rappelons quelques propriétés des intervalles :

- la tête de l'intervalle prédomine (cf. 7.3) tout sommet de l'intervalle ;
- toute composante fortement connexe (cf. 5.4.1) d'un intervalle contient la tête de l'intervalle ;
- quelque soit l'ordre de base des sommets d'un intervalle I(s) les seuls arcs montants (cf. 7.3) de I(s) sont ceux dont l'extrémité terminale est le sommet de tête s de I(s) [cf. 26].

Un exemple d'un intervalle est donné par la figure 8.

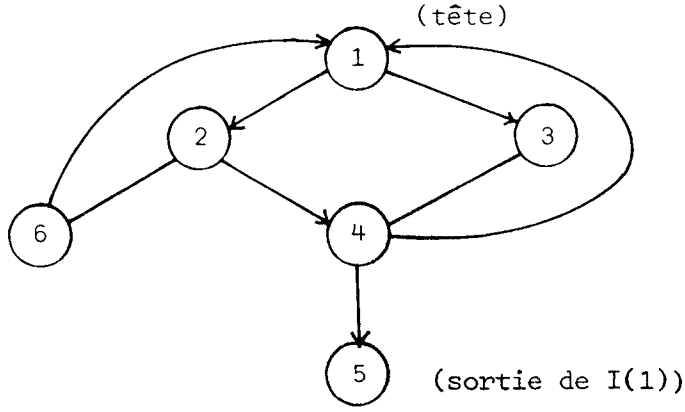


Figure 8 : Les sommets 1, 4 et 5 sont les articulations de I(1) ; les seuls arcs montants sont les arcs (6, 1) et (4, 1) ; les trois composantes fortement connexes sont {1, 2, 6}, {1, 3, 4} et {1, 2, 3, 4, 6}.

Un exemple de graphe dérivé est donné par la figure 9 : le graphe G_1 de la figure 9 est le graphe dérivé du graphe G_o de la figure 6 de 7.3.

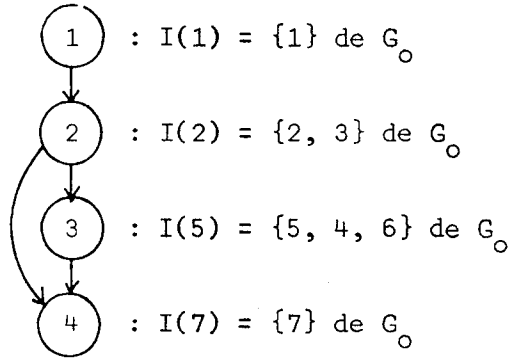


Figure 9 : Le graphe dérivé G_1 de G_o. Remarquons que le graphe G_2 de la suite dérivée est réduit à un seul sommet : le graphe G_o est réductible.

8 - OPTIMISATIONS APPLIQUEES A UN PROGRAMME INTERMEDIAIRE -

Deux optimisations seront décrites dans ce chapitre : l'élimination des instructions redondantes et le déplacement des instructions en dehors des boucles.

Les méthodes de collection d'informations seront décrites en 8.1.1. ; les deux optimisations seront présentées en 8.1.2. et les approches possibles au traitement de ces optimisations seront exposées en 8.2.

Après avoir défini une approche par décomposition nous discuterons en 8.2.5. ses avantages et ses inconvénients.

8.1 - Rappels. Optimisations proposées -

Nous décrirons par la suite les deux optimisations proposées : élimination des expressions (ou des instructions) redondantes et extraction des expressions invariantes (ou déplacement des instructions en dehors des boucles). Cette description sera précédée d'une brève exposition des méthodes de collection d'informations, informations qui sont nécessaires à la réalisation de ces optimisations.

Les optimisations décrites seront effectuées au niveau du langage intermédiaire : nous utiliserons donc la terminologie introduite dans les chapitres 6 et 7. Certains termes relatifs aux méthodes de collection d'informations seront empruntés à [33].

8.1.1 - Méthodes de collection d'informations -

Nous rappellerons ici brièvement les méthodes de résolution des systèmes d'équations booléennes. Ces méthodes permettent de calculer les différents coefficients booléens associés aux expressions d'un programme intermédiaire, coefficients dont les valeurs déterminent la possibilité d'application des optimisations considérées.

Ce chapitre présentera des systèmes généraux ; des systèmes concrets seront présentés en 8.1.2.

8.1.1.1 - Résolution d'un système d'équations booléennes

Soit G = (B,A) un graphe de programme (cf.7.2) ; be $\in$ B l'entrée de G et BS $\subset$ B les sorties de G.

On associera à G un graphe G_a dans le cas d'un système ascendant (voir plus loin) ou un graphe G_d dans le cas d'un système descendant.

Le graphe $G_d = (B_d, A_d)$ est défini ainsi :

$$B_d = B \cup \{ef\} ;$$

$$A_d = A \cup \{(ef,be)\} ;$$

Γ^- (be) = {ef} où Γ^- est la correspondance de G_d et où ef est l'entrée fictiv (un bloc vide) de G ;

Γ de G_d est identique à Γ de G sur tous les autres sommets de G.

On utilisera par la suite la notation suivante :

be : le bloc d'entrée de G ;
ef : l'entrée fictive de G (un bloc vide) ;
sp : la sortie principale de G ;
s_j, j=1,...S(S = card (BS) - 1): les sorties non-principales de G ;
sfp : la sortie fictive principale (un bloc vide) associée à sp ;
sf_j : les sorties fictives (des blocs vides) associées aux sorties s_j ;
brp : le bloc de représentation principal (cf.7.2.4.) de G ;
br_j : les blocs de représentation non-principaux associés aux sorties s_j de G.

Le graphe $G_a = (B_a, A_a)$ est défini ainsi (on ajoutera une sortie fictive sf_j pour chaque sortie s_j et une sortie fictive principale pour la sortie sp) :

$$B_a = B \cup \{sf_j\}_{j \in [1,S]} \cup \{sfp\}$$

$$A_a = A \cup \{(s_j, sf_j)\}_{j \in [1, S]} \cup \{(sp, sfp)\}$$

$$\Gamma^+ (s_j) = \{sf_j\}, \; j \in [1, S]$$

Γ^+ (sp) = {sfp}, où Γ est la correspondance de G_a ;

Γ est identique à Γ de G sur tous les autres sommets de G_a.

Note (cf.7.2.3.) : rappelons que si G est :

- un graphe de procédure,
- un graphe de procédure propre ou
- un graphe de fief-procédure

alors G ne comporte que des sorties non-principales. Il s'ensuit que dans ce cas la sortie fictive principale n'existe pas.

La forme générale d'un système d'équations booléennes attaché à (ou "sur") un graphe G_a ou G_d peut s'écrire ainsi (cf.[33]) :

$$Y_i = B_i + A_i * \underset{j \in R(i)}{\textcircled{X}} Y_j, \quad \forall i \in [1, N] \text{ , où :}$$

- les opérateurs + et Σ dénotent l'opération logique ou et les opérateurs * et Π dénotent l'opération logique et ;

- les indices dans le cas d'un système descendant (sur G_d) :
  - indice 0 correspond à ef,
  - les indices 1 à N (N = card (B)) correspondent aux sommets de B_d - {ef} ;

- les indices dans le cas d'un système ascendant (sur G_a) :
  - les indices -card (BS) + 1 à 0 correspondent aux sorties fictives de G_a,
  - les indices 1 à N correspondent aux sommets de G_a (le bloc d'entrée ayant l'indice N).

- $\textcircled{X}$ représente Σ ou Π ;

- R est la relation Γ^+ (notée aussi Suc) de G_a ou la relation Γ^- (notée aussi Pred) de G_d ;

- B_i et A_i sont des vecteurs booléens qui représentent les coefficients booléens associés au sommet i ;

- Y_i est un vecteur booléen représentant les coefficients dont on cherche la valeur ;

- l'indice j d'un élément B_i [j], A_i [j] ou Y_i [j] correspond à une expression du programme intermédiaire ; chacun de ces éléments est un coefficient booléen spécifiant une propriété donnée d'une expression j au sommet (bloc) i.

Si $\textcircled{X} = \Sigma$ le système est appelé un système-somme sinon ($\textcircled{X} = \Pi$) le système est un système-produit.

Si R = Pred le système est dit descendant sinon (R = Suc) le système est dit ascendant.

Les vecteurs Y_i représentent une information en sortie (resp. en entrée) dans le cas d'un système descendant (resp. ascendant) et les vecteurs $\overline{Y}_i = \underset{j \in R(i)}{\textcircled{X}} Y_j$ représentent une information en entrée (resp. en sortie) dans le cas d'un système descendant (resp. ascendant).

La résolution d'un système d'équations booléennes nécessite une hypothèse sur la valeur de Y_o à l'entrée du graphe G lorsque le système est descendant et aux sorties de G lorsque le système est ascendant. Cette valeur correspond à l'information qui "atteint" G de l'"extérieur" ; cette information sera associée au sommet ef de G_d ou aux sorties fictives de G_a (l'entrée fictive et les sorties fictives ont été introduites à cet effet).

Il faut en outre définir les valeurs initiales des coefficients Y_i, $\forall i \in [1, N]$. On a montré (cf. [34,33]) que pour un système-produit les valeurs initiales sont définies par $Y_i = B_i + A_i$ alors que, pour un système-somme, les valeurs initiales sont définies par $Y_i = B_i$.

Une itération du système consiste en le calcul des Y_i, $\forall i \in [1, N]$ (l'initialisation des Y_i peut être considérée comme la 0-ième itération).

La résolution d'un système ne dépend pas d'une numérotation particulière des sommets de G, numérotation définie par les indices $i \in [1, N]$. Il est toutefois évident que le système converge plus vite lorsque cette numérotation n'est pas quelconque ; on supposera par la suite que la numérotation de 1 à N des sommets de G est définie par un ordre de base 0 (cf.7.3.) des sommets de G pour un système ascendant et par l'ordre de base inversé $\overline{0}$ pour un système descendant.

La résolution d'un système consiste en des itérations répétées ; à chaque itération sont déterminées les nouvelles valeurs des coefficients $Y_1, \ldots Y_N$.

Comme on peut montrer que le système général converge on obtient, au bout de it itérations, les solutions du système. Le nombre it d'itérations est tel que : $(Y_1, \ldots Y_N)^{it-1} = (Y_1, \ldots Y_N)^{it}$ où la liste $(Y_1, \ldots Y_N)^i$ désigne les valeurs des coefficients après la i-ème itération.

Nous donnons par la suite la forme générale de l'algorithme ALGEN (cf [33]) résolvant un système d'équations booléennes. Les commentaires de ALGEN décrivent les opérations qui ne sont pas exprimées en Algol 68.

```
début entier i, j, N, CPT : = 0 ;
co Pas 1 : initialiser les Y [i], ∀i ∈ [1, N]
           c'est-à-dire faire Y [i] := B [i] + A [i] ou
                              Y [i] := B [i] ;
           déterminer les valeurs des coefficients en entrée fictive ou aux
           sorties fictives selon les hypothèses initiales.

   Pas 2 : itérer jusqu'à la stabilisation de la valeur de Y : co

   faire début
      pour i jusquà N faire
      début co Z : = B [i] + A [i] * Ⓧ Y [j], pour j ∈ R (i) co
            si Z = Y [i]
            alors CPT + : = 1 ;
                  si CPT = N alors allera fini fsi
            sinon Y [i] : = Z ; CPT : = 0
            fsi
      fin co d'une itération co
   fin ; fini :
fin co ALGEN co
```

8.1.1.2 - Nombre d'itérations de l'algorithme ALGEN

Rappelons d'abord les résultats connus (cf. [26], [33]) :

Théorème : soit O un ordre de base des sommets d'un graphe de programme G = (B, A) et soit m (ch) le nombre d'arcs montants (cf.7.3.) d'un chemin élémentaire non-descendant ch de G ; le facteur de retard fr_G de G est défini par :

$fr_G = \max_{ch \in CH_G} (m(ch))$ où CH_G est l'ensemble de tous les chemins élémentaires non-descendants de G.

Le nombre maximal d'itérations $iter_M$ - de ALGEN - nécessaires à la résolution d'un système d'équations booléennes attaché à l'un des deux graphes G_a ou G_d est égal à $fr_G + 2$.

Ce théorème sera illustré à l'aide du système concret suivant (qu'on appellera le Système-retard) :

$$RS_i = EV_i + \sum_{j \in Pred(i)} RS_j, \quad i \in [1, N]$$

Le coefficient EV_i est un vecteur booléen de N+1 éléments ; il est défini ainsi (avec 0 ≡ faux, 1 ≡ vrai) :

$$EV_i[j] = 0, \quad j \neq i \text{ ou } j = 0$$

$$EV_i[i] = 1, \quad i \neq 0$$

Les valeurs initiales des coefficients sont définies ainsi :

$$RS_i = EV_i, \quad i \in [0, N].$$

On peut associer au Système-retard l'interprétation suivante :

- $EV_i[j] = 1$ signifie que l'expression e_j est évaluée par le sommet $s_i \in B$;

- $RS_i[j] = 1$ signifie qu'une valeur de e_j "atteint" la sortie du sommet $s_i \in B$ (c'est-à-dire : il existe un chemin allant du sommet s_j qui évalue e_j au sommet s_i) ; on supposera que les expressions e_j, $j \in [0, N]$ ne sont modifiées par aucun sommet de G.

Un exemple est donné par les figures 1 et 2. La figure 2 représente l'évaluation du Système-retard attaché au graphe G_d de la figure 1, évaluation relative aux coefficients élémentaires $RS_i[2]$. La figure 2 montre que la valeur 1 se "propage depuis le sommet 2 pour atteindre finalement les sommets 3 et 4 par le chemin non-descendant [2, 7, 5, 3] qui comporte deux arcs montants. Notons que $fr_G = 2$

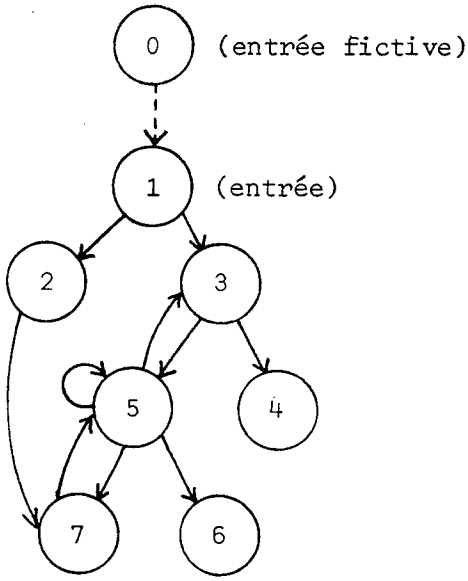


Fig.1. Un graphe de programme G et son graphe associé G_d. Les numéros des sommets correspondent à un ordre de base inversé $\overline{0}$. Les arcs montants : (7,5), (5,3) et (5,5).

	les valeurs des RS_i [2]				
sommet	init.	iter. 1	iter. 2	iter. 3	iter. 4
0	0	inchangé			
1	0	inchangé			
2	1	inchangé			
3	0	0	0	1	1
4	0	0	0	1	1
5	0	0	1	inchangé	
6	0	0	1	inchangé	
7	0	1	inchangé		

Fig. 2. L'évaluation des coefficients RS_i [2].

Le Système-retard relatif à l'expression

e_2 : $RS_i\,[2] = EV_i\,[2] + \sum_{j \in Pred(i)} RS_j\,[2]$

Les valeurs initiales des RS_i [2] :

$EV_i\,[j] = 0,\ j \neq 2$ et $EV_i\,[2] = 1.$

La 4-ième itération confirme la stabilité des solutions trouvées

Lors de la résolution du Système-retard complet (évaluation des vecteurs RS_i) la "propagation" des valeurs 1 s'effectue parallèlement depuis chacun des sommets de G. Il est par conséquent facile de voir que le Système-retard attaché à un graphe G_d fournit, par le nombre d'itérations $iter(G_d)$ que nécessite sa résolution, le nombre $iter_m(G_d) = iter(G_d) - 1$ d'itérations effectivement nécessaires à la résolution d'un système descendant quelconque attaché à G_d. En effet, l'itération finale du Système-retard descendant est une itération de vérification ; lorsque $iter(G_d)$ est connu le nombre d'itérations effectivement nécessaires à la résolution d'un système descendant quelconque sur G_d est alors $iter(G_d) - 1 = iter_m(G_d)$ (on suppose que la numérotation des sommets de G reste la même).

Remarque 1. Etant donné un graphe de programme G le nombre d'itérations $iter(G_d)$ fourni par le Système-retard descendant attaché à G_d peut en général ne pas atteindre le maximum $iter_M(G_d) = fr_G + 2$. Le nombre $iter(G_d)$ vérifie $iter(G_d) = iter_M(G)$ ou $iter(G_d) = iter_M(G) - 1$.

Dans l'exemple de la figure 2, le maximum est atteint : nous avons $fr_G = 2$, $iter_M(G) = 4 = iter(G_d)$ et $iter_m(G_d) = 3$.

La Remarque 1 est illustrée par l'exemple de la figure 3.

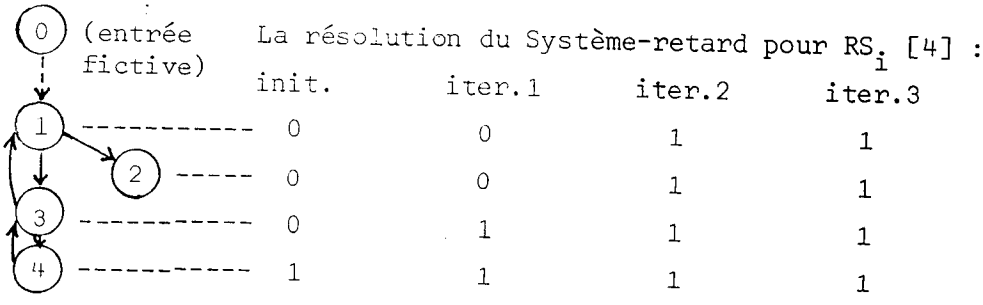


Figure 3. Nous avons : $fr_G = 2$, $iter_M(G) = 4$, $iter(G_d) = 3$.
On peut vérifier que la maximum $iter_M(G)$ n'est pas atteint.

Remarque 2. En utilisant les notations du théorème enoncé au début de ce chapitre soit $CHM_G \subset CH_G$ l'ensemble suivant :

$$CHM_G = \{ch \in CH_G \mid m(ch) = fr_G\}.$$

Le nombre $iter(G_d)$ fourni par le Système-retard descendant attaché à G_d n'atteint pas le maximum $iter_M(G)$ (c'est-à-dire $iter(G_d) = iter_M(G) - 1$) lorsque tout chemin $[s_1, s_2, \ldots s_n]$ de CHM_G commence par un arc montant (l'arc (s_1,s_2)).

Ce résultat se déduit des observations précédentes relatives à la "propagation" des valeurs vrai lors de la résolution du Système-retard descendant.

La figure 3 nous fournit un exemple : nous avons $CHM_G = \{[4, 3, 1, 2]\}$ et l'arc (4, 3) est montant.

En définissant un Système-retard ascendant :

$$RE_i = EV_i + \sum_{j \in Suc(i)} RE_j$$

nous pouvons obtenir le nombre $iter_m(G_a) = iter(G_a) - 1$ d'itérations effectivement nécessaires à la résolution de tout système ascendant attaché à G_a ; $iter(G_a)$ est le nombre d'itérations que nécessite la résolution du Système-retard ascendant sur G_a.

Les Remarques 1 et 2 s'appliquent d'une façon analogue au cas ascendant : ainsi le maximum $iter_M(G)$ n'est pas atteint lorsque tout chemin de CHM_G se termine par un arc montant.

Remarque 3. Il résulte de ce qui précède que pour un même graphe de programme G on peut avoir $iter(G_a) \neq iter(G_d)$ pour une numérotation donnée des sommets de G.

Remarque 4. Si le graphe G = (B, A) est un intervalle (cf. 7.5) alors l'ensemble des arcs montants est l'ensemble $\{(s, \text{tête de } G)\}_{s \in B} \subset A$ quelque soit l'ordre de base des sommets de G. Il s'ensuit que $fr_G = 1$.

8.1.2 - Présentation des optimisations considérées -

Nous exposerons d'abord le traitement des optimisations au niveau d'un bloc élémentaire et ensuite au niveau d'un graphe de programme.

8.1.2.1 - Optimisation au niveau d'un bloc élémentaire -

Nous définirons par la suite les termes utilisés.

On dira qu'un opérande op est <u>défini</u> par une instruction i lorsque i est une instruction d'affectation ou un rangement dont op est la destination.

On dira qu'une expression e d'une instruction i qui n'est pas un rangement est <u>évaluée</u> par i.

Ainsi, lorsqu'une expression e est évaluée par une instruction i l'opérande t_e (le temporaire associé à e) qui est la destination de i est défini par i.

On dira qu'un opérande est <u>utilisé</u> par une instruction comportant une expressio s'il est un composant immédiat (cf.6.5) de e.

Nous supposerons par la suite que la mise en commun des temporaires (cf.5.4.2.1 d'un programme intermédiaire a été effectuée ; cette mise en commun s'effectue de la façon suivante :

1.) Si i_1 et i_2 sont deux instructions d'affectation qui ne sont pas des instructions d'appel :

$$i_1 : exp_1 \rightarrow t_1$$

$$i_2 : exp_2 \rightarrow t_1$$

et si les deux expressions exp_1 et exp_2 sont littéralement égales alors toutes les occurrences de t_1 et t_2 sont remplacées -au niveau d'un corps de procédure (cf.5.4.2.1)- par un temporaire (commun) nouveau t.

2.) Si i_1 et i_2 sont deux instructions d'affectation qui sont des instructions d'appel alors la mise en commun de t_1 de i_1 et de t_2 de i_2 est effectuée si :

- exp_1 de i_1 et exp_2 de i_2 sont littéralement égales et
- $MODAP_{i_1} = MODAP_{i_2} = \phi$ (pas de modifications implicites, cf.4.2.2) et
- aucun argument (cf.6.2.6) de exp_1 ou de exp_2 n'a l'attribut MOD (cf.4.2.2.1).

La mise en commun des temporaires a deux buts :
elle peut mener à une optimisation de l'espace si les temporaires sont alloués en mémoire et, d'autre part, elle simplifie la suppression des expressions redondantes (ou plutôt des instructions évaluant des expressions redondantes). Dans le cas 2 ci-dessus, on ne peut optimiser l'espace (voir l'allocation des listes d'arguments en 5.4.2.2) ; la mise en commun ne peut viser dans ce cas que le deuxième but d'où l'introduction des conditions supplémentaires, conditions qui rendent possible la redondance d'une instruction d'appel.

Remarquons qu'un temporaire commun est défini par plusieurs instructions du programme intermédiaire alors que, avant la mise en commun, tout temporaire n'est défini qu'une seule fois (cf.6.2.1).

Une expression e d'un programme intermédiaire peut être <u>modifiée</u> seulement par les instructions suivantes :

1.) Par une instruction de rangement r_k si (cf.6.5) :
$\exists$ op $\in$ MOD (e) tel que op $\in MODR_k$.

2.) Par une instruction d'appel ap_k si :
$\exists$ op $\in$ MOD (e) tel que op $\in MODAP_k \neq \emptyset$ ou op est un argument de ap_k ayant l'attribut MOD.

3.) Par une instruction <u>sortie</u> constituant un bloc de représentation br (cf.7.2.4) si :
ML (e, br) = <u>vrai</u> (le coefficient ML est défini plus loin).

4.) Par un bloc de représentation principal brp si :
ML (e, brp) = <u>vrai</u>.

5.) Par une instruction <u>fief</u> i si i correspond à un fief de déclaration (cf.4.2.2) déclarant au moins un des opérandes de l'ensemble MOD (e). Il s'ensuit que le bloc élémentaire be commençant par l'instruction <u>fief</u> i a le coefficient :
ML (e, be) = <u>vrai</u>.

Les modifications par les instructions <u>sortie</u> et par les blocs de représentation principaux correspondent à des modifications implicites (cf.4.2.2).

Nous pouvons maintenant définir la <u>redondance locale</u> c'est-à-dire la redondance au niveau d'un bloc élémentaire. Soit un bloc élémentaire b composé des instructions $i_1, i_2, \ldots i_n$, dans cet ordre ;

Soit e_j de i_j une expression dont le temporaire associé t_j est un temporaire commun. S'il existe un indice k, $j < k \leq n$, tel que e_k de i_k (de b) est littéralement égale à e_j (de i_j de b) et si t_k est égal à t_j alors l'expression e_k est localement redondante dans b (et i_k est suppressible) à la condition suivante :

$\forall m$, $j + 1 \leq m \leq k - 1$: l'instruction i_m ne modifie pas l'expression e_j (ou e_k).

Remarquons que l'optimisation qui élimine la redondance locale définie ci-dessus supprime les instructions d'affectation qui ne sont pas des rangements. Il ne semble pas que la possibilité de suppression d'un rangement soit fréquente ; la suppression des rangements ne sera par conséquent pas traitée.

Nous définirons par la suite les coefficients booléens associés à une expressio d'un programme intermédiaire, coefficients qui seront utilisés pour décrire les optimisations au niveau des graphes des programmes.

La propriété de <u>l'anticipabilité locale</u> d'une expression e sera représentée par un coefficient booléen AL associé à un bloc élémentaire b : AL (e, b) est <u>vrai</u> si et seulement si (ssi en abrégé) :

- $\exists$ i tel que $inst_i$ de b = ($inst_1$, ... $inst_n$) évalue e et
- $\nexists$ j, $j < i$ tel que $inst_j$ modifie e.

Si l'optimisation de la redondance locale a déjà été appliquée au bloc b alors l'indice i ci-dessus est unique.

La propriété de la <u>disponibilité locale</u> d'une expression e sera représentée par un coefficient booléen DL associé à un bloc b : DL (e, b) est vrai ssi :

- $\exists$ i tel que $inst_i$ de b = ($inst_1$, ... $inst_n$) évalue e et
- $\nexists$ j, $j > i$ tel que $inst_j$ modifie e.

La remarque relative à l'unicité de l'indice i s'applique également ici.

La propriété de la <u>modifiabilité locale</u> (appelée encore la <u>non-transparence</u>) d'une expression e sera représentée par un coefficient booléen ML associé à un bloc b : ML (e, b) est <u>faux</u> ssi e n'est modifiée par aucune instruction de b.

Si b est un bloc de représentation (cf. 7.2.4) la valeur du coefficient ML (e, b), ∀e, est déduite (cf. 8.2.2) des blocs du graphe-composant représenté par b.

8.1.2.2 - Optimisations au niveau d'un graphe de programme

Nous rappellerons dans ce sous-chapitre l'optimisation de l'élimination des expressions redondantes et l'optimisation qui consiste en des déplacements des instructions en dehors des boucles. Cette dernière optimisation est appelée dans la littérature extraction des expressions invariantes dans des boucles ou encore anticipation (cf. [33]).

Les deux optimisations seront brièvement décrites par la suite et des systèmes d'équations booléennes permettant de traiter ces optimisations seront exposés.

Les optimisations décrites et la plupart des systèmes présentés sont connus ; toutefois, la description des optimisations qui est présentée par la suite suggérera une version modifiée (exposée en 8.2.2) de la méthode de Allen [5] de la collection des informations pour ces optimisations.

Les systèmes d'équations booléennes seront résolus sur des graphes des programmes (graphe de boucle propre, graphe de fief, graphe de procédure, cf.ch.7) ; nous utiliseront les notations de 8.1.1.1 pour le graphe de programme G considéré.

Les optimisations peuvent être locales ou globales (cf. 8.2.1). Une optimisation locale est toute optimisation qui est effectuée au niveau d'un graphe de programme qui n'est pas un graphe de procédure ; une optimisation globale est toute optimisation qui est effectué au niveau d'un graphe de procédure.

La collection des informations nécessaires à une optimisation locale s'effectue par la résolution des systèmes locaux d'équations booléennes. La collection des informations nécessaires à une optimisation globale peut être faite de deux façons (voir aussi 8.2.2) :

- dans l'approche directe on résout des systèmes globaux sur le graphe de procédure traité ;

- dans les approches par décomposition on résout des systèmes globaux sur les

graphes-composants (cf. 7.2.4) du graphe de procédure traité. Cette résolution procède en plusieurs phases : dans une phase ascendante on résout les systèmes en partant des graphes-composants les plus "internes" et dans une phase descendante on part du graphe-composant le plus "externe".

La collection des informations pour les optimisations globales sera présentée d'une façon plus détaillée en 8.2.2.

8.1.2.2.1 - Traitement des redondances

Une (valeur d'une) expression e est disponible en un point p d'un programme intermédiaire si une évaluation de e en p ne change pas la valeur de e ; il s'ensuit qu'une instruction qui évalue e en p peut être supprimée.

A la propriété de disponibilité d'une expression e en entrée d'un bloc élémentaire $b \in B$ (b est un sommet de G) est associé un coefficient booléen DE (e, b) qui est vrai ssi e est disponible en entrée de b.

A la propriété de disponibilité d'une expression e en sortie d'un bloc $b \in B$ est associé un coefficient booléen DS (e, b) qui est vrai ssi e est disponible en sortie de b.

Un système de disponibilité peut alors être défini avec les coefficients DE, DS, DL et ML (cf. 8.1.2.1) ; ce système descendant est attaché au graphe G_d associé à G.

Le système de disponibilité (relatif à une expression e) :

$$DS(e, b) = DL(e, b) + \neg ML(e, b) * \prod_{b' \in Pred(b)} DS(e, b'), \forall b \in B_d - \{ef\}$$

et $$DE(e, b) = \prod_{b' \in Pred(b)} DS(e, b'), \forall b \in B_d - \{ef\}.$$

Le système de disponibilité sera dit local à G si l'hypothèse en entrée est fausse (c'est l'hypothèse "pessimiste") :

$$DS_\ell(e, ef) = \text{faux}.$$

La résolution du système local fournit les valeurs des coefficients DE_ℓ et DS_ℓ (où l'indice ℓ indique qu'il s'agit des solutions du système local) en tout bloc $b \in B$.

Le système de disponibilité sera dit global à G si l'on tient compte de la valeur de DS qui "arrive" à l'entrée de G de "l'extérieur" ; cette valeur est donnée par l'expression suivante :

$$DS\ (e, ef) = DE\ (e, ef) = DE\ (e, brp) * \prod_{j \in [1,S]} DE\ (e, br_j)$$

où la première égalité est due au fait que ef est un bloc vide, ou encore :

DS (e, ef) = DS (e, predrep) en remarquant (cf. 7.2.4) que chacun des blocs de représentation de G a pour prédécesseur unique le bloc noté predrep.

Dans la phrase ascendante initiale (cf. 8.2.2) du processus de la collection des informations on résoudra sur G un <u>système global initial</u> de disponibilité en prenant pour l'hypothèse en entrée fictive de G l'hypothèse "optimiste" suivante (la valeur de DS (e, predrep) n'étant pas encore connue à ce stade) :

$$DS_{gi}\ (e, ef) = \underline{vrai}$$

si le système est résolu sur un graphe de programme G qui est un graphe-composant "interne" du graphe de procédure traité et

$$DS_{gi}\ (e, ef) = \underline{faux}$$

si le système est résolu sur le graphe de programme G qui est le graphe-composant "externe" du graphe de procédure traité (G est un graphe de procédure propre ou un graphe de fief-procédure).

Dans les phrases descendantes et ascendantes subséquentes, par contre, une valeur de DS (e, predrep) sera connue et on résoudra sur G un <u>système global</u> de disponibilité en posant :

$$DS_g\ (e, ef) = DS_g\ (e, predrep).$$

La résolution du système global initial (indices gi) fournit les valeurs des coefficients DE_{gi} et DS_{gi} et la résolution du système global (indices g) fournit les valeurs des coefficients DE_g et DS_g.

Les valeurs des coefficients DE_g et DS_g obtenues en fin du processus de la collection des informations (cf. 8.2.2) et les valeurs des coefficients DE_ℓ et DS_ℓ vérifient les inégalités suivantes :

$DE_g\ (e, b) \geq DE_\ell\ (e, b)$,

$DS_g\ (e, b) \geq DS_\ell\ (e, b)$, $\forall b \in B_d$ et $\forall e$.

L'optimisation d'élimination des expressions redondantes consiste alors en la suppression d'une instruction i d'un bloc b de G lorsque i évalue e et lorsque la condition suivante est vérifiée :

$$AL(e, b) * DE(e, b) = \underline{vrai}.$$

L'optimisation d'élimination sera dite <u>locale</u> si, dans la condition ci-dessus on considère le coefficient DE_ℓ et <u>globale</u> si l'on considère le coefficient DE_g.

Il s'ensuit des inégalités ci-dessus qu'une instruction i qui n'est pas supprimée par l'optimisation locale d'élimination peut l'être par l'optimisation globale d'élimination : on dira, dans ce cas, que l'optimisation locale de i est une optimisation partielle de i (voir encore la Remarque 6 de 8.1.2. et 8.2.1).

8.1.2.2.2 - Déplacements en dehors des boucles

L'optimisation de déplacement sera appliquée à un graphe de boucle propre G = (B, A) (cf. 7.2.3).

Deux cas seront distingués selon la nature du graphe de boucle propre G considéré.

Considérons le schéma d'un graphe de boucle propre G donné par la figure 4 (cf. 7.2.4).

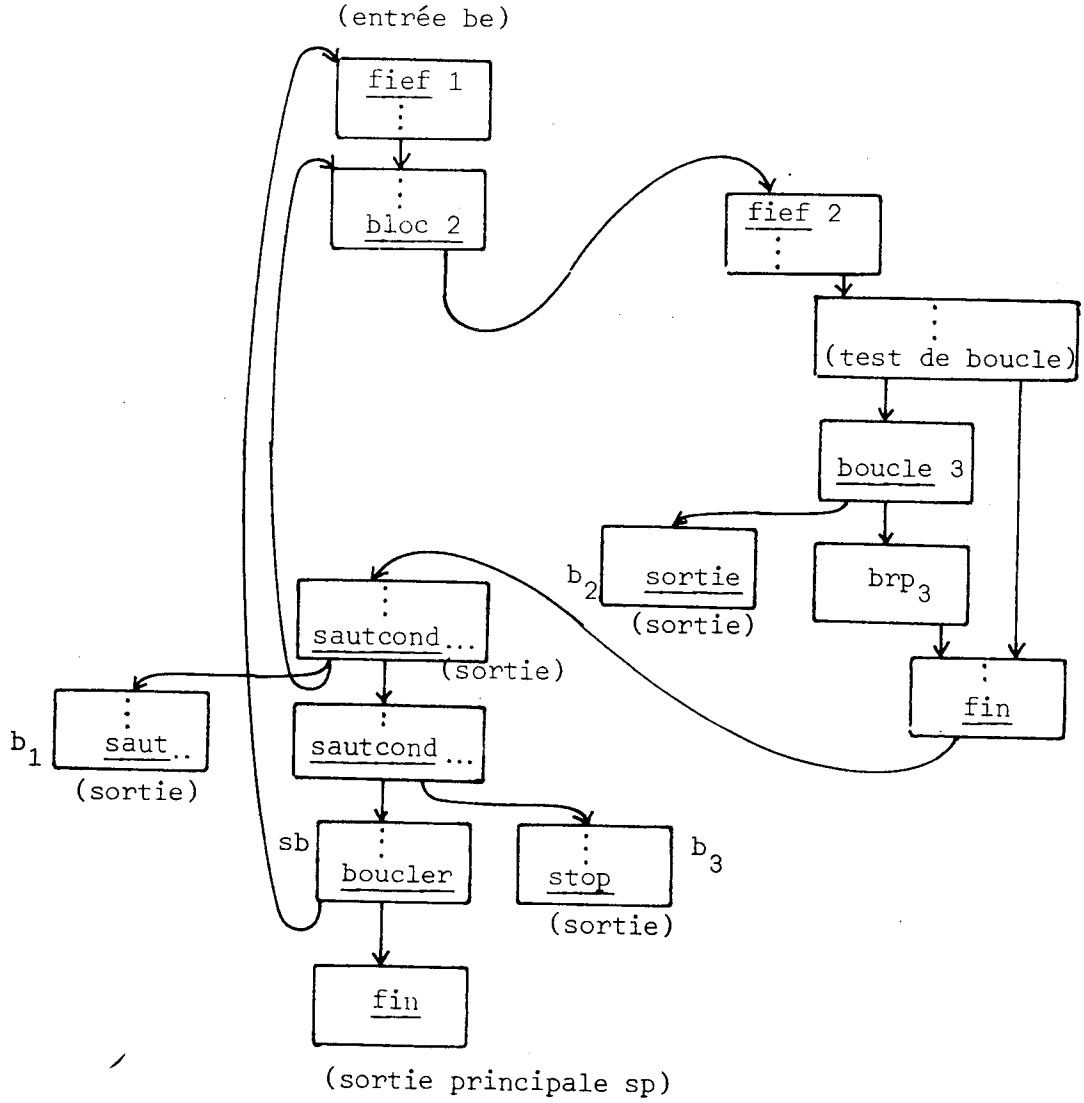


Figure 4. Le schéma d'un graphe de boucle propre G.
(Notons qu'on a toujours : $\Gamma^{-}(sp) = \{sb\}$).

On appellera sortie de boucle sb le bloc (unique) de G dont la dernière instruction est une instruction boucler.

On dira qu'un bloc $b \in B$ n'est pas sous l'arc (sb, be) de G (ou encore que b est un bloc pendant) s'il n'existe pas de chemin dans G allant de b à la sortie principale sp de G. Dans la figure 4, les blocs b_1, b_2, b_3 et sp sont pendants.

Les blocs sous l'arc (sb, be) d'un graphe de boucle propre G peuvent être déterminés par une méthode semblable à celle qui détermine les articulations d'un graphe (cf.7.4). En outre, si G est un intervalle (cf.7.5), il existe un ordre O des blocs de G (appelé numérotation stricte dans la littérature [17,34]) tel que, pour tout bloc b sous l'arc (sb, be) de G on a $O(b) \leq O(sb)$

et pour tout bloc b' pendant on a O(b') > O(sb) ce qui fournit d'une façon simple l'ensemble des blocs sous l'arc (sb, be) de G.

Remarque 5. Si le seul bloc pendant d'un graphe de boucle propre G = (B, A) est le bloc de sortie principale alors le sousgraphe de G engendré par B - {sp} est une composante fortement connexe (cf. 5.4.1).

Soit SP(G) l'ensemble des blocs pendants de G qui sont les sorties de G et qui sont constitués d'une seule instruction (cette instruction peut être : une instruction de "saut" (cf. 6.3), fin ou sortie). Les blocs de cet ensemble se caractérisent par le fait qu'ils ne contiennent aucune instruction déplaçable ; les instructions déplaçables sont les instructions qui évaluent des expressions.

On dira qu'un graphe de boucle propre G = (B, A) possède la propriété P lorsque l'ensemble des blocs pendants de G est égal à l'ensemble SP(G).

Considérons l'exemple de la figure 4 ; nous avons :

SP(G) = $\{b_2, sp\}$ et G ne possède pas la propriété P.

Cas 1) Traitement des déplacements au niveau d'un graphe de boucle propre G = (B, A) qui ne possède pas la propriété P.

Nous décrirons d'abord les conditions générales qui rendent possibles les déplacements des instructions déplaçables en dehors de G.

Une instruction qui évalue une expression peut être déplacée en dehors du graphe de boucle G si l'expression évaluée vérifie les deux propriétés d'invariance et d'anticipabilité.

La propriété de l'invariance d'une expression e dans G peut s'énoncer ainsi : s'il existe une instruction i d'un bloc b de G qui modifie e (c'est-à-dire si ML (e, b) = vrai) alors la modification de e effectuée par i ne doit pas "atteindre" tous les blocs de G ; en particulier elle ne doit pas "atteindre" la sortie du bloc de sortie de boucle sb de G.

La propriété d'invariance est déterminée par la résolution d'un système de modifiabilité descendant attaché au graphe G_d associé à G.

Le système de modifiabilité :

$$MS(e, b) = ML(e, b) + \sum_{b' \in Pred(b)} MS(e, b'),\ \forall b \in B_d - \{ef\}$$

avec $$ME(e, b) = \sum_{b' \in Pred(b)} MS(e, b'),\ \forall b \in B_d - \{ef\}.$$

L'hypothèse en entrée sera fausse : MS (e, ef) = faux.

La résolution du système fournit les valeurs des coefficients MS et ME en tout bloc $b \in B_d - \{ef\}$.

La propriété de l'invariance dans G d'une expression e est vérifiée si :

MS (e, sb) = faux où sb est la sortie de boucle de G.

En effet, si MS (e, sb) = vrai alors la "modification" de e qui "atteint" la sortie de sb se "propage" par l'arc (sb, be) et "atteint" par conséquent tous les blocs de G.

Un exemple est donné par la figure 5 : α représente une instruction évaluant l'expression α et mod (α) représente une instruction qui modifie l'expression α.

Nous avons :

- MS (α, sb) = vrai, MS (β, sb) = MS (γ, sb) = faux
- SP(G) = $\{b_2, sp\}$, l'ensemble des blocs pendants de G est
  BP(G) = $\{b_1, b_2, b_3, sp\}$
- la propriété P n'est pas vérifiée car SP(G) $\neq$ BP(G).

Si une expression e est invariante dans un graphe de boucle G alors toute instruction de G qui modifie e doit se trouver dans un bloc pendant de G ; c'est le cas des expressions β et γ de la figure 5.

La propriété de cohérence (voir aussi 8.1.2.4) introduite par la suite est liée à la propriété d'anticipabilité : la dernière implique la première.

La propriété de la cohérence locale à G d'une expression e peut s'énoncer ainsi : il existe sur tout chemin de G partant de l'entrée be de G une instruction évaluant e.

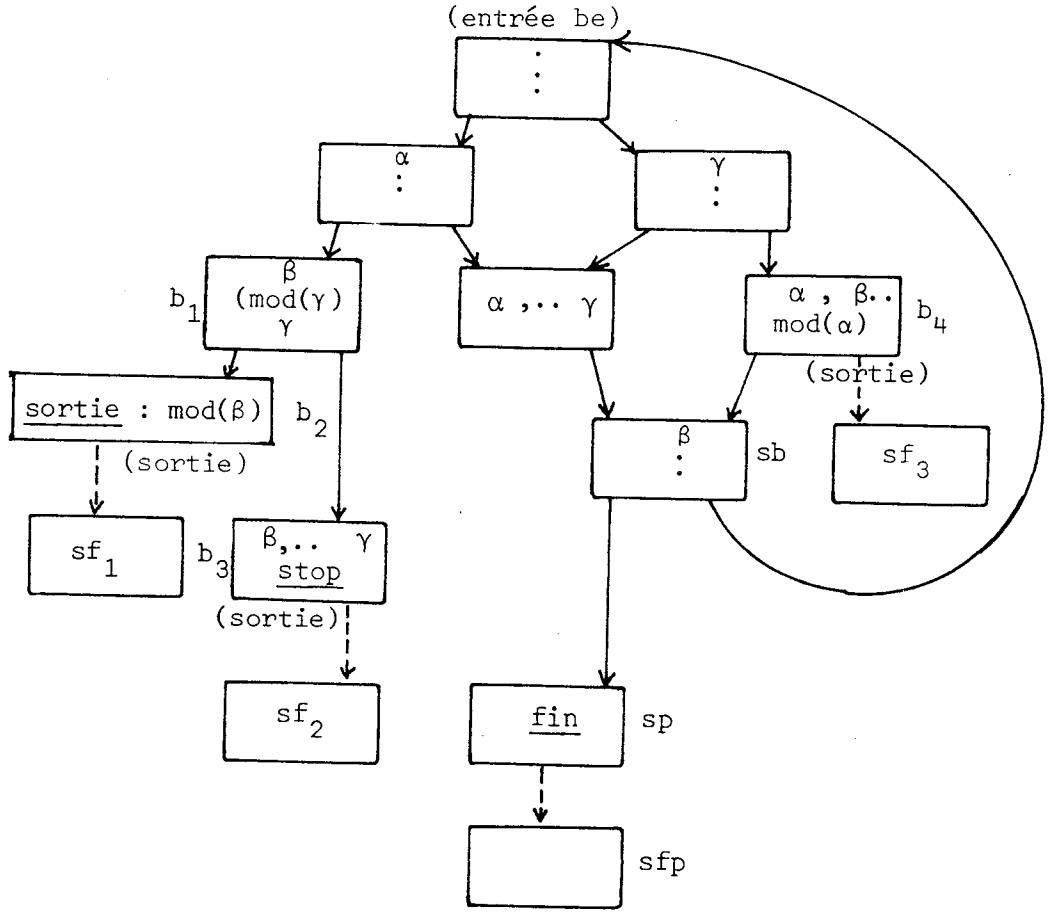


Figure 5. Un graphe de boucle propre G. Le graphe G_a associé à G. Le graphe G ne vérifie pas la propriété P.

La propriété de la cohérence globale à G d'une expression e est relative au graphe de procédure GP (cf. 7.2.2) dont G est un composant (cf. 7.2.4) ; elle peut s'énoncer ainsi : il existe sur tout chemin de GP partant de l'entr be de G une instruction qui évalue e.

Remarque 6. Les optimisations présentées étant traitées au niveau d'un corps de procédure la définition de la cohérence globale est relative à un graphe d procédure. La propriété de cohérence pourrait être définie au niveau d'un graphe qui correspond au programme intermédiaire tout entier ; ceci n'a pas été fait. Il s'ensuit que les deux propriétés de cohérence énoncées ci-dessus sont "approximatives" c'est-à-dire que la non-cohérence au niveau d'un graphe boucle propre ou au niveau d'un graphe de procédure n'implique pas la non-cohérence au niveau d'un graphe du programme intermédiaire tout entier.

Les propriétés de l'anticipabilité locale à G et de l'anticipabilité globale à G d'une expression e s'énoncent d'une façon analogue : l'expression e est anticipable localement (resp. globalement) en entrée de G s'il existe sur tout chemin c de G (resp. de GP, voir la cohérence globale) partant de l'entrée be de G une instruction qui évalue e et qui n'est pas précédée, sur c, par une instruction qui modifie e.

Le système d'anticipabilité attaché au graphe G_a associé à G est connu :

$$AE(e, b) = AL(e, b) + \neg ML(e, b) * \prod_{b' \in Suc(b)} AE(e, b'), \forall b \in B_a - \{sfp\} - \{sf_j\} \quad j \in [1, S]$$

avec $AS(e, b) = \prod_{b' \in Suc(b)} AE(e, b'), \forall b \in B_a - \{sfp\} - \{sf_j\}, j \in [1, S]$.

Le système sera dit local à G si les hypothèses en sorties fictives de G sont les hypothèses "pessimistes" suivantes :

$AE_\ell(e, f_k)$ = faux

pour toute sortie fictive $f_k \in \{sf_j\} \cup \{sfp\}$, $j \in [1, S]$.

La résolution du système local fournit les valeurs des coefficients AE_ℓ et AS_ℓ ; la propriété de l'anticipabilité locale à G d'une expression e est vérifiée si :

$AE_\ell(e, be)$ = vrai où be est l'entrée de G.

Le système sera dit global à G si l'on tient compte des valeurs qui "arrivent" aux sorties de G de "l'extérieur" ;
ces valeurs sont données par les coefficients (voir les notations introduites en 8.1.1.1) :

$AS(e, br_j)$, $j \in [1, S]$ et

$AS(e, brp)$.

Dans la phase ascendante initiale (cf. 8.2.2) on résoudra sur G un système global initial d'anticipation en prenant pour les hypothèses en sorties fictives de G les hypothèses "optimistes" suivantes (les valeurs des coefficients AS associées aux blocs de représentation de G n'étant pas encore connues à ce stade) :

pour toute sortie fictive $f_k \in \{sfp\} \cup \{sf_j\}$, $j \in [1, S]$ nous poserons :

$$AE_{gi}\ (e,\ f_k) = \underline{faux}$$

si la sortie s_k de G associée à f_k est une sortie absolue (cf. 7.2) et nou poserons

$$AE_{gi}\ (e,\ f_k) = \underline{vrai}$$

dans le cas contraire. Les hypothèses en sorties absolues tiennent compte du fait que l'optimisation globale de déplacement est traitée au niveau d'u corps de procédure (cf. Remarque 6.).

Dans les phases descendantes et ascendantes subséquentes, par contre, on résoudra sur G un <u>système global</u> d'anticipabilité en posant :

$$AE_g\ (e,\ sfp) = AS_g\ (e,\ brp) \text{ et}$$
$$AE_g\ (e,\ sf_j) = AS_g\ (e,\ br_j),\ j \in [1, S].$$

Les indices g (resp. gi) indiquent les coefficients obtenus par la résoluti d'un système global (resp. d'un système global initial).

La propriété de l'anticipabilité globale à G d'une expression e est vérifié si :

$$AE_g\ (e,\ be) = \underline{vrai} \text{ où be est l'entrée de G.}$$

En raison de la Remarque 6, nous avons l'inégalité suivante :

$$AE_g\ (e,\ be) \geq AE_\ell\ (e,\ be) \text{ où be est l'entrée de G.}$$

<u>Remarque 7</u>. L'égalité

$$AE_g\ (e,\ \text{entrée de G}) = AE_\ell\ (e,\ \text{entrée de G})$$

est vérifiée

- si MS (e, sb) = <u>vrai</u> et G est un graphe de boucle propre, ou
- si $(\prod_j MS\ (e, sj)) * MS\ (e, sp) = \underline{vrai}$, $j \in [1, S]$ et G est un graphe-composant quelconque.

Nous pouvons maintenant définir l'optimisation locale de déplacement des instructions : si G est un graphe de boucle propre alors une instruction d'un bloc b de G, instruction qui évalue une expression e (c'est-à-dire AL(e,b)=vrai) peut être déplacée en dehors de G si l'expression e vérifie les conditions suivantes :

C1 : la propriété de l'invariance dans G de e est vérifiée soit :
MS (e, sb) = faux = MS (e, sp)

C2ℓ : la propriété de l'anticipabilité locale à G de e est vérifiée soit :
AE_ℓ (e, be) = vrai

C3 : AL (e, b) * ¬ ME (e, b) = vrai où b est le bloc qui contient l'instruction (qui évalue e) à déplacer.

L'optimisation globale de déplacement est définie d'une façon analogue : il suffit de remplacer la condition C2ℓ par : C2g : AE_g (e, be) = vrai.

L'optimisation locale de déplacement est une optimisation partielle c'est-à-dire qu'une instruction qui n'est pas déplacée par l'optimisation locale peut l'être par l'optimisation globale de déplacement.

Reprenons l'exemple de la figure 5 et considérons d'abord les propriétés de la cohérence locale et de l'anticipation locale des expressions α, β et γ :

- α et β vérifient les deux propriétés
- γ vérifie la propriété de cohérence locale mais ne vérifie pas la propriété d'anticipabilité locale.

Comme MS (α, sb) = vrai les instructions α ne sont pas déplacées ; γ ne vérifie pas C2ℓ (ni C2g, en raison de la Remarque 7) et n'est pas déplaçable ; β par contre vérifie C1 et C2ℓ : toutes les instructions β sont déplaçables en dehors de G excepté l'instruction β du bloc b_3 qui ne vérifie pas la condition C3 (car ME (β, b_3) = vrai).

Remarquons que la nécessité de la condition C3 est due au fait que G ne vérifie pas la propriété P ; cette propriété rend en effet possible l'existence d'une instruction i qui évalue e telle que i se trouve dans un bloc pendant b de G ; l'occurrence de e évaluée par i peut vérifier C1 et C2 et peut ne pas vérifier C3 car ME (e, b) peut être vrai (c'est le cas de β du bloc b_3 de la figure 5.).

Cette remarque suggère un traitement simplifié des déplacements pour les graphes des boucles propres qui vérifient la propriété P.

Remarque 8. Lorsqu'une expression e n'est évaluée par aucune instruction d'un bloc pendant de G le traitement des déplacements des instructions qui évaluent e dans G sera le traitement défini par le Cas 2.

Cas 2) Traitement des déplacements au niveau d'un graphe de boucle propre G = (B, A) qui possède la propriété P.

Il résulte de la définition de l'invariance exposée au Cas 1. et de la propriété P de G que la propriété de l'invariance dans G d'une expression e est vérifiée si :

$$\sum_{b \in B - SP(G)} ML(e, b) = \underline{faux}.$$

Remarque 9. Si BP(G) est l'ensemble des blocs pendants de G alors le sous-graphe de G engendré par B - BP(G) est une composante fortement connexe (voir aussi la Remarque 5). Il s'ensuit que la propriété de l'invariance d'une expression e peut toujours être définie par :

$$\sum_{b \in B - BP(G)} ML(e, b) = \underline{faux}.$$

La détermination de la propriété d'invariance par un système de modifiabili dans le Cas 1 est justifiée par le fait que les coefficients ME doivent êtr calculés pour qu'on puisse vérifier la condition C3 de l'optimisation de déplacement.

La condition C2ℓ du Cas 1 relative à l'optimisation locale de déplacement peut être remplacée par :

C2ℓ' : la propriété de la cohérence locale à G de e est vérifiée.

Comme les blocs de SP(G) n'évaluent aucune expression la condition C2ℓ' es encore équivalente à :

DS_ℓ (e, sb) = vrai, où sb est la sortie de boucle de G.

Il résulte de ce qui précède que l'optimisation locale de déplacement consiste en le déplacement de toute instruction de G, instruction qui évalue

(c'est-à-dire AL(e,b)=vrai) lorsque l'expression e vérifie les deux conditions :

C1' : $\sum_{b \in B - SP(G)} ML(e, b) = \underline{faux}$

C2ℓ' : $DS_{\ell}(e, sb) = \underline{vrai}$.

Les conditions de l'optimisation globale de déplacement sont les deux conditions suivantes :

C1' et

C2g du Cas 1.

Remarque 10. (elle correspond à la Remarque 7.)

Si $\sum_{b \in B - BP(G)} ML(e, b) = \underline{vrai}$ alors l'optimisation globale de déplacement des instructions qui évaluent e dans G donne les mêmes résultats que l'optimisation locale.

La remarque qui suit s'applique aussi bien au Cas 1 qu'au Cas 2.

Remarque 11. Lorsqu'une expression e vérifie la condition C1 (Cas 1) ou C1' (Cas 2) elle vérifie
également la condition C2ℓ (Cas 1) ou C2ℓ' (Cas 2) si e est évaluée par une instruction qui se trouve dans un bloc d'articulation (cf.7.4) de G.

8.1.2.2.3 - Exploitation des informations que représentent les ensembles $MODB_i$ -

Rappelons que les ensembles $MODB_i$ (cf. 4.2.2) sont constitués des variables-source modifiées par une instruction bloc i ou boucle i ; ces variables-source sont des composants (cf. 6.5) des expressions du langage intermédiaire.

Nous avons associé (cf. 7.2.4) à tout graphe de programme G qui est un composant d'un graphe de procédure GP un bloc de représentation principal brp_G.

Les considérations qui suivent concernent la condition C qui doit être vérifiée pour que le coefficient

ML (e, brp_G), ∀e (cf. 8.1.2.1)

puisse être déduit de l'ensemble $MODB_G$ de la façon suivante (cf. 6.5) :

ML (e, brp_G) = <u>vrai</u> ssi $\exists$ op $\in$ MOD (e) tel que op $\in$ $MODB_G$.

A tout graphe-composant G d'un graphe de procédure GP peut être associé d'une façon évidente un sous arbre SASF (G) de l'arbre ASF associé à GP. Lorsque G n'est pas un graphe d'un fief-procédure, un graphe de procédure ou un graphe de procédure propre l'arbre SASF (G) est un sous-arbre propre de ASF et l'ensemble des blocs de représentation de G n'est pas vide.

La <u>condition C</u> peut alors s'énoncer de la façon suivante : un graphe-composant G vérifie C lorsque à chacun des graphes des fiefs associés aux fiefs qui sont les sommets de SASF (G) correspond un unique bloc de représentation (qui est un bloc de représentation principal).

En d'autres termes, si G vérifie C alors chacun des graphes des fiefs associés aux sommets de l'arbre SASF (G) ne possède qu'une seule sortie (qui est la sortie principale). Ainsi, si G est un graphe de boucle propre et si G vérifie C alors il possède la propriété P et SP(G) = {sp} (voir le sous-chapitre précédent).

<u>Remarque 12</u>. (voir la Remarque 9). La propriété de l'invariance d'une expression e dans un graphe de boucle propre G = (B, A) qui vérifie la condition C est vérifiée si :

$\nexists$ op $\in$ MOD (e) tel que op $\in$ $MODB_G$ (ce qui équivaut à $\sum_{b \in B}$ ML (e, b) = <u>faux</u>).

8.1.2.2.4 - Sécurité (cohérence) et efficacité des optimisations considérées

Nous rappellerons ici brièvement la contrainte de sécurité (cf. la propriété de cohérence de 8.1.2.2.2) qui doit être respectée lors des déplacements des instructions et nous considérerons l'efficacité des optimisations proposées.

Considérons l'exemple de la figure 6 : l'expression e = <u>div</u> a, b vérifie la propriété d'invariance sans vérifier la propriété de cohérence. Si l'instruction i qui évalue e est placée dans le bloc bp alors une interruption peut survenir si b = 0, interruption qui a été impossible auparavant.
D'où l'introduction de la contrainte de sécurité qu'exprime la propriété de cohérence de 8.1.2.2.2.

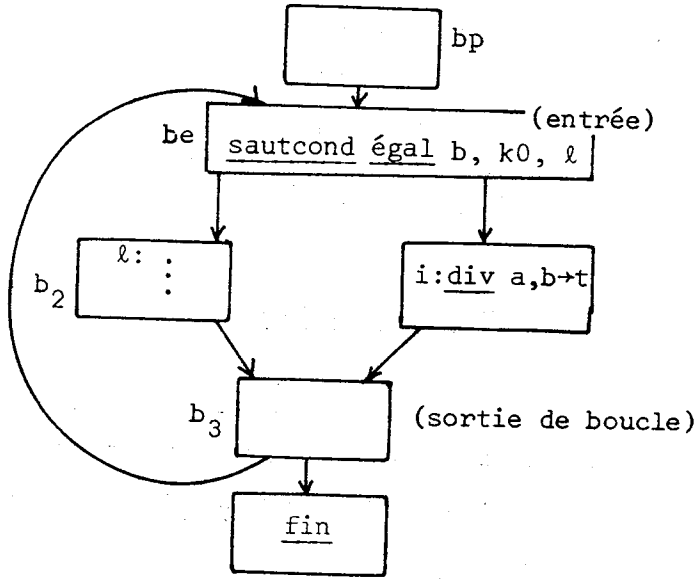


Figure 6. Un graphe de boucle propre ; l'expression associée au temporaire t est supposée invariante dans la boucle.

On peut cependant observer que la contrainte de sécurité est nécessaire seulement lorsque l'évaluation à déplacer est susceptible de provoquer une interruption.
On dira qu'une optimisation est efficace si :

- elle diminue l'espace nécessaire pour le code du programme ou pour les données du programme ;
- elle diminue le nombre d'évaluations des expressions du programme.

L'élimination des expressions redondantes est efficace car elle entraîne toujours une diminution de l'espace nécessaire pour le code et une diminution du nombre d'évaluations ; elle peut en outre entraîner une diminution de l'espace nécessaire pour les données.

Le déplacement des instructions évaluant des expressions en dehors d'une boucle (tel qu'il est défini en 8.1.2.2.2) est efficace lorsque la boucle est exécutée au moins deux fois ce qui entraîne une diminution du nombre d'évaluations. Lorsque la boucle est exécutée une seule fois le gain en nombre d'évaluations peut être nul et le déplacement n'est efficace que si plusieurs instructions évaluant une même expression sont déplacées ce qui entraîne une diminution de l'espace nécessaire pour le code.

Notons que la contrainte de sécurité (en tant que la propriété de cohérence) est aussi un facteur essentiel de l'efficacité des déplacements ; on peut voi sur l'exemple de la figure 6 que le non-respect de cette contrainte fait dépendre l'efficacité du déplacement de l'instruction i de la fréquence d'exé cution du circuit [be, b_2, b_3].

8.2 - Organisation des optimisations proposées -

Nous exposerons dans ce sous-chapitre une approche possible à l'optimisation d'éliminations et de déplacements des instructions d'un programme intermédiai Les avantages et les inconvénients de l'approche proposée seront brièvement examinés à la fin du sous-chapitre 8.2.

8.2.1 - Niveaux des optimisations -

On appellera occurrence d'une expression d'un programme intermédiaire une occu rence de l'instruction qui évalue (cf. 8.1.2.1.) l'expression considérée.

Pour optimiser les occurrences d'une expression e c'est-à-dire pour appliquer à occurrences l'optimisation d'éliminations et celle des déplacements, il faut examiner au moins tous les graphes des programmes d'un programme intermédiair qui contiennent des occurrences de e.

Nous avons vu que pour effectuer une optimisation il faut résoudre au préalab des systèmes d'équations booléennes. On dira que le niveau d'une optimisation est un graphe de programme GP si les informations nécessaires à l'optimisatio en question sont obtenues par la résolution des systèmes sur le graphe GP lui-même ou sur les graphes-composants (cf. 7.2.4.) de GP.

L'optimisation sera dite globale si son niveau est un graphe de procédure et locale si son niveau est un graphe-composant d'un graphe de procédure.
Ces termes ont déjà été utilisés en 8.1.2.

Une optimisation locale ou globale d'une expression e sera dite totale (ou complète) si la prise en compte du programme intermédiaire tout entier ne permet pas de mettre en évidence d'autres optimisations de e. Dans le cas contraire une optimisation sera dite partielle.

Les optimisations dont le niveau est le programme intermédiaire tout entier ne seront pas considérées (voir 8.2.5) ; par conséquent, lorsque le niveau d'une optimisation (globale) d'éliminations et de déplacements sera un graphe de procédure PG on résoudra sur PG les systèmes (globaux) de disponibilité et d'anticipabilité avec les hypothèses fausses en entrée et en sorties de PG.

Nous décrirons par la suite une méthode possible de collection d'informations nécessaires pour les optimisations globales (et locales) d'éliminations et de déplacements.

8.2.2 - Collection des informations pour les optimisations globales -

Deux approches sont possibles : l'approche directe et les approches par décomposition des graphes des procédures (cf. 7.2.4).

Dans l'approche directe, les divers systèmes d'équations booléennes sont résolus sur chacun des graphes des procédures du programme intermédiaire.

Dans les approches par décomposition on résout les systèmes qu'on a appelés globaux (cf. 8.1.2.2) successivement sur chacun des composants d'un graphe de procédure ; les informations sont passées d'un composant à l'autre au niveau des blocs de représentation (cf. 7.2.4).

Nous décrirons par la suite, sur un exemple schématique la collection des informations pour les optimisations globales effectuée par une approche par décomposition.

Les figures 7 à 10 illustrent la décomposition choisie des graphes des procédures. La figure 7 représente l'arbre ASF d'un corps de procédure ; les somme

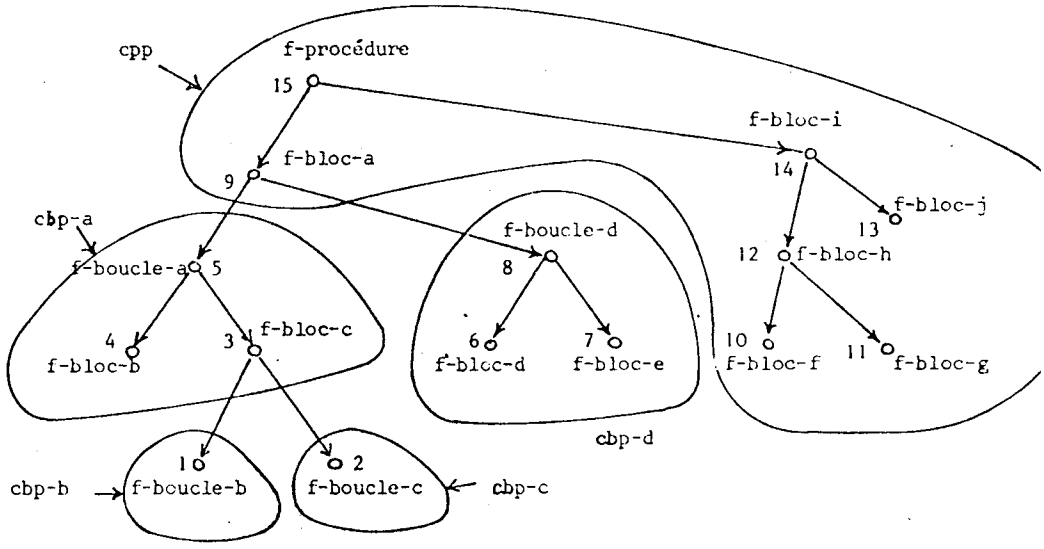


Figure 7. L'arbre ASF et un ordre de base de ses sommets.
Notations : cbp = corps de boucle propre,
cpp = corps de procédure propre.

de ASF (en tant que graphe orienté avec Γ^{-} (racine de ASF) = $\emptyset$) sont numéroté suivant un ordre de base (cf. 7.3) ; le graphe de la procédure qui correspon à l'arbre ASF est décomposé par la deuxième décomposition (cf.7.2.4) en un graphe de procédure propre et en des graphes des boucles propres ; ces graphe composants (qui sont les sommets d'un arbre ARB des boucles, cf. 7.2.4) sont également numérotés suivant un ordre de base des sommets de l'arbre ARB. Ce der est représenté dans la figure 8.

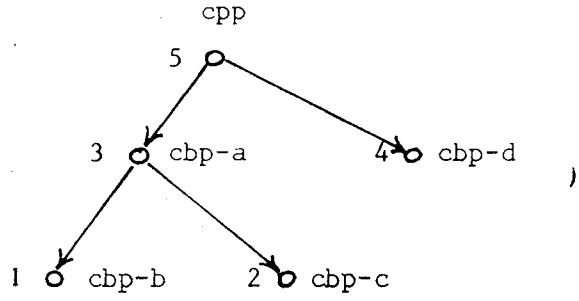


Figure 8. L'arbre ARB déduit de la figure 7 et un ordre de base de ses sommets.

Le graphe de procédure propre (la racine de ARB) est décomposé à son tour par le première décomposition. L'arbre AC des composants qui résultent de la

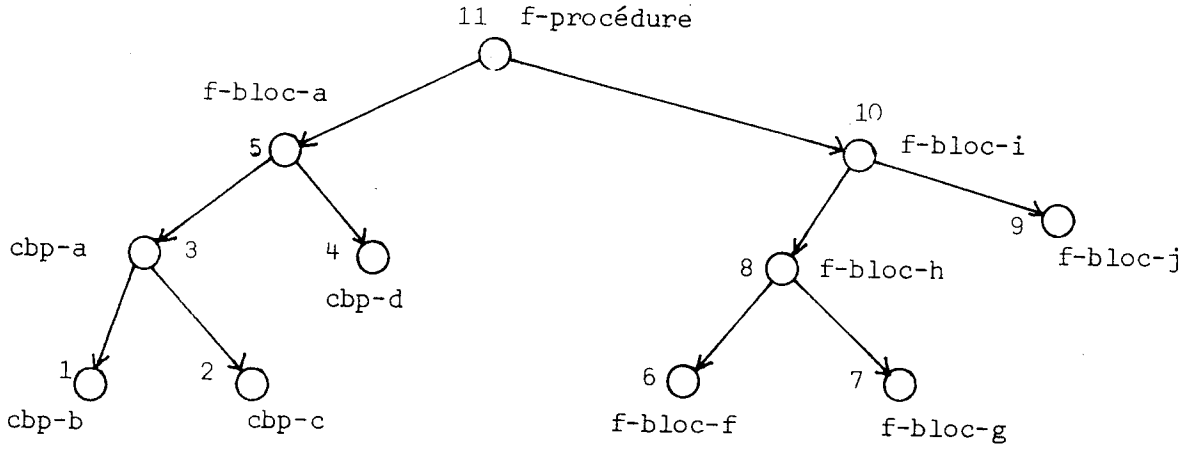


Figure 9. L'arbre AC des composants finals et un ordre de base de ses sommets.

combinaison de ces deux décompositions est représenté dans la figure 9. Un schéma des graphes-composants de la figure 9 avec leurs blocs de représentation (cf. 7.2.4) est représenté dans la figure 10.

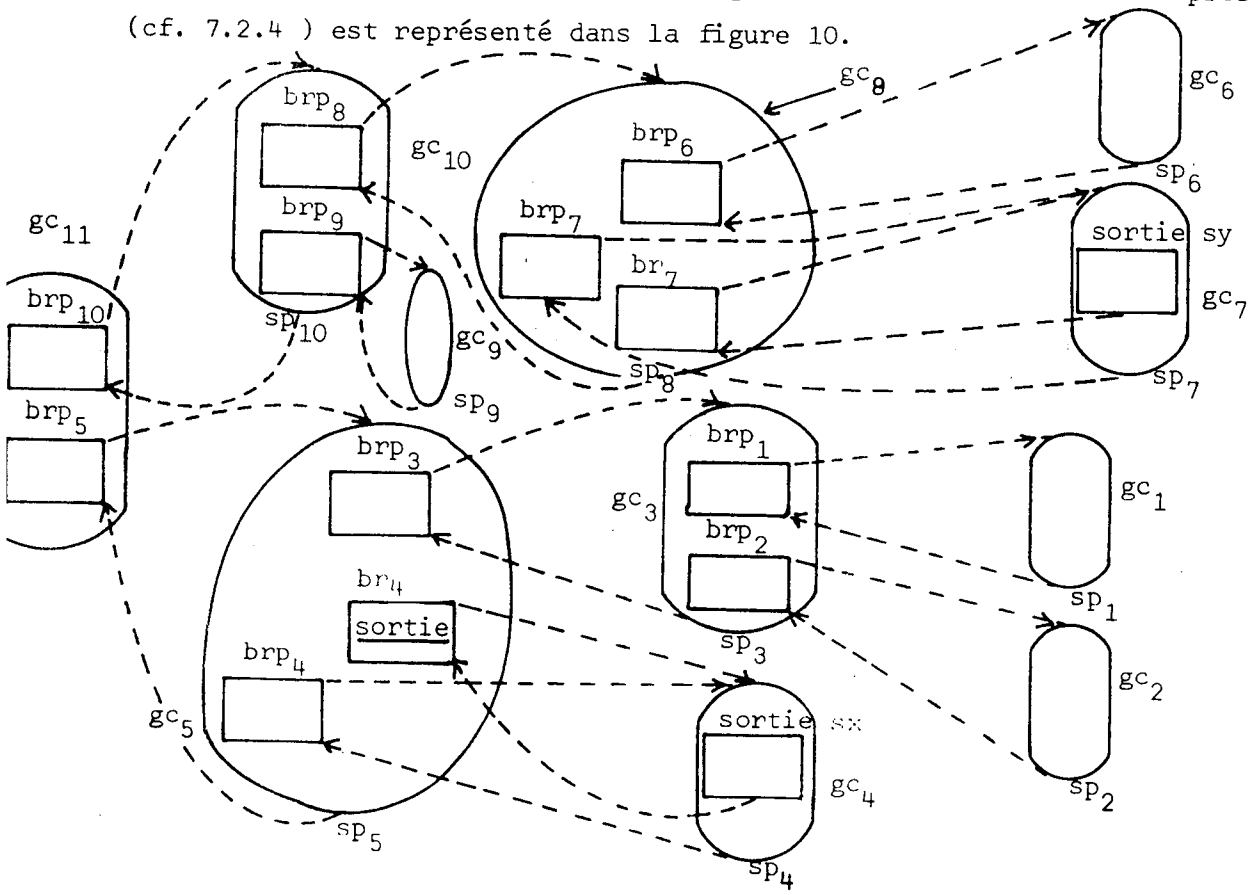


Figure 10. Le schéma des graphes-composants de la figure 9.

Notations : gc = graphe-composant, br = bloc de repr.,
brp = br principal, sp = sortie principale,
□ = bloc élémentaire.

La collection des informations (les coefficients DE, DS, AE, AS, ME, MS) commence par une phase ascendante initiale et se poursuit alternativement par une phase descendante et une phase ascendante jusqu'à la "stabilisation" (voi plus loin) (Cette méthode est fondée sur celle de [5]) : dans une phase ascendante l'ordre de résolution des systèmes d'équations booléennes sur les composants du graphe de procédure considéré est un ordre de base 0 de ces composan et dans une phase descendante l'ordre de résolution est l'ordre inverse $\overline{0}$ des composants du graphe de procédure considéré.

On utilisera par la suite les notations suivantes (cf. 8.1.1.1) :

AC : l'arbre des composants du graphe de procédure considéré ;
gc_i : le graphe-composant traité ;
brp_i : le bloc de représentation principal du graphe gc_i ;
br^i_j : les blocs de représentation non-principaux du graphe gc_i ;
be_i : le bloc d'entrée de gc_i ;
sp_i : la sortie principale de gc_i ;
s^i_j : les sorties non-principales de gc_i ; un bloc de représentation br^i_j correspond à une sortie s^i_j ;
$SAC(gc_i)$: le sous-arbre de AC de racine gc_i ;
ef_i : l'entrée fictive de gc_i ;
sfp_i : la sortie fictive (principale) associée à sp_i de gc_i ;
sf^i_j : les sorties fictives associées aux sorties s^i_j ;
$predrep_i$: le bloc-prédécesseur des blocs de représentation du graphe-composant gc_i (cf. 7.2.4).

Rappelons qu'un bloc de représentation principal brp_i est initialement vide (cf. 7.2.4) ; des instructions pourront être insérées (cf. 8.2.4) dans le bloc brp_i au cours de l'optimisation de déplacement. Un bloc de représentation non-principal est constitué d'une instruction sortie et ne sera pas l'objet des insertions.

Le processus complet de la collection des informations consiste donc :

- en une phase ascendante initiale et
- en des itérations successives dont chacune comprend :
  - une phase descendante et
  - une phase ascendante.

Dans la phase ascendante initiale on résout les systèmes que nous avons appelés systèmes globaux initiaux et dans toute itération subséquente les systèmes résolus-quelque soit la phase considérée- sont les systèmes que nous avons appelés systèmes globaux tout court.

Nous décrirons par la suite les opérations effectuées au cours d'une phase ascendante et au cours d'une phase descendante. Dans la description qui suit les coefficients ne sont pas indéxés ; il suffit de supposer que les coefficients sont indéxés par gi s'il s'agit de la phase ascendante initiale et par g dans le cas contraire.

Phase ascendante

Les graphes-composants sont traités dans l'ordre 0 ; soit gc_i le i-ième graphe-composant traité. Après la résolution des systèmes (cf. 8.1.2.2) sur le graphe g_i les coefficients aux sorties ou à l'entrée de gc_i sont transmis aux blocs de représentation correspondants de gc_i ; ces blocs (voir la figure 10) se situent dans un composant gc_j qui précède gc_i dans l'arbre AC et qui, en vertu de l'ordre 0, n'a pas encore été traité. La transmission des coefficients dépend de la nature du graphe gc_i (voir la propriété P de 8.1.2.2.2 et la condition C de 8.1.2.2.3) et sera décrite par la suite.

Dans le cas général la transmission des coefficients DS, AE, et MS, obtenus par la résolution du système de disponibilité, du système d'anticipabilité et du système de modifiabilité consiste à poser :

$DL\ (e, brp_i) = DS\ (e, sp_i)$,

$AL\ (e, brp_i) = AE\ (e, be_i)$,

$ML\ (e, brp_i) = MS\ (e, sp_i)$,

$DL\ (e, br^i_j) = DS\ (e, s^i_j),\ \forall_j$,

$AL\ (e, br^i_j) = AE\ (e, be_i),\ \forall_j$,

$ML\ (e, br^i_j) = MS\ (e, s^i_j),\ \forall_j$.

Notons que les coefficients AL et DL des blocs de représentation n'ont pas la signification habituelle (telle qu'elle est définie en 8.1.2.1) ; ainsi, AL (e, br) = vrai ou DL (e, br) = vrai, où br est un bloc de représentation, n'implique pas la présence, dans br, d'une instruction évaluant e.

Nous définirons par conséquent un coefficient EV attaché à tout bloc de représentation principal brp ; EV (e, brp) sera mis à vrai lorsqu'une occurrence de e sera insérée dans le bloc brp par l'optimisation de déplacement. Le coefficient EV remplacera le coefficient AL dans la condition relative à l'optimisation d'élimination (cf. 8.1.2.2.1) ; ainsi, pour un bloc de représentation principal brp, cette condition deviendra :

EV (e, brp) * DE (e, brp) = vrai.

Deux cas particuliers de la transmission des coefficients seront distingués par la suite.

Cas 1) Le graphe gc_i est un graphe de boucle propre vérifiant la condition C de 8.1.2.2.3 ou un graphe de fief vérifiant C (gc_i a un seul bloc de représentation brp_i) :

- le coefficient ML (e, brp_i) est défini par :
  $\exists$ op $\in$ MOD (e) tel que op $\in$ $MODB_i$;
- les coefficients DL (e, brp_i) et AL (e, brp_i) sont définis comme au cas général.

Cas 2) Le graphe gc_i est un graphe de boucle propre qui ne vérifie pas la condition C de 8.1.2.2.3 mais qui possède la propriété P (voir le Cas 2 de 8.1.2.2.2) :

- le coefficient ML (e, brp_i) est donné par la valeur de
  $\sum_{b \in B - SP(gc_i)}$ ML (e, b) ;
- tous les autres coefficients sont définis comme au cas général.

A la fin de la phase ascendante initiale les informations relatives à la modifiabilité (et à l'invariance) représentées par les coefficients ML, MS et ME sont exactes ; rappelons en effet que la définition du coefficient ML (cf.8.1.2.1) tient compte des ensembles MODAP (cf. 4) et que les coefficients ML reflètent par conséquent aussi les modifications implicites et indirectes que peuvent provoquer les instructions d'appel.

Phase descendante

Les graphes-composants sont traités dans l'ordre inverse $\overline{0}$; les systèmes à résoudre doivent être par conséquent des systèmes globaux (cf. 8.1.2.2.). Soit gc_i le i-ième graphe-composant traité. Nous rappellerons par la suite les hypothèses à l'entrée et aux sorties de gc_i pour les systèmes globaux :

- les coefficients DE_g et DS_g sont obtenus par la résolution du système global de disponibilité sur gc_i avec l'hypothèse suivante à l'entrée (cf. 8.1.2.2.1) :
  $DS_g\ (e, ef_i) = DS_g\ (e, predrep_i)$ où $predrep_i$ est le prédécesseur des blocs de représentation de gc_i ;

- les coefficients AS_g et AE_g sont obtenus par la résolution du système global d'anticipabilité sur gc_i avec les hypothèses suivantes en sorties de gc_i (cf. 8.1.2.2.2) :
  $AE_g\ (e, sfp_i) = AS_g\ (e, brp_i)$,
  $AE_g\ (e, sf^i_j) = AS_g\ (e, br^i_j)$, $\forall j \in [1, S]$.

Le processus de la collection des informations esquissé ci-dessus peut être représenté par une séquence S :

$$fa_1, \underbrace{(fd_2, fa_3)^1}_{\text{itération 1}}, \ldots, \underbrace{(fd_{2p}, fa_{2p+1})^p}_{\text{itération p}}, \ldots$$

où fa désigne une phase ascendante et fd une phase descendante.

La convergence (ou la "stabilisation") du processus peut se déduire en le comparant aux méthodes similaires de [5, 33, 34].

La convergence de la séquence S signifie qu'il existe un entier k tel que :

itération (k-1) = itération k

soit : les valeurs des coefficients relatifs aux systèmes résolus, valeurs obtenues en fin de l'itération k-1 sont identiques aux valeurs obtenues en fin de l'itération k.

Le test de convergence peut être fait, par exemple, en considérant les valeurs des coefficients AL et DL relatifs aux blocs de représentation de tous graphes-

composants du graphe de procédure considéré.

Ainsi, il suffit de comparer, en fin de toute phase ascendante (2p+1), p > 0, de l'itération p, les valeurs des coefficients qui sont transmises aux blocs de représentation :

$$AL^{2p+1}(e, br), \forall e, \forall br,$$

$$DL^{2p+1}(e, br), \forall e, \forall br$$

aux valeurs de ces mêmes coefficients qui sont transmises au cours de la phas ascendante 2(p-1) +1 précédente.

Remarquons que le test de convergence pourrait être fait également en fin d'une phase descendante 2p, p > 1 de l'itération p ; dans ce cas, si le test se révèle positif, il est bien sûr inutile d'effectuer la phase ascendante (2p+1) de l'itération p.

8.2.3 - Prise en compte des portées et de l'invariance -

Un programme intermédiaire peut comporter un grand nombre d'expressions et un choix des expressions à optimiser doit généralement être fait. Ce choix ne peut être fondé, lorsqu'on ne dispose pas d'autres critères, que sur le nombre d'occurrences des expressions.

On appellera expression commune une expression associée à un temporaire commun (cf. 5.4.2.1) ; toute expression commune possède au moins deux occurrences.

Nous supposerons par la suite que l'ensemble des expressions à optimiser choisi sera l'ensemble des expressions optimisables ; cet ensemble est déterminé en fonction du nombre minimal nécessaire d'occurrences d'une expression e pour que e puisse être théoriquement l'objet d'une optimisation d'élimination ou de déplacement.

L'applicabilité théorique des optimisations à des expressions sera examinée par la suite et déterminera l'ensemble des expressions optimisables.

L'optimisation de la redondance locale (cf. 8.1.2.1) ne sera appliquée qu'à des expressions communes possédant au moins deux occurrences dans un même bloc élémentaire.

L'optimisation d'élimination au niveau d'un graphe de programme ne sera appliquée qu'à des expressions communes possédant au moins deux occurrences dans au moins deux blocs élémentaires distincts.

L'optimisation de déplacement au niveau d'un graphe de programme ne sera appliquée qu'à des expressions suivantes :

- expressions communes possédant au moins deux occurrences dans au moins deux blocs élémentaires distincts d'un graphe de boucle (associé à un corps de boucle, cf. 4) ;

- expressions possédant au moins une occurrence dans un bloc d'articulation (cf. 7.4) d'un graphe de boucle.

Les optimisations globales fondées sur la collection d'informations par l'approche directe (cf. 8.2.2) doivent traiter en même temps ("en parallèle") toutes les expressions optimisables qui restent optimisables à la suite de l'optimisation de la redondance locale.

L'avantage des approches par décomposition consiste en ce que ces approches permettent de restreindre le nombre des expressions optimisables lorsqu'on prend en compte les portées (cf. 6.5) et l'invariance de ces expressions.

Nous supposerons par la suite que la collection des informations pour les optimisations globales est effectuée par l'approche par décomposition décrite en 8.2.2.

8.2.3.1 - Portées des expressions -

Les restrictions qu'entraîne la prise en compte des portées des expressions optimisables sont évidentes.

Ainsi, lorsqu'on doit résoudre un système global sur un graphe-composant gc_i les expressions optimisables à prendre en considération sont celles dont les portées (cf. 6.4.3 et 6.5) vérifient la condition C4 suivante :

- portée (e) > racine (SASF (gc_i)), ou
- portée (e) est l'un des fiefs de portée (cf. 4.1) du graphe gc_i.

Un exemple est donné par la figure 11 : les portées des expressions α, β, γ et δ sont supposées être, respectivement, les fiefs f-bloc-b, f-bloc-d, f-bloc-c et f-bloc-a.

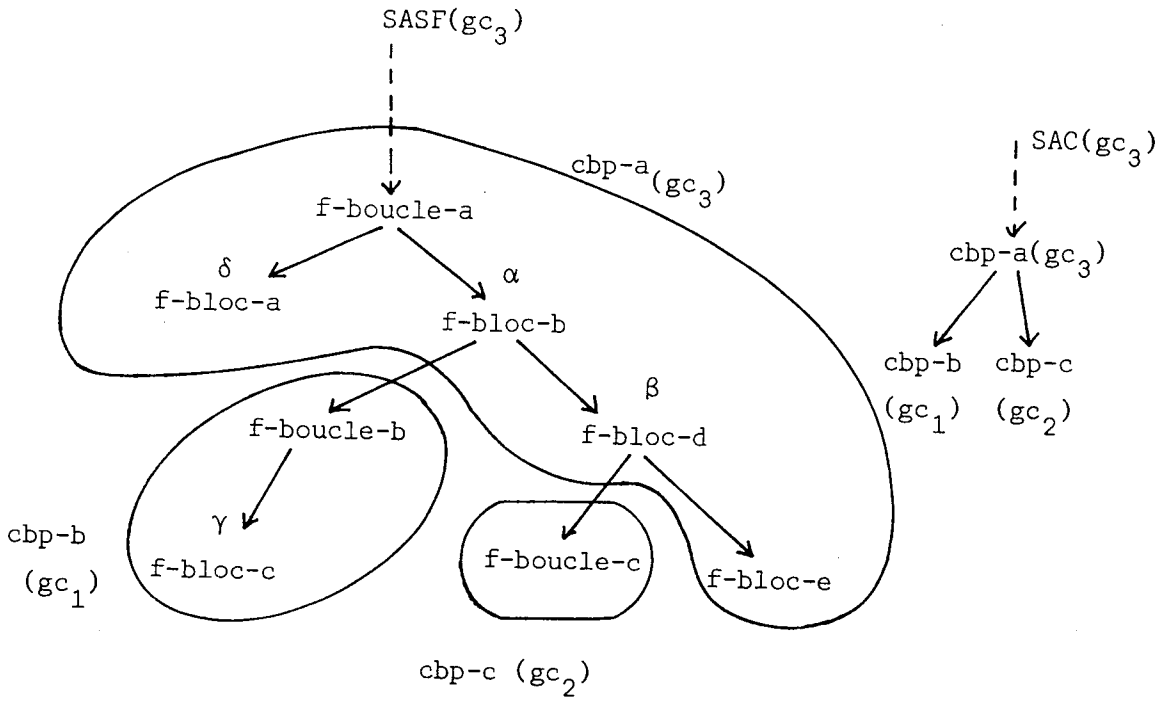


Figure 11. Le sousarbre SASF (gc_3) d'un arbre ASF avec ses graphes-composants et le sous arbre SAC (gc_3) de l'arbre AC des graphes-composants. (Voir les notations de la figure 7)

Lors de la résolution d'un système global sur les composants gc_1, gc_2 et gc_3 les expressions à prendre en considération sont les suivantes :

- pour le gc_1 : α et γ ;
- pour le gc_2 : α et β ;
- pour le gc_3 : α, β et δ.

Considérons le cas général d'une expression e dont la portée (qui est un fief de portée) est "contenue" dans un graphe de procédure GP.

Soit GC l'ensemble des graphes-composants de GP et soit gc (e) ∈ GC le graphe-composant qui "contient" la portée de e. Lors de la collection des informations pour l'optimisation globale de e il suffit alors de résoudre les systèmes sur les graphes-composants qui sont en accord avec la condition C4.
Soit GC (e) ⊆ GC l'ensemble de ces graphes-composants. L'ensemble GC (e) contient le graphe-composant gc (e) et les graphes-composants qui se trouvent "dans la portée" de e (Notons que GC (e) = GC dans le cas particulier où la portée de e n'est pas "contenue" dans GP c'est-à-dire e est "globale" à GP).

Reprenons l'exemple de la figure 11 ; nous avons

GC (α) = $\{gc_1, gc_2, gc_3\}$;

GC (β) = $\{gc_2, gc_3\}$;

GC (γ) = $\{gc_1\}$ avec gc_1 = gc (γ) ; GC (δ) = $\{gc_3\}$.

Lors de la résolution des systèmes sur les graphes de GC (e) $\neq$ GC, les hypothèses en entrées fictives et en sorties fictives de ces graphes -hypothèses relatives à e- peuvent être les mêmes que dans le cas général (cf. 8.2.2)- où les systèmes sont résolus sur tous les graphes-composants de GC. La raison en est évidente si l'on se rappelle que gc (e) $\in$ GC (e) "contient" la portée de e et que, par conséquent, l'instruction

<u>fief</u> i, avec gc(e) = gc_i

- instruction qui commence le programme intermédiaire associé à gC (e) - modifie e (cf. 8.1.2.1).

Remarquons que dans le cas où

GC (e) = {gc(e)}

l'optimisation globale de e (fondée sur les coefficients obtenus par les systèmes globaux) donne les mêmes résultats que l'optimisation locale de e (fondée sur les coefficients obtenus par les systèmes locaux de 8.1.2.2).

Notons encore que l'optimisation globale d'une expression e -optimisation qui est effectuée au niveau du graphe de procédure GP qui "contient" le fief de portée qui est la portée de e- est totale (cf. 8.2.1).

8.2.3.2 - Prise en compte de l'invariance -

Nous expliquerons dans ce sous-chapitre l'avantage que représente la détermination de la propriété d'invariance des expressions. Cette propriété, rappelons-le, est liée aux valeurs des coefficients ML, ME et MS évalués -dans le cas général (cf. 8.1.2.2.2) - par un système de modifiabilité.

La propriété d'invariance est une condition nécessaire au déplacement des occurrences d'expressions. La vérification de la propriété d'invariance permet donc d'éliminer les expressions optimisables (cf. 8.2.3) qui ne pourront pas être déplacées.

La signification de l'élimination d'une expression optimisable e est la suivante :

- si e est sujette théoriquement (cf. 8.2.3) à la fois à l'optimisation de déplacement et à l'optimisation d'élimination alors l'élimination de e signifi que e n'est pas traitée par les systèmes d'anticipabilité car elle n'est pas déplaçable ;

- si e est sujette uniquement à l'optimisation de déplacement alors e est éliminée de l'ensemble des expressions optimisables au niveau du graphe de procédure considéré.

Dans le cas où tous les graphes-composants d'un graphe de procédure GP vérifient la propriété P ou la condition C (cf. 8.1.2.2.2) l'invariance des expressions est donnée par les coefficients ML ou par les ensembles MODB. Les expressions optimisables (dans GP) qui ne vérifient pas la propriété d'invariance peuvent alors être éliminées avant même qu'on procède à la collection des informations (cf. 8.2.2).

Dans le cas général toutefois (c'est-à-dire lorsque existent des graphes-composants qui ne vérifient pas la propriété P) on ne peut procéder à l'élimination qu'à la fin de la phase ascendante initiale du processus de collection d'informations. En effet, à la fin de cette phase les coefficients ML, MS, ME sont connus (cf. 8.2.2).

La condition d'élimination d'une expression e sera présentée par la suite.

Soit GP le graphe de procédure considéré ; soit GC l'ensemble des graphes-composants de GP et GCB $\subset$ GC l'ensemble des graphes des boucles propres de GC. Si e est "déclarée" dans GP soit gc(e) le graphe-composant (gC (e) $\in$ GC) qui "contient" la portée de e et soit GC (e) l'ensemble défini dans le sous-chapitre précédent.

Nous définirons l'ensemble GCB (e) $\subset$ GCB ainsi :

- si e n'est pas "déclarée" dans GP alors
  GCB (e) = GCB ;
- si e est "déclarée" dans GP alors
  GCB (e) = (GC (e) $\cap$ GCB) - gc(e)

L'ensemble GCB (e) est tout simplement l'ensemble des graphes des boucles propres de GP qui se trouvent "dans la portée" de e.

Rappelons qu'à la fin de la phase ascendante initiale au plus tard on dispose des valeurs des coefficients ML aux blocs de représentation principaux qui représentent les graphes-composants de l'ensemble GC. La condition d'élimination de e est alors donnée par la valeur de l'expression :

$EL = \prod_{i \in I} ML(brp_i)$, où les indices de I correspondent aux corps des boucles propres de l'ensemble GCB (e).

L'expression e sera éliminée si EL = vrai c'est-à-dire si e est modifiée dans chacun des graphes des boucles propres qui se trouvent "dans la portée" de e.

8.2.4 - Eliminations et déplacements des instructions -

A la fin du processus de collection d'informations (décrit en 8.2.2) on peut procéder aux optimisations globales d'élimination et de déplacement, dans cet ordre. On suppose que les redondances locales (au niveau des blocs élémentaires) ont été déjà traitées.

Nous décrirons par la suite le processus de déplacement des occurrences d'expressions.

L'optimisation de déplacement d'une expression e consiste :

- en l'insertion (si c'est nécessaire) d'une occurrence nouvelle de e dans un bloc de représentation principal qui représente un graphe de boucle propre et

- en la suppression des occurrences déplaçables de e (cf. la condition C3 de 8.1.2.2.2), occurrences qui se trouvent dans le graphe de boucle propre considéré.

Dans le cas simple où tous les graphes des boucles propres vérifient la propriété P la suppression est simple. Si, par contre, un graphe de boucle propre gc_i ne vérifie pas la propriété P la suppression des occurrences de e dans gc_i doit tenir compte non seulement de la condition C3 (cf. 8.1.2.2.2) mais aussi du fait que l'insertion d'une occurrence nouvelle de e dans le bloc brp_i peut créer de nouvelles redondances dans le graphe gc_i.

Nous expliquerons par la suite le processus de déplacement d'une expression e.

Les notations utilisées seront celles de 8.2.3.2 (et de 8.2.2) :

- GP est le graphe de procédure considéré
- GCB (e) est l'ensemble des graphes des boucles propres de GP qui se trouvent "dans la portée" de e.

Les graphes-composants de GCB (e) correspondent aux sommets de certains sous-graphes (qui sont des arbres) de l'arbre AC des composants de GP (l'arbre AC est considéré ici en tant que graphe ; l'orientation est telle que la racine de AC n'a aucun prédécesseur).

Reprenons l'exemple de la figure 9 (8.2.2) et supposons que la portée de l'expression e considérée est le fief de portée désigné par f-procédure (voir figure 9). Les sous-graphes de AC de la figure 9 dont les sommets correspondent aux éléments de GCB (e) sont alors donnés par la figure 12.

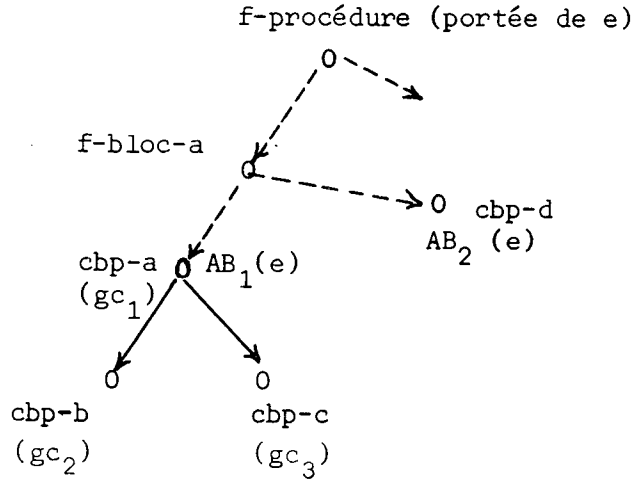


Figure 12. Les arbres AB_1 (e) et AB_2 (e) sont les deux sous-graphes de AC de la figure 9 dont les sommets correspondent aux éléments de GCB (e) = {cbp - a, cbp - b, cbp - c, cbp - d}.

Ces sous-graphes de AC seront désignés par AB_i (e). Les sommets d'un arbre AB_i (e) correspondent aux boucles qui peuvent "contenir" des occurrences déplaçables de e.

Nous considérerons par la suite un seul de ces arbres : soit AB (e) l'arbre considéré. Tout sommet de AB (e) représente un graphe-composant gc_i (qui est un graphe de boucle propre) ; un graphe gc_i contient les blocs de représentation principaux représentant les graphes gc_i qui sont les prédécesseurs de gc_i dans AB (e).

Les valeurs des coefficients ML et AL attachés aux blocs de représentation principaux des graphes-composants représentés par les sommets de AB (e) détermineront les possibilités de déplacement des occurrences de e.

Les sommets de AB (e) seront numérotés suivant un ordre de base inverse (cf. 7.3) ; le graphe-composant qui est le prédécesseur -dans AC- de la racine de AB (e) aura le numéro 0. Aux blocs de représentation principaux qui correspondent aux sommets de AB (e) sera associée la structure de l'arbre AB (e) : voir les figures 13 et 14. L'arbre des blocs de représentation ainsi obtenu sera noté ABR (e) et le prédécesseur (s'il en existe un) d'un sommet brp_i de ABR (e) sera noté pred (brp_i).

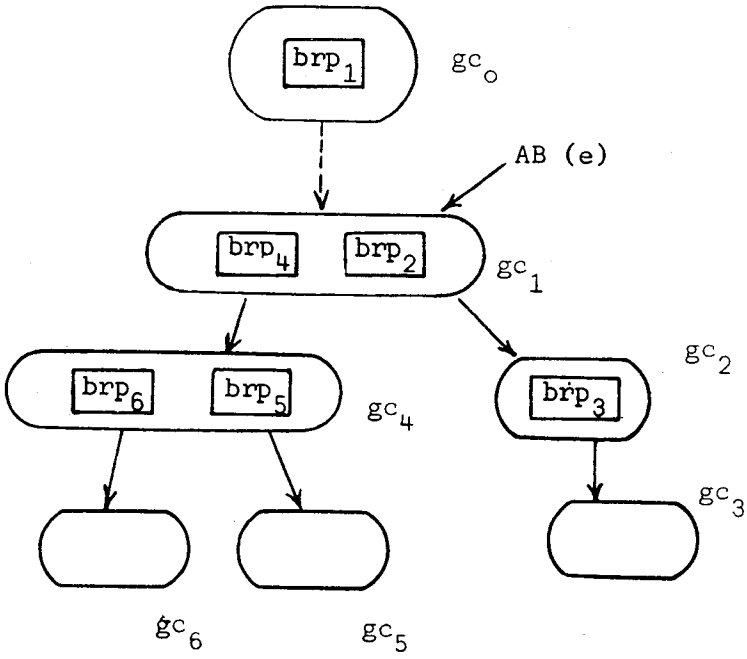


Figure 13.

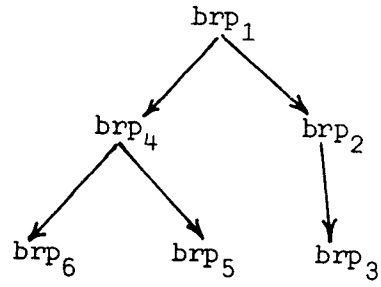


Figure 14. L'arbre ABR (e) associé à l'arbre AB (e) de la figure 13. L'ordre des sommets est un ordre de base inverse.

Notations pour l'algorithme de déplacement :

- gc_1, ... gc_I : les graphes-composants qui correspondent aux sommets de AB (e) ; les indices correspondent à un ordre de base inverse des sommets de AB (e) (voir la figure 13) ;
- COEF (b) ≡ COEF (e, b)
- les autres notations sont celles de 8.1.2.2.2 et de 8.2.2.

Algorithme de déplacement (cas général) :

Initialisation : i : = 1 ; ELIM (brp_i) = faux, $\forall i \in I$.

Pas 1. Vérification de la condition de déplacement en "dehors" de la boucle propre gc_i (voir les conditions C1 et C2g de 8.1.2.2.2 et la transmission des coefficients de 8.2.2).

Si $\neg ML(brp_i) * AL(brp_i) * \neg DE_g(brp_i)$ = vrai alors aller à Pas 2
sinon si ELIM (brp_i) * $\neg DEg(brp_i)$ = vrai alors aller à Pas 3
sinon aller à Pas 4.

Pas 2. Insertion :
si $\neg ELIM(brp_i)$ alors insérer une nouvelle occurrence de e dans le bloc brp_i.
Suppression des occurrences de e déplacées :
si gc_i vérifie la propriété P (cf. 8.1.2.2.2) alors supprimer toutes les occurrences de e dans gc_i
sinon supprimer toute occurrence de e qui se trouve dans un bloc b de gc_i et qui vérifie

$AL(b) * \neg ME(b)$ = vrai
(voir la condition C3 de 8.1.2.2.2).

Détermination du coefficient ELIM :
il s'ensuit de ce qui précède que e est devenue disponible en entrée de de gc_i. Cette disponibilité peut entraîner la disponibilité de e en entrée d'un bloc de représentation principal brp_j qui se trouve dans le graphe gc_i ; ce fait sera exprimé au moyen du coefficient ELIM :

si $\neg ME(brp_j)$ = vrai alors poser

ELIM (brp_j) = vrai, pour tout j tel que brp_i = pred (brp_j).

Aller à Pas 4.

Pas 3. Elimination :
supprimer toute occurrence de e qui se trouve dans un bloc b de gc_i et qui vérifie :

$AL(b) * \neg ME(b)$ = vrai.

Détermination du coefficient ELIM :
si ¬ME (brp_j) = vrai alors poser
ELIM (brp_j) = vrai, pour tout j tel que brp_i = pred (brp_j).

Pas 4. Incrémentation : si i < I alors i : = i + 1 et aller à Pas 1
sinon finir.

Remarque 13. L'insertion -si elle est nécessaire- d'une occurrence nouvelle de e dans un bloc brp_i peut créer de nouvelles possibilités d'élimination dans les blocs de gc_i. L'algorithme ci-dessus n'en tient pas compte : il ne supprime que les occurrences déplaçables (voir la condition C3) de e. Une modification de l'algorithme peut cependant être faite pour supprimer également les occurrences éliminables de e dans gc_i ; cette modification consiste :

- en l'évaluation du système de disponibilité sur gc_i, système que nous avons appelé système global initial de disponibilité (hypothèses "optimistes" à l'entrée c'est-à-dire DS (ef_i) = vrai). Notons par DE'_{gi} les valeurs des coefficients ainsi obtenus, aux blocs de gc_i ;

- en le remplacement, dans l'algorithme, du coefficient ¬ME (brp_j) par DE'_{gi} (brp_j). La condition de suppression du Pas 2 et du Pas 3 devient ainsi :

AL (b) * DE'_{gi} (brp_j)

et le coefficient ELIM (brp_j) est mis à vrai si
DE'_{gi} (brp_j) = vrai.

L'algorithme ainsi modifié sera illustré par l'exemple de la figure 15.
L'expression optimisée est

e ≡ plus a, b.

Une occurrence de e est une instruction
plus a, b → t.

On pourra vérifier -voir la figure 15- que les valeurs des coefficients obtenues en fin du processus de collection d'information (cf. 8.2.2) sont ainsi :

AL (brp_1) = vrai car AE_g (be_1) = vrai,
ML (brp_1) = faux car MS (sp_1) = faux,

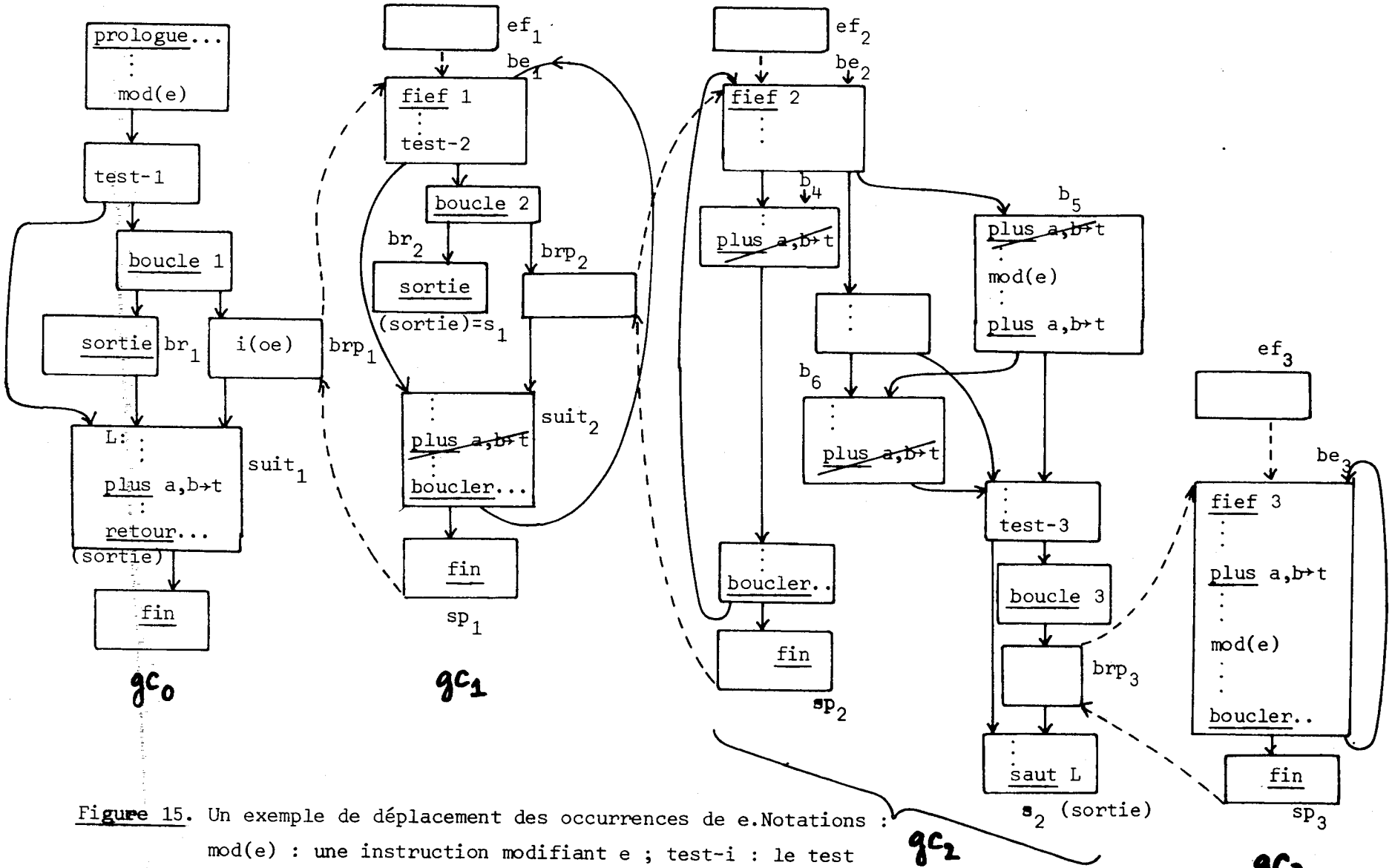


Figure 15. Un exemple de déplacement des occurrences de e.Notations : mod(e) : une instruction modifiant e ; test-i : le test relatif à la boucle i (cf.chapitre 6) ; i(oe) : insertion d'une occurrence nouvelle de e.

DE_g (brp_1) = faux, DE_g ($suit_1$) = faux,

AL (brp_2) = vrai car AE_g (be_2) = vrai, AL (br_2) = vrai,

ML (brp_2) = faux car MS (sp_2) = faux,

DE_g (brp_2) = faux,

AL (brp_3) = vrai, AL (s_2) = vrai

ML (brp_3) = vrai et DE_g (brp_3) = faux

D'autre part :

DE'_{gi} (brp_1) = faux,

DE'_{gi} (brp_2) = vrai car (hypothèse) : DS'_{gi} (ef_1) = vrai

DE'_{gi} (brp_3) = vrai car (hypothèse) : DS'_{gi} (ef_2) = vrai.

L'algorithme modifié procède comme suit :

- i = 1 ; ELIM (brp_i) = faux, i = 1, 2, 3.

Pas 1. AL (brp_1) * ¬ML (brp_1) = vrai => Pas 2 ;

Pas 2. ¬DE'_{gi} (brp_1) ∧ ¬ELIM (brp_1) = vrai => insérer une nouvelle occurrence de e dans brp_1 ;

gc_1 vérifie P => supprimer l'occurrence du bloc $suit_2$;

DE'_{gi} (brp_2) = vrai => ELIM (brp_2) = vrai (notons que ¬ME (brp_2) = vrai également) ;

Pas 4. i = 2 ;

Pas 1. AL (brp_2) * ¬ML (brp_2) = vrai => PAS 2 ;

Pas 2. DE_g (brp_2) + ELIM (brp_2) = vrai (grâce à ELIM !)

pas d'insertion ;

suppression : AL (b_4) * DE'_{gi} (b_4) = vrai,

AL (b_5) * DE'_{gi} (b_5) = vrai,

AL (b_6) * DE'_{gi} (b_6) = vrai

=> supprimer les occurrences de b_4, b_6 et la première occurrence de b_5. Notons que l'occurrence de b_6 ne serait pas supprimée par l'algorithme initial car

$$AL(b_6) * \neg ME(b_6) = \underline{faux} ;$$

$DE'_{gi}(brp_3) = \underline{vrai}$ => $ELIM(brp_3) = \underline{vrai}$ (notons que $\neg ME(brp_3) = \underline{faux}$!)

Pas 4. i = 3 ;

Pas 1. $AL(brp_3) * \neg ML(brp_3) = \underline{faux}$ mais $ELIM(brp_3) = \underline{vrai}$ => Pas 3 ;

Pas 3. $DE'_{gi}(be_3) = \underline{faux}$ => aucune suppression.

Pas 4. finir.

Remarque 14. Si tous les graphes des boucles propres représentés par les sommets de AB (e) vérifient la propriété P, l'algorithme de déplacement est beaucoup plus simple :

Initialisation : i : = 1 ; aucun des blocs brp_i, i ∈ I n'est marqué au départ.

Pas 1. Si $\neg DE_g(brp_i) \wedge (AL(brp_i) \vee \neg ML(brp_i) \vee$ (pred (brp_i) est marqué))
alors aller à Pas 2
sinon aller à Pas 4.

Pas 2. Si pred (brp_i) n'est pas marqué alors insérer une nouvelle occurrence de e dans brp_i.
Supprimer toutes les occurrences de e dans brp_i.
Marquer brp_i.

Pas 4. Si i < I alors i : = i + 1 et aller à Pas 1
sinon finir.

A la fin de l'optimisation de déplacement les instructions qui ont été insérées dans un bloc de représentation principal brp_i doivent être placées avant l'instruction boucle i correspondante. La figure 16 illustre ce déplacement des instructions insérées dans un bloc de représentation principal (voir 7.2.4).

Nous examinerons par la suite la raison et les conséquences de la duplication des tests relatifs aux boucles DO du programme-source (cf. 6.2.7).

La raison et les conséquences de cette duplication sont relatives à l'optimisation de déplacement des instructions.

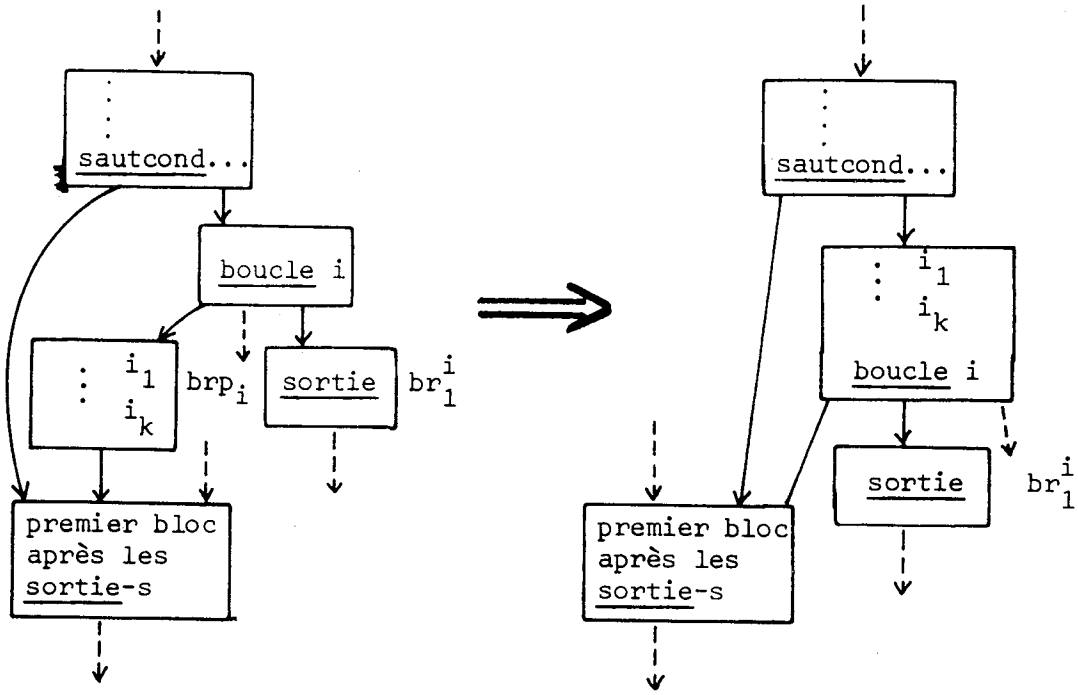


Figure 16. Le déplacement des instructions insérées dans un bloc de représentation principal.

Remarque 15. Une première conséquence de la duplication des tests relatifs aux boucles est la suivante : l'insertion -par l'algorithme de déplacement- d'une occurrence nouvelle d'une expression e dans un bloc de représentation principal brp_i ne peut créer de nouvelles redondances dans le graphe gc_j qui "contient" le bloc brp_i. Examinons, en effet, la figure 17 qui représente le schéma d'un graphe de programme à proximité du bloc qui contient une instruction boucle i (cf. 7.2.4). L'algorithme de déplacement n'effectue l'insertion dans brp_i que dans le cas où e n'est pas disponible en entrée de brp_i ; donc :

$$DE(brp_i) = \underline{faux}$$

ce qui entraîne :

$$DS(test_i) = \underline{faux} \text{ car } ML(predrep_i) = \underline{faux} \text{ (cf. 8.1.2.1).}$$

Comme

$$DE(suite_i) = DS(brp_i) * DS(test_i) * x$$
$$= DS(brp_i) * \underline{faux}$$

il s'ensuit qu'une insertion dans brp_i n'a pas d'effet sur la valeur du coefficient DE ($suite_i$).

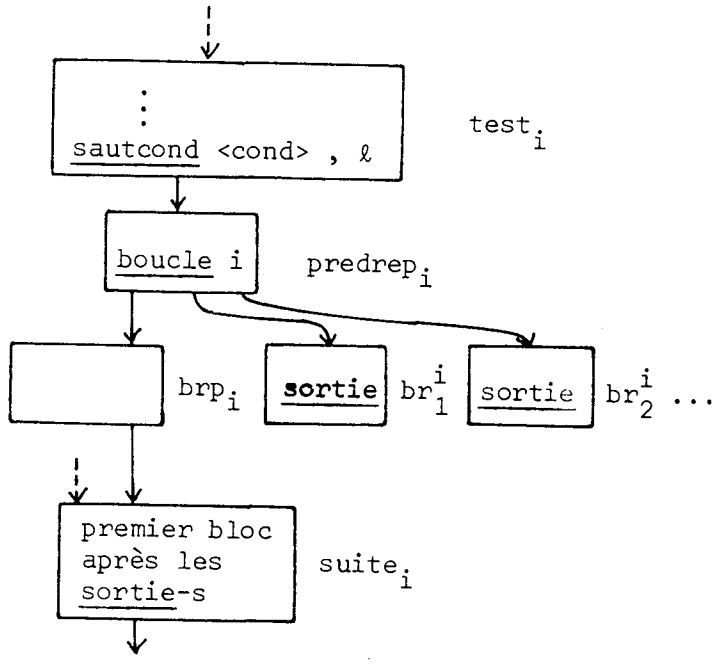


Figure 17.

Remarque 16. Des considérations analogues à celles de la Remarque 15 permettent de conclure que, d'une part :

- la duplication du test d'une boucle est préférable au test au début de la boucle car elle permet d'augmenter les possibilités de déplacement en dehors de la boucle considérée et que, d'autre part

- le test en fin d'une boucle est préférable au test au début d'une boucle, pour les mêmes raisons.

L'algorithme de déplacement décrit dans ce sous-chapitre procède des boucles les plus "externes" vers les boucles "internes" (contrairement à la méthode de Allen [3] où l'on traite d'abord les boucles les plus internes) : cet ordre de traitement permet de réduire -la réduction étant relative à la méthode de déplacement de [3]- le nombre des insertions effectuées.

Nous n'aborderons pas ici le problème de l'ordre dans lequel doivent être effectués les déplacements des différentes expressions optimisables ; ce problème a été traité en [3, 33].

8.2.5 - Caractéristiques de l'approche proposée -

La collection des informations effectuée par l'approche par décomposition décrite en 8.2.2 est une généralisation de la méthode de Allen [5]. Notons que l'on pourrait généraliser d'une façon analogue la méthode par décomposition de Cocke [13].

Nous exposerons par la suite les avantages et les inconvénients de notre approche.

Les avantages :

- les graphes-composants correspondent à des entités déterminables syntaxiquement. Dans les méthodes de Allen [5] et de Cocke [13] les "composants" sont des intervalles (cf. 7.5) qui, eux, ne correspondent pas, dans le cas général, à des entités syntaxiques ; leur détermination est par conséquent plus coûteuse que celle des graphes-composants ;

- l'introduction d'un système de modifiabilité est originale ; on peut estimer que la vérification de la propriété d'invariance permet de réduire (cf. 8.2.3.2) d'une façon notable l'ensemble des expressions sujettes à l'optimisation de déplacement ;

- la prise en compte des portées des expressions (cf. 8.2.3.1) peut se faire d'une façon naturelle et permet de limiter l'ensemble des expressions à prendre en considération lors de la résolution des systèmes sur un graphe-composant donné.

- notre décomposition permet d'effectuer les optimisations locales d'une façon naturelle. Les optimisations locales peuvent être conçues de deux manières :

  - des systèmes locaux (cf. 8.1.2.2) peuvent être résolus sur un graphe-composant gc_i choisi ; les hypothèses à faire en sorties et en entrée de gc_i sont "pessimistes", de même que celles qui concernent les valeurs des coefficients aux blocs de représentation éventuellement "contenus" dans gc_i ;
  - des systèmes locaux peuvent être résolus au cours d'une phase ascendante (cf. 8.2.2) sur tous les graphes-composants d'un graphe de procédure.

Les inconvénients :

- les méthodes par décomposition sont en général plus compliquées (transmissio des coefficients) et plus coûteuses que la méthode directe [26] ;

- si l'on prend en compte les portées et l'invariance des expressions (cf. 8.2.3) alors les ensembles des expressions à traiter au niveau d'un composant diffèrent d'un composant à l'autre.
Il s'ensuit que les vecteurs A, B et Y des systèmes (cf. 8.1.1.1) doivent être de longueurs variables ; la manipulation de ces vecteurs peut poser des problèmes d'implémentation et peut se révéler trop coûteuse ;

- l'optimisation de déplacement est appliquée aux seules "boucles-source" (les boucles DO de SPL/1) ; les boucles éventuellement dues aux sauts ne sont pas identifiées. Il s'ensuit que notre approche est appropriée aux programmes "bien structurés".

Les expressions dont les occurrences apparaissent dans plusieurs graphes des procédures ne sont pas optimisées totalement. Les problèmes d'une optimisation totale de ces expressions proviennent de la récursivité possible des procédures d'un programme-source ; ces problèmes n'ont pas été abordés (voir à ce propos Allen [6]).

9 - CONCLUSION -

Nous avons montré que les possibilités et la facilité d'acquisition des informations nécessaires aux optimisations et l'exactitude de ces informations dépendent de la structure du langage-source et des informations fournies par le programme-source.

L'information de base qu'il faut acquérir est celle de la structure des graphes des programmes. L'exactitude de cette information dépend :

- de la présence -dans le langage-source ou dans le programme-source- des étiquettes variables et des informations disponibles relatives aux valeurs possibles de ces étiquettes variables ;

- de la présence des paramètres ENTRY et des informations disponibles (ou acquises) relatives aux valeurs possibles de ces paramètres (cf. 4.2.1)
Des conjectures (cf. 4.3) ont été formulées quant à l'exactitude du graphe des appels (GPA).

L'exactitude des informations relatives aux modifications des variables dépend de l'exactitude de l'information que représente le graphe GPA (cf.4.3) et l'utilité des informations que représentent les ensembles $MODB_i$ dépend de la structure du programme (voir la condition C de 8.1.2.2.3).

La facilité d'acquisition des informations sur l'invariance dépend de la structure du programme : voir la propriété P et la condition C (8.1.2.2).

De même, la facilité d'acquisition (c'est-à-dire le coût du processus de 8.2.2) des informations relatives à l'anticipabilité et à la disponibilité des expressions dépend de la structure des graphes-composants (voir la Remarque 4 de 8.1.1.2 et la condition C de 8.1.2.2).

Nous avons montré d'autre part que les possibilités, la facilité et l'efficacité (au sens de 8.1.2.2.4) des optimisations traitées dépendent de la structure du langage-source ou de programme-source et des informations fournies par le programme-source ou déduites de ce dernier.

L'optimisation de l'implémentation (ch.5) dépend ainsi de l'exactitude de l'information que représente le graphe GPA.

La facilité de l'optimisation de déplacement (c'est-à-dire le coût de l'algorithme de déplacement de 8.2.4) dépend de la structure des graphes-composants (cf. la propriété P, 8.1.2.2.2). Les possibilités des déplacements dépendent (cf. la remarque 16, 8.2.4) de la position des tests relatifs aux boucles.

Le manque des informations disponibles (cf. 3.1) entraîne la nécessité des contraintes telles que la contrainte de sécurité (cf. 8.1.2.2.4) et celle de cohérence (8.1.2.2.2). En effet, la contrainte de cohérence assure l'efficacité (au sens de 8.1.2.2.4) des déplacements lorsque les informations sur la fréquence des chemins d'un graphe de programme ne sont pas disponibles.

Il s'avère donc que des langages (ou des programmes) "bien structurés" (au sens de [55]) accroissent les possibilités et la facilité des optimisations. Dans un langage "bien structuré" les boucles, en particulier, sont explicites ce qui justifie notre approche par décomposition de 8.2.2. En effet, les graphes des boucles propres sont alors des intervalles (cf. 7.5) et notre approche devient celle de Allen [5] sans que, pour autant, la recherche des intervalles soit nécessaire (voir [39] pour le coût de la détermination des intervalles). Nous pensons, d'ailleurs, qu'il n'est raisonnable d'appliquer des optimisations qu'à des programmes "bien structurés".

Il s'avère en outre que, pour accroître les possibilités des optimisations d'élimination, de déplacement ou de redondances partielles (cf [33]), il est essentiel de connaître les informations sur les possibilités d'interruptions et les informations statistiques (voir ci-dessus les contraintes de sécurité et de cohérence).

Deux orientations générales du développement futur des optimisations semblent envisageables :

- dans une première orientation on tendrait vers une amélioration de la structure des programmes et vers un accroissement des informations fournies par le programmeur, ce qui permettrait d'améliorer, de faciliter et d'étendre les optimisations existantes ;

- une seconde orientation possible serait l'introduction des langages de spécification qui permettraient de transformer les programmes par des méthodes de l'intelligence artificielle (voir [48, 44, 49, 50]).

ANNEXE 1

L'annexe 1 se compose d'une liste (listing) comprenant :

- l'algorithme A1 de 4.2.1 : pages A1/4 à A1/7 ;
- les données à l'entrée pour l'exemple de la figure 1 de 4.2.1 : pages A1/2 à A1/3
- les résultats partiels et définitifs relatifs à l'exemple de la figure 1 de 4.2.1 : pages A1/8 à A1/12.

On pourra vérifier que la matrice MA1 de la page A1/12 (itération 4) est la matrice associée au graphe GPA de la figure 2 de 4.2.1 et que la matrice MA2 de la page A1/12 indique les éléments des ensembles AP_k, k = 1, ... 12 - ensembles relatifs à l'exemple de la figure 1 de 4.2.1.

```
LES DONNEES A L'ENTREE:

 12 612 2 2 2 2 4      II,JJ,KK,MAX1,MAX2,MAX3,MAX4,LL
  0 0                  PROC.1: NP,NE1
  1 1                  PROC.2: NP,NE1
 1                     EP
  1                    PX
  1 1                  PROC.3: NP,NE1
 1                     EP
  5                    PX
  1 1                  PROC.4: NP,NE1
 1                     EP
  4                    PX
  0 0                  PROC.5: NP,NE1
  0 0                  PROC.6: NP,NE1
  2 2                  PROC.7: NP,NE1
 11                    EP
  2 3                  PX
  0 0                  PROC.8: NP,NE1
  0 0                  PROC.9: NP,NE1
  1 0                  PROC.10: NP,NE1
  2 0                  PROC.11:NP,NE1
  1 1                  PROC.12:NP,NE1
 1                     EP
  6                    PX
  2 1 1 1              APNF: PA,CA,NA,NE
 1                     EA
 0                     TA
  7                    EX
  6 3 0 0              APNF: PA,CA,NA,NE
  3 7 1 1              APNF: PA,CA,NA,NE
 1                     EA
 1                     TA
  3                    EX
 12 6 1 1              APNF: PA,CA NA,NE
 1                     EA
 1                     TA
  5                    EX
  1 2 1 3 4 1          NAPF
  1 2 4 5 812          OFAPF
 1 2 2 2               APPEL FORMEL 1:PFA,CA,NA,NE
 11                    EA
 01                    TA
  4 1                  EX
  2 7 1 1              APPEL FORMEL 2:PFA,CA,NA,NE
 1                     EA
 0                     TA
  3                    EX
  2 7 0 0              APPEL FORMEL 3:PFA,CA,NA,NE
  3 7 0 0              APPEL FORMEL 4:PFA,CA,NA,NE
  4 4 1 1              APPEL FORMEL 5:PFA,CA,NA,NE
 1                     EA
 0                     TA
  5                    EX
  4 4 1 1              APPEL FORMEL 6: PFA,CA,NA,NE
 1                     EA
 0                     TA
 10                    EX
```

```
 4 4 1 1          APPEL FORMEL 7: PFA,CA,NA,NE
1                 EA
0                 TA
11                EX
 5 3 0 0          APPEL FORMEL 8: PFA,CA,NA,NE
 5 6 2 2          APPEL FORMEL 9: PFA,CA,NA,NE
11                EA
00                TA
 8 9              EX
 5 6 1 0          APPEL FORMEL 10: PFA,CA,NA,NE
 5 6 2 1          APPEL FORMEL 11: PFA,CA,NA,NE
10                EA
0                 TA
 8                EX
 6 2 2 0          APPEL FORMEL 12:PFA,CA,NA,NE
```

Note:
^ : ¬
! : ou
& : et

```
BEGIN" "INTEGER" II,JJ,KK,LL,MAX1,MAX2,MAX3,MAX4;
 "COMMENT"    II: LE NOMBRE DE  PROCEDURES
              JJ: LE NOMBRE DE  PARAMETRES ENTRY
              KK: LE NOMBRE  D'APPELS FORMELS
              LL: LE NOMBRE D'APPELS NON-FORMELS
              MAX1: LE NOMBRE MAXIMAL D'ARGUMENTS D'UNE LISTE  DES ARGUMEN
              MAX2: LE NOMBRE MAXIMAL DE PARAMETRES D'UNE LISTE DES PARAME
              MAX3: LE NOMBRE MAXIMAL D'ARGUMENTS ENTRY D'UNE LISTE
                    DES ARGUMENTS
              MAX4: LE NOMBRE MAXIMAL DE PARAMETRES ENTRY D'UNE LISTE
                    DES PARAMETRES
              MP2B: MATRICE INDIQUANT LES ELEMENTS DE MP2 DEJA TRAITES
              MA1: MATRICE DES APPELS DES PROCEDURES
              MA2: LA LIGNE K: LES PROCEDURES APPELEES PAR L'APPEL FORMEL
              MP1: MATRICE DES PASSAGES ARGUMENT ENTRY-ARGUMENT ENTRY
              MP2: MATRICE DES PASSAGES PARAMETRE ENTRY- ARGUMENT ENTRY
              EA: PRESENCE DES ARGUMENTS ENTRY DANS LES LISTES D'APPELS
              TA: SI "TRUE":ARGUMENT=ARGUMENT ENTRY
                  SINON:ARGUMENT=PARAMETRE ENTRY
              EX:  LES INDICES DES ARGUMENTS DE TYPE ENTRY
              EP: PRESENCE DES PARAMETRES ENTRY DANS LES LISTES DES PARAME
              PX: LES INDICES DES PARAMETRES ENTRY
              PFA: PARAMETRES FORMELS APPELES DANS LES APPELS FORMELS
              CA: LE CORPS DE PROCEDURE CONTENANT  L'APPEL FORMEL
              NA: LE NOMBRE D'ARGUMENTS D'UN APPEL FORMEL
              NE: LE NOMBRE D'ARGUMENTS   DE TYPE ENTRY D'UN APPEL FORMEL
              NP: LES NOMBRES DES PARAMETRES DES PROCEDURES
              NE1: LES NOMBRES DES PARAMETRES ENTRY DES PROCEDURES
              NAPF:NAPF[J],1<=J<=JJ,EST LE NOMBRE D'APPELS FORMELS AU
                   PARAMETRE ENTRY D'INDICE J.LES KK APPELS FORMELS SONT
                   ORDONNES DANS L'ORDRE CROISSANT DES INDICES J DES
                   PARAMETRES ENTRY DESIGNANT LES PROCEDURES FORMELLEMENT
                   APPELEES.
              OFAPF:OFAPF[J] PERMET DE TROUVER LE NUMERO K,1<=K<=KK, DU PA
                   APPEL FORMEL AU PARAMETRE ENTRY D'INDICE J;
ECT:INPUT(60,"(8(ZD),/)",II,JJ,KK,MAX1,MAX2,MAX3,MAX4,LL);
 "BEGIN" "BOOLEAN" "ARRAY" MA1[1:II,1:II],MP1[1:JJ,1:JJ], MA2[1:KK,1:II]
 MP2[1:II,1:JJ],MP2B[1:II,1:JJ],EA[1:KK,1:MAX1],TA[1:KK,1:MAX3],
 EP[1:II,1:MAX2];
 "INTEGER""ARRAY" NAPF[1:JJ],PFA[1:KK],CA[1:KK],NA[1:KK],NE[1:KK],OFAPF[
 NP[1:II],NE1[1:II],EX[1:KK,1:MAX3],PX[1:II,1:MAX4];
 "INTEGER" I,J,K,ITERCPT;
 "BOOLEAN" TMP1,TMP2,TEST,TMA1,TMA2;
"PROCEDURE" WARSHALL; "COMMENT"
 **********************;
                        "BEGIN" "BOOLEAN" "ARRAY" B[1:JJ];
    "FOR" I:=1 "STEP" 1 "UNTIL" JJ "DO"
    "BEGIN""FOR"J:=1"STEP"1"UNTIL"JJ"DO" B[J]:=MP1[I,J];
      "FOR"J:=1"STEP"1"UNTIL"JJ"DO""IF"MP1[J,I]"THEN"
      "FOR" K:=1 "STEP" 1 "UNTIL" JJ "DO"
         MP1[J,K]:=MP1[J,K]!B[K]
    "END"
                        "END" WARSHALL;
"PROCEDURE" MP1MP2; "COMMENT"
 **********************;
    "COMMENT" CALCUL DE MP2 EN FONCTION DE MP1;
                        "BEGIN" "FOR" I:=1 "STEP" 1 "UNTIL" JJ "DO"
                        "FOR" J:=1"STEP"1"UNTIL" JJ "DO"
```

```
                    "FOR" K:=1"STEP"1"UNTIL" II "DO"
    ┌"BEGIN" TMP2:=TMP2!^(MP2[K,I]"EQUIV"MP2[K,J]);
    │           MP2[K,I]:=MP2[K,I]!MP2[K,J]
    └"END"
                    "END" MP1MP2;
'PROCEDURE" SET; "COMMENT"
***************;
'BEGIN" "COMMENT" SI L'APPEL AU PARAMETRE K PEUT CONVENIR:MA1 ET MA2 CHANGEN
                ET MP1 ET/OU MP2 PEUVENT CHANGER;
    "INTEGER" L;
    "IF" NA[K]^=NP[I]!NE[K]^=NE1[I] "THEN" "GOTO" EXIT1;
    "IF" NE[K]=0 "THEN"
    "FOR" L:=1 "STEP" 1 "UNTIL" NA[K] "DO"
    "IF" ^(EA[K,L] "EQUIV" EP[I,L]) "THEN" "GOTO" EXIT1;
    TMA1:=TMA1 !^ MA1[CA[K],I];
    MA1[CA[K],I]:="TRUE";
    "IF" TEST "THEN"
    "BEGIN" TMA2:=TMA2 !^ MA2[K,I];
            MA2[K,I]:="TRUE"
    "END";
    "IF" NE[K]=0 "THEN" "GOTO" EXIT1;
    "FOR" L:=1 "STEP" 1 "UNTIL" NE[K] "DO"
    ┌"BEGIN" "INTEGER" IND,IND1;
    │ IND:=PX[I,L];IND1:=EX[K,L];
    │ "IF" TA[K,L] "THEN" "BEGIN" TMP1:=TMP1 !^ MP1[IND1,IND];
    │                             MP1[IND1,IND]:="TRUE"
    │                     "END"
    │                "ELSE" "BEGIN" TMP2:=TMP2 !^ MP2[IND1,IND];
    │                             MP2[IND1,IND]:="TRUE"
    │                     "END"
    └"END" BOUCLE L;EXIT1:
'END" SET;
'PROCEDURE" LIRE; "COMMENT"
***************;
'BEGIN"
NPUT(60,"("4(ZD),/")",PFA[K],CA[K],NA[K],NE[K]);
'IF" NE[K]^=0 "THEN"
'BEGIN"
'FOR" I:=1"STEP" 1 "UNTIL" NA[K] "DO" INPUT(60,"("P")",EA[K,I]);
NPUT(60,"("/")");
'FOR" I:=1"STEP" 1 "UNTIL" NE[K] "DO" INPUT(60,"("P")",TA[K,I]);
NPUT(60,"("/")");
'FOR" I:=1"STEP" 1 "UNTIL" NE[K] "DO" INPUT(60,"("ZD")",EX[K,I]);
NPUT(60,"("/")")
'ND"
'END" LIRE;
PROCEDURE" SORTIR; "COMMENT"
*****************;
                    "BEGIN"
UTPUT(61,"("X"("ITERATION NO=")",ZD/")",ITERCPT);
UTPUT(61,"("/"("X"("**************")"2/")");
UTPUT(61,"("/"("MA1=")"2/")");
FOR" I:=1 "STEP" 1 "UNTIL" II "DO" "BEGIN" "FOR" J:=1 "STEP" 1 "UNTIL"II"DO
                                    OUTPUT(61,"("P")",MA1[I,J]);
                                    OUTPUT(61,"("/")")
                                    "END";
UTPUT(61,"("2/,"("MP1=")"2/")");
FOR" I:=1 "STEP" 1 "UNTIL" JJ "DO" "BEGIN" "FOR" J:=1 "STEP" 1 "UNTIL"JJ"DO
                                    OUTPUT(61,"("P")",MP1[I,J]);
```

```
                                        OUTPUT(61,"("/")")
                                        "END";

OUTPUT(61,"("2/"("MP2=")"2/")");
"FOR" I:=1 "STEP" 1 "UNTIL" II"DO" "BEGIN" "FOR" J:=1 "STEP" 1 "UNTIL"JJ
                                        OUTPUT(61,"("P")",MP2[I,J]);
                                        OUTPUT(61,"("/")")
                                        "END"  ;

OUTPUT(61,"("2/"("MA2=")"2/")");
"FOR" I:=1 "STEP" 1 "UNTIL" KK"DO" "BEGIN" "FOR" J:=1 "STEP" 1 "UNTIL"II
                                        OUTPUT(61,"("P")",MA2[I,J]);
                                        OUTPUT(61,"("/")")
                                        "END"

"END" SORTIR;
"COMMENT" INITIALISER A FAUX MA1,MA2,MP1,MP2,MP2B
          ***********  ;
"FOR" I:=1 "STEP" 1 "UNTIL" II "DO" "FOR" J:=1 "STEP" 1 "UNTIL" II "DO"
MA1[I,J]:="FALSE";
"FOR" I:=1 "STEP" 1 "UNTIL" JJ "DO" "FOR" J:=1 "STEP" 1 "UNTIL" JJ "DO"
MP1[I,J]:="FALSE";
"FOR" I:=1 "STEP" 1 "UNTIL" II "DO" "FOR" J:=1 "STEP" 1 "UNTIL" JJ "DO"
"BEGIN" MP2[I,J]:="FALSE";MP2B[I,J]:="FALSE" "END";
"FOR" I:=1 "STEP" 1 "UNTIL" KK "DO" "FOR" J:=1 "STEP" 1 "UNTIL" II "DO"
MA2[I,J]:="FALSE";
"COMMENT" LIRE LES DONNEES DES PROCEDURES
          ********************************  ;
"FOR" I:=1"STEP"1"UNTIL" II "DO"
"BEGIN"
INPUT(60,"("2(ZD),/")",NP[I],NE1[I]);
"IF" NE1[I]^=0 "THEN"
"BEGIN"
"FOR" J:=1 "STEP" 1 "UNTIL" NP[I] "DO" INPUT(60,"("P")",EP[I,J]);
INPUT(60,"("/")");
"FOR" J:=1 "STEP" 1 "UNTIL" NE1[I] "DO" INPUT(60,"("ZD")",PX[I,J]);
INPUT(60,"("/")")
"END"
"END";
"COMMENT" LIRE DONNEES/APPELS NON-FORMELS ET, POUR TOUT APPEL, INITIAL
          ********************************  AUSSITOT MA1,MP1 ET MP2;
K:=1;TMP1:="FALSE";TMP2:="FALSE";TEST:="FALSE";
"FOR" J:=1 "STEP" 1 "UNTIL" LL "DO"
"BEGIN"
        LIRE; "COMMENT" POUR UN SEUL APPEL NON-FORMEL;
        I:=PFA[1];
        SET
"END" NOTE: UN ELEMENT DE MA1 EST MIS A VRAI DES Q'UN APPEL NON-FORMEL
CONVIENT;
TEST:="TRUE";
"COMMENT"  LIRE LES DONNEES/APPELS FORMELS
           ********************************* ;
"FOR" I:=1"STEP"1"UNTIL" JJ"DO"
INPUT(60,"("ZD")",NAPF[I]);INPUT(60,"("/")");
"FOR" I:=1"STEP"1"UNTIL" JJ"DO"
INPUT(60,"("ZD")",OFAPF[I]);INPUT(60,"("/")");
"FOR" K:=1"STEP"1"UNTIL" KK"DO"
LIRE;
"COMMENT" LA FIN DES LECTURES
          ******************* ;
"IF" TMP1 "THEN" "BEGIN" MARSHALL; MP1MP2 "END";
ITERCPT:=0;
```

```
ORTIR  ;
F" ^TMP2 "THEN" "GOTO" FINI;
TER:TMP2:="FALSE";
    TMP1:="FALSE";
    TMA1:="FALSE";
    TMA2:="FALSE";
    "FOR" I:=1"STEP"1"UNTIL" II"DO"
    "FOR" J:=1"STEP"1"UNTIL" JJ"DO"
BEGIN""COMMENT" BOUCLES I,J;
    "IF" MP2[I,J] "THEN"
"BEGIN"
    "IF" MP2B[I,J] "THEN" "GOTO" EXIT;
    MP2B[I,J]:="TRUE";
    "FOR" K:=OFAPF[J] "STEP" 1 "UNTIL" OFAPF[J]+NAPF[J]-1 "DO"
    SET;
"END";EXIT:
END" BOUCLES J,I;
TERCPT:=ITERCPT+1;
IF" TMP1 "THEN" WARSHALL;
IF" TMP1!TMP2 "THEN" MP1MP2;
IF" TMP2 "THEN" "BEGIN" SORTIR;"GOTO" ITER "END"
        "ELSE" "IF" ^TMA2 & ^TMA1 "THEN"
          "BEGIN"
          OUTPUT(61,"(""("ITERATION NB=")",ZD"(" NO CHANGE")"/")",ITERCPT);
          "GOTO" FINI
          "END";
ORTIR;
I:
  OUTPUT(61,"(""("****************FIN D'ITERATIONS")""")")
FND"
D" LA FIN DU PROGRAMME;
```

```
ITERATION NO= 0
***************

MA1=

010000000000
000000000000
000001000000
000000000000
000000000000
000000000001
001000000000
000000000000
000000000000
000000000000
000000000000
000000000000

MP1=

000000
000000
000011
000000
000001
000000

MP2=

000000
000000
000000
000000
000000
000000
100000
000000
000000
000000
000000
000000

MA2=

000000000000
000000000000
000000000000
000000000000
000000000000
000000000000
000000000000
000000000000
000000000000
000000000000
000000000000
000000000000
```

A119

ITERATION NO= 1
****

MA1=

010000000000
000000100000
000001000000
000000000000
000000000000
000000000001
001000000000
000000000000
000000000000
000000000000
000000000000
000000000000

MP1=

001011
000000
000011
000000
000001
000000

MP2=

000000
000000
000000
010000
000000
000000
101011
000000
000000
000000
000000
000000

MA2=

000000100000
000000000000
000000000000
000000000000
000000000000
000000300000
000000000001
000000000000
000000000000
000000000000
000000000000

AA/10

ITERATION NO= 2
****

MA1=

010000000000
000001000000
000001000000
000000000000
000000000000
000001000001
001100011000
000000000000
000000000000
000000000000
000000000000
000000000000

MP1=

001011
000000
000011
000000
000001
000000

MP2=

000000
000000
000100
010000
000000
000000
101011
010000
001011
000000
000000
000000

MA2=

000000100000
000100000000
000000010000
000000001000
000000000000
000000000000
000000000000
000000000000
000000100000
000000000000
000000000000
000000000000

ITERATION NO= 3
****

MA1=

010000000000
000001000000
000011001000
001000000000
000000000000
000000100101
001100011000
000000000000
000000000000
000000000000
000000000000
000000000000

MP1=

001011
000000
000011
000000
000001
000000

MP2=

000000
000000
000100
010000
000011
000000
101011
010000
001011
000011
000011
000000

MA2=

000000100000
000100000000
000000010000
000000001000
001000000000
001000000000
001000000000
000010001000
000000100000
000000000100
000000000000
000000000000

```
 ITERATION NO= 4
 ***************

 MA1=

 010000000000
 000000100000
 000011001000
 001000000000
 000000000000
 000000100101
 001100011000
 000000000000
 000000000000
 000000000000
 000000000000
 000000000010

 MP1=

 001011
 000000
 000011
 000000
 000001
 000000

 MP2=

 000000
 000000
 000100
 010000
 000011
 000000
 101011
 010000
 001011
 000011
 000011
 000000

 MA2=

 000000100000
 000100000000
 000000010000
 000000001000
 001000000000
 001000000000
 001000000000
 000010001000
 000000100000
 000000000100
 000000000000
 000000000010
 ***************FIN D'ITERATIONS
```

ANNEXE 2

Lexique des termes introduits

Notations

BIBLIOGRAPHIE

B.1 - Optimisation -

[1] - AHO, A.V. - ULLMAN, J.D.
Transformations on straight line programs-preliminary version 2nd annual ACM symposium on the theory of computing.
May 3-6, 1970, Northampton, Mass.

[2] - AHO, A.V. - ULLMAN, J.D.
Equivalence of programs with structured variables
Journal of Computer and System Sciences, 6 : 2.

[3] - ALLEN, F.E.
Program optimization
Annual Review in automatic programming, vol. 5. Pergamon P., N.Y. 1969.

[4] - ALLEN, F.E.
Control flow analysis
SIGPLAN notices, july 1970.

[5] - ALLEN, F.E.
A basis for program optimization
IFIP 1971

[6] - ALLEN, F.E.
Interprocedural data flow analysis
IFIP 1974

[7] - ALLEN, F.E. - COCKE, J.
A catalogue of optimizing transformations
In Design and optimization of compilers. Courant computer science symp. 5, march 29-30, 1971. Prentice Hall 1972.

[8] - BEATTY, J.C.
An axiomatic approach to code optimization for expressions
J.A.C.M. vol. 19, N° 4, oct. 1972

[9] - BEATTY, J.C.
Register assignment algorithm for generation of highly optimized object code.
IBM J. Res. Develop. Jan. 1974.

[10] - BREUER, M.A.
Generation of optimal code for expressions via factorization
C.A.C.M. Vol. 12, n° 6, june, 1969.

[11] - BUSAM, V.A. - ENGLUND, D.E.
Optimization of expressions in Fortran
C.A.C.M. Vol 12, n° 12, dec., 1969.

[12] - COCKE, J.
Global common subexpression elimination
SIGPLAN notices, july, 1970

[13] - COCKE, J. - SCHWARTZ, J.T.
Programming languages and their compilers (preliminary notes)
Courant institute of mathematical sciences, N.Y. U., N.Y., april 197(

[14] - COCKE, J.
On certain graph-theoretic properties of programs.
RC 3391. June 9, 1971. Programming and programming languages

[15] - DAY, W.H.
Compiler assignment of data items to registers.
IBM Sys. J. Vol 9, n° 4, 1970

[16] - EARNEST, C.
Some topics in code optimization
J.A.C.M., vol 21, n° 1, jan. 1974

[17] - EARNEST, C.P. - BALKE, K.G. - ANDERSON, J.
Analysis of graphs by ordering of nodes
J.A.C.M., vol 19, n° 1, jan. 1972

[18] - ELSON, M. - RAKE, S.T.
Code-generation technique for large-language compilers
IBM Sys. J.3., 1970

[19] - FINKELSTEIN, M.
A compiler optimization technique
Comp. J. 1968

[20] - FRAILEY, D.J.
Expression optimization using unary complement operators
Proc. of a Symp. on Comp. Optimization, SIGPLAN Notices, july 1970

[21] - FREIBURGHOUSE, R.A.
Register allocation via usage counts
C.A.C.M. vol 17, n° 11, nov. 1974

[22] - GEAR, C.W.
High speed compilation of efficient object code
C.A.C.M., vol 8, n° 8, aug., 1965

[23] - GESCHKE, C.M.
Global program optimizations
Dept. of Comp. Science - Carnegie-Mellon, U. oct, 1972

[24] - GOLDBERG, P.C.
A comparison of certain optimization techniques
Courant Comp. Science Symp. 5. In Design and optimization of
Compilers. Prentice-Hall, 1972.

[25] - HECHT, M.S.
Topological sorting and flow graphs.
IFIP 1974

[26] - HECHT, M.S.
Analysis of a simple algorithm for global data flow problems
Conf. record of ACM symp. on principles of progr. languages,
Boston, oct. 1-3, 1973

[27] - HOPKINS, M.E.
An optimizing compiler design.
IFIP 1971

[28] - JOHNSSON, R.K.
A survey of register allocation.
Comp. Science Dept. Carnegie-Mellon, U., may, 1973

[29] - KENNEDY, K.
A global flow analysis algorithm. Intern. J
Computer Math. Section A, vol. 3, 1971

[29 bis] - KENNEDY, K.
Index register allocation in straight line code and simple loops
Courant Comp. Sci. Symp. 5, march 29-30, 1971
In Design and optimization of compilers, Prentice-Hall, 1972

[30] - KILDALL, G.A.
A unified approach to global program optimization
Conf. record of ACM symp. on principles of progr. languages,
Boston, Mass., oct., 1973

[31] - LOUIT, G.
Optimisation des expressions arithmétiques
Extension des algorithmes de R. SETHI et J.D. ULLMAN
Les techniques de l'informatique - Congrès AFCET 1972, Grenoble

[32] - LOWRY, E.D. - MEDLOCK, C.W.
Object code optimization
C.A.C.M., vol. 12, n° 1, jan., 1969

[33] - MOREL, E. - RENVOISE, C.
Etude et réalisation d'un optimiseur global
Thèse. L'Université de Paris VI, le 21 juin 1974

[34] - SCHAEFFER, M.A.
Mathematical theory of global program optimization
Prentice Hall 1973

[35] - SETHI, R. - ULLMAN, J.D.
The generation of optimal code for arithmetic expressions
J.A.C.M., vol 17, n° 4, oct., 1970

[36] - SPILLMAN, T.C.
Exposing side-effects in a PL/1 optimising compiler
IFIP 1971

[37] - VEILLON, G.
Transformation de programmes récursifs
Laboratoire d'Informatique, Université de Grenoble - A paraître dans l'AFCET - Revue bleue.

[38] - ULLMAN, J.D.
Fast algorithms for the elimination of common subexpressions
Acta informatica 2, 1973

B.2 : Ouvrages généraux sur la compilation -

[39] - AHO, A.V. - ULLMAN, J.D.
The theory of parsing, translation and compiling
Vol II : Compiling. Prentice-Hall 1972

[40] - GRIES, D.
Compiler construction for digital computers
Wiley, N.Y., 1971

B.3 : Intelligence artificielle - Schémas - Récursivité -

[41] - DE BAKKER, J.W. - DE ROEVER, W.P.
A calculus for recursive program schemes
Automata, languages and programming, IRIA symp. july 3-7, 1972

[42] - HITCHCOCK, P. - PARK, D.
Introduction rules and termination proofs
IRIA symp., july 3-7, 1972

[43] - MANNA, Z.
Mathematical theory of computation
Mc Graw Hill Book C^{ie}. New-York, Londres, Paris 1974

[44] - MANNA, Z. - WALDINGER, R.J.
Toward automatic program synthesis
C.A.C.M. Vol 14, n° 3, march 1971

[45] - MANNA, Z. - VUILLEMIN, J.
Fixpoint approach to the theory of computation
IRIA symp., july 3-7, 1972

[46] - PARK, D.
Fixpoint induction and proofs of program properties
Machine Intelligence 5. Edinburgh U. Press, 1970

[47] - PATERSON, M.S.
Program schemata
Mach. Intelligence 3, Edinburgh U. Press., 1968

[48] - SCHWARTZ, J.T.
Principles of specification language design with some observations concerning the utility of specification languages
Courant Comp. Symp. 4, march 1-2, 1971.
In Algorithm specification, Prentice-Hall

[49] - STRONG, H.R.
Flowchartable recursive specifications
Courant Comp. Symp. 4, march 1-2, 1971.
In Algorithm specification, Prentice-Hall

[50] - STRONG, H.R.
Translating recursion equations into flow charts
2nd annual ACM symp. on the theory of computing,
May 4-6, Northampton, Mass., 1970

B.4 : Divers -

[51] - BERGE, C.
Graphes et hypergraphes
Dunod. 1970

[52] - DAVIS, M.
Computability and unsolvability
Mc Graw-Hill, N.Y. 1958

[53] - GROUPE ALGOL DE L'AFCET - Edité par J. BUFFET, P. ARNAL, A. QUERE
Définition du langage algorithmique Algol 68
Herrmann, Paris 1972

[54] - DERNIAME, J.C. - PAIR, C.
Problèmes de cheminement dans les graphes
Dunod 1971

[55] - DAHL, O.I. - DIJKSTRA, E.W.
Structured programming
A.P.I.C. Studies in Data Processing n° 8
Academic Press, Londres, New-York, 1972

[56] - GRIFFITHS, M.
Relationship between definition and implementation of a language
Paru dans Software Engineering, édité par F.L. BAUER dans la série
Lecture Notes in Computer Science n° 30
Springer-Verlag 1975

[57] - KNUTH, D.E.
The art of computer programming
Vol I. Addison-Wesley, 1968

[58] - IBM
PL/1 reference manual
IBM System 360,
GC 28-8201-3

[59] - QUEYSANNE, M.
Algèbre - Librairie Armand Colin - 1964

[60] - IBM
PL/1 (F) Compiler - Program logic manual
IBM system 360 Operating System. Program number 360S - NL - 511, GY 28 - 6800 - 5

[61] - NAUR, P.
Report on the algorithmic language - Algol 60
Regnencentralen, Copenhague 1960

[62] - WULF, W.A. - RUSSEL, D.B. - HABERMANN, A.N.
Bliss : a language for systems programming.
C.A.C.M., dec. 1971, vol 14, n° 12

[63] - WILLIS, B.
Définition et implantation de la sémantique des langages de programmation
Thèse présentée à l'Université Scientifique et Médicale de Grenoble
le 22.3.1974