Vers une programmation systématique : étude de quelques méthodes, techniques et outils

Pierre-Claude Scholl

THESE

présentée à

Université Scientifique et Médicale de Grenoble

Institut National Polytechnique de Grenoble

pour obtenir le grade de
DOCTEUR D'ETAT ES- SCIENCES
Mathématiques

par

Pierre - Claude SCHOLL

VERS UNE PROGRAMMATION SYSTEMATIQUE :

ETUDE DE QUELQUES METHODES , TECHNIQUES ET OUTILS.

Thèse soutenue le 29 Juin 1979 devant la commission d'examen.

L. BOLLIET	Président
M. GALINIER	
M. GRIFFITHS	
P. JORRAND	
M. LUCAS	Examinateurs
M. SAKAROVITCH	
M. SINTZOFF	
G. VEILLON	

THESE

présentée à

Université Scientifique et Médicale de Grenoble

Institut National Polytechnique de Grenoble

pour obtenir le grade de
DOCTEUR D'ETAT ES- SCIENCES
Mathématiques

par

Pierre - Claude SCHOLL

VERS UNE PROGRAMMATION SYSTEMATIQUE :

ETUDE DE QUELQUES METHODES , TECHNIQUES ET OUTILS.

Thèse soutenue le 29 Juin 1979 devant la commission d'examen.

L. BOLLIET	Président
M. GALINIER	
M. GRIFFITHS	
P. JORRAND	
M. LUCAS	Examinateurs
M. SAKAROVITCH	
M. SINTZOFF	
G. VEILLON	

UNIVERSITE SCIENTIFIQUE ET MEDICALE DE GRENOBLE

Monsieur Gabriel CAU : Président
Monsieur Joseph KLEIN : Vice-Président

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

PROFESSEURS TITULAIRES

MM.	AMBLARD Pierre	Clinique de dermatologie
	ARNAUD Paul	Chimie
	ARVIEU Robert	I.S.N.
	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 A
	BEAUDOING André	Clinique de pédiatrie et puériculture
	BELORIZKY Elie	Physique
	BARNARD Alain	Mathématiques pures
Mme	BERTRANDIAS Françoise	Mathématiques pures
MM.	BERTRANDIAS Jean-Paul	Mathématiques pures
	BEZES Henri	Clinique chirurgicale et traumatologie
	BLAMBERT Maurice	Mathématiques pures
	BOLLIET Louis	Informatique (I.U.T. B)
	BONNET Jean-Louis	Clinique ophtalmologie
	BONNET-EYMARD Joseph	Clinique hépato-gastro-entérologie
Mme	BONNIER Marie-Jeanne	Chimie générale
MM.	BOUCHERLE André	Chimie et toxicologie
	BOUCHEZ Robert	Physique nucléaire
	BOUSSARD Jean-Claude	Mathématiques appliquées
	BOUTET DE MONVEL Louis	Mathématiques pures
	BRAVARD Yves	Géographie
	CABANEL Guy	Clinique rhumatologique et hydrologique
	CALAS François	Anatomie
	CARLIER Georges	Biologie végétale
	CARRAZ Gilbert	Biologie animale et pharmacodynamie

.../...

MM.	CAU Gabriel	Médecine légale et toxicologie
	CAUQUIS Georges	Chimie organique
	CHABAUTY Claude	Mathématiques pures
	CHARACHON Robert	Clinique ot-rhino-laryngologique
	CHATEAU Robert	Clinique de neurologie
	CHIBON Pierre	Biologie animale
	COEUR André	Pharmacie chimique et chimie analytique
	COUDERC Pierre	Anatomie pathologique
	DEBELMAS Jacques	Géologie générale
	DEGRANGE Charles	Zoologie
	DELORMAS Pierre	Pneumophtisiologie
	DEPORTES Charles	Chimie minérale
	DESRE Pierre	Métallurgie
	DODU Jacques	Mécanique appliquée (I.U.T. I)
	DOLIQUE Jean-Michel	Physique des plasmas
	DREYFUS Bernard	Thermodynamique
	DUCROS Pierre	Cristallographie
	FONTAINE Jean-Marc	Mathématiques pures
	GAGNAIRE Didier	Chimie physique
	GALVANI Octave	Mathématiques pures
	GASTINEL Noël	Analyse numérique
	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
	LACAZE Albert	Thermodynamique
	LACHARME Jean	Biologie végétale
Mme	LAJZEROWICZ Janine	Physique
MM.	LAJZEROWICZ Joseph	Physique
	LATREILLE René	Chirurgie générale
	LATURAZE Jean	Biochimie pharmaceutique
	LAURENT Pierre	Mathématiques appliquées
	LEDRU Jean	Clinique médicale B
	LE ROY Philippe	Mécanique (I.U.T. I)

.../...

MM.	LLIBOUTRY Louis	Géophysique
	LOISEAUX Jean-Marie	Sciences nucléaires
	LONGEQUEUE Jean-Pierre	Physique nucléaire
	LOUP Jean	Géographie
Mlle	LUTZ Elisabeth	Mathématiques pures
MM.	MALINAS Yves	Clinique obstétricale
	MARTIN-NOEL Pierre	Clinique cardiologique
	MAYNARD Roger	Physique du solide
	MAZARE Yves	Clinique Médicale A
	MICHEL Robert	Minéralogie et pétrographie
	MICOUD Max	Clinique maladies infectieuses
	MOURIQUAND Claude	Histologie
	MOUSSA André	Chimie nucléaire
	NEGRE Robert	Mécanique
	NOZIERES Philippe	Spectrométrie physique
	OZENDA Paul	Botanique
	PAYAN Jean-Jacques	Mathématiques pures
	PEBAY-PEYROULA Jean-Claude	Physique
	PERRET Jean	Séméiologie médicale (neurologie)
	RASSAT André	Chimie systématique
	RENARD Michel	Thermodynamique
	REVOL Michel	Urologie
	RINALDI Renaud	Physique
	DE ROUGEMONT Jacques	Neuro-Chirurgie
	SARRAZIN Roger	Clinique chirurgicale B
	SEIGNEURIN Raymond	Microbiologie et hygiène
	SENGEL Philippe	Zoologie
	SIBILLE Robert	Construction mécanique (I.U.T. I)
	SOUTIF Michel	Physique générale
	TANCHE Maurice	Physiologie
	VAILLANT François	Zoologie
	VALENTIN Jacques	Physique nucléaire
Mme	VERAIN Alice	Pharmacie galénique
MM.	VERAIN André	Physique biophysique
	VEYRET Paul	Géographie
	VIGNAIS Pierre	Biochimie médicale

.../...

PROFESSEURS ASSOCIES

MM.	CRABBE Pierre	CERMO
	SUNIER Jules	Physique

PROFESSEURS SANS CHAIRE

Mlle	AGNIUS-DELORS Claudine	Physique pharmaceutique
	ALARY Josette	Chimie analytique
MM.	AMBROISE-THOMAS Pierre	Parasitologie
	ARMAND Gilbert	Géographie
	BENZAKEN Claude	Mathématiques appliquées
	BIAREZ Jean-Pierre	Mécanique
	BILLET Jean	Géographie
	BOUCHET Yves	Anatomie
	BRUGEL Lucien	Energétique (I.U.T. I)
	BUISSON René	Physique (I.U.T. I)
	BUTEL Jean	Orthopédie
	COHEN-ADDAD Jean-Pierre	Spectrométrie physique
	COLOMB Maurice	Biochimie médicale
	CONTE René	Physique (I.U.T. I)
	DELOBEL Claude	M.I.A.G.
	DEPASSEL Roger	Mécanique des fluides
	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 médecine préventive
	IDELMAN Simon	Physiologie animale
	JOLY Jean-René	Mathématiques pures
	JULLIEN Pierre	Mathématiques appliquées
Mme	KAHANE Josette	Physique
MM.	KRAKOWIACK Sacha	Mathématiques appliquées
	KUHN Gérard	Physique (I.U.T. I)
	LUU DUC Cuong	Chimie organique - pharmacie
	MICHOULIER Jean	Physique (I.U.T. I)
Mme	MINIER Colette	Physique (I.U.T. I)

.../...

MM.	PELMONT Jean	Biochimie
	PERRIAUX Jean-Jacques	Géologie et minéralogie
	PFISTER Jean-Claude	Physique du solide
Mlle	PIERY Yvette	Physiologie animale
MM.	RAYNAUD Hervé	M.I.A.G.
	REBECQ Jacques	Biologie (CUS)
	REYMOND Jean-Charles	Chirurgie générale
	RICHARD Lucien	Biologie végétale
Mme	RINAUDO Marguerite	Chimie macromoléculaire
MM.	SARROT-REYNAULD Jean	Géologie
	SIROT Louis	Chirurgie générale
Mme	SOUTIF Jeanne	Physique générale
MM.	STIEGLITZ Paul	Anesthésiologie
	VIALON Pierre	Géologie
	VAN CUTSEM Bernard	Mathématiques appliquées

MAITRES DE CONFERENCES ET MAITRES DE CONFERENCES AGREGES

MM.	ARMAND Yves	Chimie (I.U.T. I)
	BACHELOT Yvan	Endocrinologie
	BARGE Michel	Neuro-chirurgie
	BEGUIN Claude	Chimie organique
Mme	BERIEL Hélène	Pharmacodynamie
MM.	BOST Michel	Pédiatrie
	BOUCHARLAT Jacques	Psychiatrie adultes
Mme	BOUCHE Liane	Mathématiques (CUS)
MM.	BRODEAU François	Mathématiques (I.U.T. B) (Personne étrangère habilitée à être directeur de thèse)
	BERNARD Pierre	Gynécologie
	CHAMBAZ Edmond	Biochimie médicale
	CHAMPETIER Jean	Anatomie et organogénèse
	CHARDON Michel	Géographie
	CHERADAME Hervé	Chimie papetière
	CHIAVERINA Jean	Biologie appliquée (EFP)
	COLIN DE VERDIERE Yves	Mathématiques pures
	CONTAMIN Charles	Chirurgie thoracique et cardio-vasculaire
	CORDONNER Daniel	Néphrologie
	COULOMB Max	Radiologie
	CROUZET Guy	Radiologie

.../...

MM.	CYROT Michel	Physique du solide
	DENIS Bernard	Cardiologie
	DOUCE Roland	Physiologie végétale
	DUSSAUD René	Mathématiques (CUS)
Mme	ETERRADOSSI Jacqueline	Physiologie
MM.	FAURE Jacques	Médecine légale
	FAURE Gilbert	Urologie
	GAUTIER Robert	Chirurgie générale
	GIDON Maurice	Géologie
	GROS Yves	Physique (I.U.T. I)
	GUIGNIER Michel	Thérapeutique
	GUITTON Jacques	Chimie
	HICTER Pierre	Chimie
	JALBERT Pierre	Histologie
	JUNIEN-LAVILLAVROY Claude	O.R.L.
	KOLODIE Lucien	Hématologie
	LE NOC Pierre	Bactériologie-virologie
	MACHE Régis	Physiologie végétale
	MAGNIN Robert	Hygiène et médecine préventive
	MALLION Jean-Michel	Médecine du travail
	MARECHAL Jean	Mécanique (I.U.T. I)
	MARTIN-BOUYER Michel	Chimie (CUS)
	MASSOT Christian	Médecine interne
	NEMOZ Alain	Thermodynamique
	NOUGARET Marcel	Automatique (I.U.T. I)
	PARAMELLE Bernard	Pneumologie
	PECCOUD François	Analyse (I.U.T. B) (Personnalité étrangère habilitée à être directeur de thèse)
	PEFFEN René	Métallurgie (I.U.T. I)
	PERRIER Guy	Géophysique-glaciologie
	PHELIP Xavier	Rhumatologie
	RACHALL Michel	Médecine interne
	RACINET Claude	Gynécologie et obstétrique
	RAMBAUD Pierre	Pédiatrie
	RAPHAEL Bernard	Stomatologie
Mme	RENAUDET Jacqueline	Bactériologie (pharmacie)
MM.	ROBERT Jean-Bernard	Chimie-physique
	ROMIER Guy	Mathématiques (I.U.T. B) (Personnalité étrangère habilitée à être directeur de thèse)
	SAKAROVITCH Michel	Mathématiques appliquées

.../...

MM.	SCHAERER René	Cancérologie
Mme	SEIGLE-MURANDI Françoise	Crytogamie
MM.	STOEBNER Pierre	Anatomie pathologie
	STUTZ Pierre	Mécanique
	VROUSOS Constantin	Radiologie

MAITRES DE CONFERENCES ASSOCIES

MM.	DEVINE Roderick	Spectro Physique
	KANEKO Akira	Mathématiques pures
	JOHNSON Thomas	Mathématiques appliquées
	RAY Tuhina	Physique

MAITRE DE CONFERENCES DELEGUE

M.	ROCHAT Jacques	Hygiène et hydrologie (pharmacie)

Fait à Saint Martin d'Hères, novembre 1977

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

Année universitaire 1977-1978

Président : M. Philippe TRAYNARD

Vice-présidents : M. René PAUTHENET
M. Georges LESPINARD

PROFESSEURS TITULAIRES

MM.	BENOIT Jean	Electronique - automatique
	BESSON Jean	Chimie minérale
	BLOCH Daniel	Physique du solide - cristallographie
	BONNETAIN Lucien	Génie chimique
	BONNIER Etienne	Métallurgie
*	BOUDOURIS Georges	Electronique - automatique
	BRISSONNEAU Pierre	Physique du solide - cristallographie
	BUYLE-BODIN Maurice	Electronique - automatique
	COUMES André	Electronique - automatique
	DURAND Francis	Métallurgie
	FELICI Noël	Electronique - automatique
	FOULARD Claude	Electronique - automatique
	LANCIA Roland	Electronique - automatique
	LONGEQUEUE Jean-Pierre	Physique nucléaire corpusculaire
	LESPINARD Georges	Mécanique
	MOREAU René	Mécanique
	PARIAUD Jean-Charles	Chimie - physique
	PAUTHENET René	Electronique - automatique
	PERRET René	Electronique - automatique
	POLOUJADOFF Michel	Electronique - automatique
	TRAYNARD Philippe	Chimie - physique
	VEILLON Gérard	Informatique fondamentale et appliquée

* en congé pour études

PROFESSEURS SANS CHAIRE

MM.	BLIMAN Samuël	Electronique - automatique
	BOUVARD Maurice	Génie mécanique
	COHEN Joseph	Electronique - automatique
	GUYOT Pierre	Métallurgie physique
	LACOUME Jean-Louis	Electronique - automatique
	JOUBERT Jean-Claude	Physique du solide - cristallographie

.../...

MM.	ROBERT André	Chimie appliquée et des matériaux
	ROBERT François	Analyse numérique
	ZADWORNY François	Electronique - automatique

MAITRES DE CONFERENCES

MM.	ANCEAU François	Informatique fondamentale et appliquée
	CHARTIER Germain	Electronique - automatique
	CHIAVERINA Jean	Biologie, biochimie, agronomie
	IVANES Marcel	Electronique - automatique
	LESIEUR Marcel	Mécanique
	MORET Roger	Physique nucléaire - corpusculaire
	PIAU Jean-Michel	Mécanique
	PIERRARD Jean-Marie	Mécanique
	SABONNADIERE Jean-Claude	Informatique fondamentale et appliquée
Mme	SAUCIER Gabrielle	Informatique fondamentale et appliquée
M.	SOHM Jean-Claude	Chimie Physique

CHERCHEURS DU C.N.R.S. (Directeur et Maîtres de Recherche)

M.	FRUCHART Robert	Directeur de Recherche
MM.	ANSARA Ibrahim	Maître de Recherche
	BRONOEL Guy	Maître de Recherche
	CARRE René	Maître de Recherche
	DAVID René	Maître de Recherche
	DRIOLE Jean	Maître de Recherche
	KLEITZ Michel	Maître de Recherche
	LANDAU Ioan-Doré	Maître de Recherche
	MATHIEU Jean-Claude	Maître de Recherche
	MERMET Jean	Maître de Recherche
	MUNIER Jacques	Maître de Recherche

Personnalités habilitées à diriger des travaux de recherche (décision du Conseil Scientifique)

E.N.S.E.E.G.

MM.	BISCONDI Michel	Ecole des Mines St. Etienne (dépt. Métallurgie)
	BOOS Jean-Yves	Ecole des Mines St. Etienne (Métallurgie)
	DRIVER Julian	Ecole des Mines St. Etienne (Métallurgie)

.../...

MM.	KOBYLANSKI André	Ecole des Mines St. Etienne (Métallurgie)
	LE COZE Jean	Ecole des Mines St. Etienne (Métallurgie)
	LESBATS Pierre	Ecole des Mines St. Etienne (Métallurgie)
	LEVY Jacques	Ecole des Mines St. Etienne (Métallurgie)
	RIEU Jean	Ecole des Mines St. Etienne (Métallurgie)
	SAINFORT	C.E.N. Grenoble (Métallurgie)
	SOUQUET	U.S.M.G.
	CAILLET Marçel	Ecole des Mines St. Etienne (Chim. Min. Ph.)
	COULON Michel	Ecole des Mines St. Etienne (Chim. Min. Ph.)
	GUILHOT Bernard	Ecole des Mines St. Etienne (Chim. Min. Ph.)
	LALAUZE René	Ecole des Mines St. Etienne (Chim. Min. Ph.)
	LANCELOT Francis	Ecole des Mines St. Etienne (Chim. Min. Ph.)
	SARRAZIN Pierre	Ecole des Mines St. Etienne (Chim. Min. Ph.)
	SOUSTELLE Michel	Ecole des Mines St. Etienne (Chim. Min. Ph.)
	THEVENOT François	Ecole des Mines St. Etienne (Chim. Min. Ph.)
	THOMAS Gérard	Ecole des Mines St. Etienne (Chim. Min. Ph.)
	TOUZAIN Philippe	Ecole des Mines St. Etienne (Chim. Min. Ph.)
	TRAN MINH Canh	Ecole des Mines St. Etienne (Chim. Min. Ph.)

E.N.S.E.R.G.

MM.	BOREL	Centre d'études nucléaires de Grenoble
	KAMARINOS	Centre national recherche scientifique

E.N.S.E.G.P.

M.	BORNARD	Centre national recherche scientifique
Mme	CHERUY	Centre national recherche scientifique
MM.	DAVID	Centre national recherche scientifique
	DESCHIZEAUX	Centre national recherche scientifique

REMERCIEMENTS

Je tiens à remercier Monsieur L. BOLLIET, Professeur à l'I.U.T. B de Grenoble, pour l'honneur qu'il me fait de présider le Jury de cette thèse. Qu'il me soit permis d'évoquer ici sa présence bienveillante aux étapes importantes de ma vie professionnelle : lors de ma thèse de Docteur Ingénieur, à la fin de mon séjour aux Etats-Unis, à l'issue de nos premières recherches sur l'enseignement de la programmation. Je suis particulièrement sensible à l'intérêt discret qu'il a ainsi porté aux travaux que j'ai entrepris.

Je remercie Monsieur M. SAKAROVITCH, Maître de Conférences à l'U.S.M.G., Directeur du Laboratoire IMAG, pour avoir bien voulu lire cette thèse. Son jugement m'est d'autant plus précieux qu'il émane d'une personne non spécialisée dans le domaine abordé ici.

Je remercie Monsieur Ph. JORRAND, Maître de Recherches au CNRS, pour avoir accepté de participer à ce Jury. J'ai toujours été encouragé par sa curiosité pour mes idées dans les domaines de la méthodologie et de la didactique de la programmation.

Monsieur M. GALINIER, Maître de Conférences à l'Université Paul Sabatier de Toulouse, s'est intéressé dès le début aux travaux que je présente ici. Il a toujours apporté un soutien amical et confiant à mon travail, et il m'a permis de prendre un recul nécessaire au cours de discussions passionnées. J'ai grand plaisir à le remercier ici.

L'intérêt de Monsieur M. SINTZOFF, chercheur de la M.B.L.E. à Bruxelles, pour les travaux que je lui présentais, a été un facteur prédominant dans la rédaction de cette thèse. Je le remercie vivement d'avoir su m'encourager par un enthousiasme constant et de m'avoir communiqué une vision précise des problèmes importants de la méthodologie de la programmation.

Monsieur G. VEILLON, Professeur à l'I.N.P.G. a constamment suivi mes travaux en algorithmique et a toujours soutenu les expériences d'enseignement que nous avons pu proposer. Je tiens à le remercier particulièrement de m'avoir accueilli dans son équipe de chercheurs, et d'avoir accepté la responsabilité de mes recherches.

Monsieur M. GRIFFITHS, Professeur à l'I.U.T. de Nancy, a guidé mes débuts à l'Université. Il a su m'apprendre à définir mes centres d'intérêt et à organiser mon travail. Il m'a toujours témoigné une grande confiance, au sein de l'Institut de Programmation où il m'a laissé exercer mes propres responsabilités et par la suite tout au long d'un travail en commun, fructueux malgré la distance géographique. Je suis heureux de lui présenter enfin cette thèse, et j'ai grand plaisir à profiter de cette occasion très formelle pour le remercier sincèrement.

Monsieur M. LUCAS, Maître de Conférences à l'Université de Nantes, a été le compagnon de toutes ces années de travail. Son efficacité et son réalisme ont été prédominants dans la concrétisation de nos idées, notamment par les ouvrages réalisés en commun et par les expériences didactiques et pédagogiques que nous avons partagées. Son enthousiasme a été une source constante d'encouragement dans la conduite de mes travaux. Enfin, son oreille attentive et sa grande expérience ont toujours été à ma disposition au cours de ces derniers mois pendant lesquels cette thèse a été conçue puis rédigée. Je le remercie ici pour sa disponibilité d'esprit, sa patience et sa fidélité.

Ces remerciements ne seraient pas complets, si je ne tentais pas d'évoquer ici les personnes qui ont participé de près ou de loin, à l'élaboration et à la validation des recherches présentées dans cette thèse :

Mes collègues enseignants tout d'abord, qui ont accepté d'expérimenter les idées que nous proposions et qui les ont fait progresser par leurs critiques constructives : F. MARTINEZ, J.P. PEYRIN, E. TOURNIER, F. VEILLON, J. VOIRON ;

Les étudiants et notamment ceux de l'Institut de Programmation qui ont permis l'expérimentation immédiate de toute nouvelle idée sur la méthodologie de la programmation ou sur les notations algorithmiques ;

Les membres du groupe "enseignement de l'informatique" de l'AFCET qui ont toujours écouté nos présentations d'une oreille attentive et nous ont permis de prendre le recul nécessaire au cours de confrontations riches en enseignements.

P. Y. CUNIN, chercheur à l'Université de Nancy, avec lequel j'ai passé de longues heures à discuter de problèmes méthodologiques lors de la conception du langage MEFIA.

et enfin les membres du sous-groupe de travail "ANNAGRAM" du groupe "programmation et langages" de l'AFCET, dont les réactions intéressées et très critiques ont été déterminantes dans la rédaction de cette thèse.

Je souhaite que le lecteur apprécie la réalisation pratique de cette thèse due au service tirage du Laboratoire, et sa présentation : la frappe de cette thèse est l'oeuvre de G. BICAIS que je remercie ici pour sa patience, son efficacité et sa bonne humeur.

Pierre-Claude SCHOLL

TABLE DES MATIERES

VERS UNE PROGRAMMATION SYSTEMATIQUE

ETUDE DE QUELQUES METHODES
TECHNIQUES ET OUTILS

AVANT-PROPOS

Cette thèse est une contribution à l'effort collectif contemporain dans le domaine de la méthodologie de la programmation. Nous proposons deux méthodes originales de construction systématique de programmes qui ont été développées à Grenoble au cours des cinq dernières années ([LuS 75], [Sch 76],[LSV 77], [Sch 78 a et b]), et nous discutons du rôle des notations dans le processus de construction de programmes. Nous contribuons ainsi à la programmation systématique, tant par les démarches illustrées que par des perspectives sur le plan des logiciels d'aide à la programmation. La thèse débouche sur un projet de système interactif d'aide à la programmation : ce projet dont une première forme avait été donnée dans ([ScC 76]) se base sur le langage MEFIA, fruit d'une collaboration interuniversitaire ([Gri 77], [CGS 78] ,[SGC 78]) et qui a fait par ailleurs l'objet de la thèse d'Etat de P.Y. CUNIN ([Cun 79]).

Dix années d'expériences diverses en informatique sont ainsi concrétisées :

- conception et réalisation de logiciels de taille importante ([Sch 70], [Sch 72]) : base essentielle pour la compréhension de l'activité de programmation, et pour la perception du besoin de méthodes.

- expériences didactiques en enseignement assisté par ordinateur ([ScS 72], [FPS 76]) : ouverture aux problèmes ressentis par les non spécialistes en informatique, vis à vis d'une discipline impérialiste et mystifiante.

- expériences didactiques et pédagogiques dans le cadre de l'université, en formation initiale et continue ([LMS 75], [LuS 75 et 76] ,[Sch 77 b] [Sch 79 a et b]) : catalyseur d'une aspiration à la maîtrise des connaissances et révélateur d'un goût pour leur transmission. Ces expériences didactiques sont à l'origine de notre recherche médhogoloqique, et sont le support permanent d'une validation des idées et des résultats.

- expériences d'encadrement de projets : ouverture sur l'animation et l'orientation de projets de recherche.

INTRODUCTION

La programmation des ordinateurs a radicalement changé durant les années 70 du fait de la reconnaissance d'une simple vérité par tous les milieux professionnels, que ce soit dans l'Industrie ou à l'Université.

La programmation doit être pratiquée de manière systématique, avec méthode.

Evolution de la programmation

Le besoin d'une programmation méthodique est devenu crucial avec l'apparition dans les années 60 des premiers grands systèmes logiciels. Les problèmes de fiabilité posés par ces programmes semblent à l'origine de plusieurs phénomènes complémentaires qui caractérisent les débuts de l'évolution de la programmation :

. Le développement de la notion de preuve de programmes, annoncée dans [Mac 62] et concrétisée par les articles parus à partir de 1966 ([Nau 66],[Flo 67 a],[Bur 69], [Hoa 69]).

. La promotion de nouvelles habitudes de programmation ([Dij 65]), et l'élaboration d'un premier groupe de méthodes ([Dij 71], [Wir 71 b], [Par 72 a et b]). La parution du livre "Structured Programming" ([DDH 72]) marque cette première phase d'étude du processus de construction de programmes.

. Une nouvelle orientation dans la conception des langages, intégrant les idées modernes sur la programmation. Le langage SIMULA ([DMN 68]) est un précurseur par les outils d'abstraction qu'il comporte. Le langage PASCAL ([Wir 71 a]) marque une étape dans le développement de la programmation systématique ([Wir 73]). Il répond de manière satisfaisante à des critères variés (puissance d'expression, simplicité didactique, systématique dans l'utilisation, facilité dans l'écriture de compilateurs) tout en restant relativement simple (par rapport à la complexité croissante des langages).

Remarque :
On notera au passage la controverse spectaculaire mais a postériori secondaire, relative à l'instruction *GOTO* (cf. [Lea 72] pour un bref résumé et une importante bibliographie, et [LeM 75] pour une synthèse sur les structures de contrôle).

L'avènement de ce changement dans la programmation est confirmé par son impact dans le monde industriel (parution d'un numéro de la revue américaine DATAMATION consacré à la "programmation structurée, cf. [DAT 73]), et par son intégration dans l'enseignement (de formation initiale et de formation permanente). Ceci apparaît nettement dans l'évolution des ouvrages didactiques : au fil des ans, on passe du manuel de référence de tel ou tel langage à une présentation "structurée" de ces langages, pour finalement aboutir à des traités sur la construction des programmes (cf. Annexe A). Par ailleurs, on voit apparaître de nombreuses synthèses sur la programmation : dans les revues spécialisées, ([CS 74]), à l'occasion d'écoles sur le sujet ([CGL 75]) et dans des ouvrages de synthèse (notamment [Yeh 77] où l'on trouve une bibliographie annotée comportant plus de deux cents titres).

Méthodologie de la programmation

C'est sur ces bases que s'est développée une nouvelle discipline informatique, la "méthodologie" (étude des méthodes) de la programmation. Elle présente deux particularités qui la distingue des autres disciplines :

. elle s'appuie sur de nombreux domaines, dont il faut pourtant la dissocier ([Pai 78]) : spécification formelle des programmes et des langages, preuve (automatique ou non) des programmes, construction systématique et synthèse automatique des programmes, génie du logiciel.
Ceci est clairement apparu dans le dernier congrès de l'AFCET [AFC 78] et dans les écoles récentes sur le sujet ([CCE 77], [BRE 78], [MOD 78]).

. elle concerne directement tous les utilisateurs de l'ordinateur quel que soit le niveau où se situe leur communication avec l'ordinateur, i.e. quels que soient le niveau de langage de programmation qu'ils utilisent, et l'application informatique qu'ils pratiquent.

Le premier point donne à la méthodologie de la programmation une mission fondamentale de synthèse, le second point accentue l'importance de ses aspects didactiques.

Cette thèse est une contribution à l'effort collectif sur la méthodologie de la programmation. Nous y proposons deux méthodes originales de construction de programmes, le *"traitement séquentiel"* et le *"traitement arborescent"* et nous discutons du rôle des *"notations algorithmiques"* dans le processus de construction de programmes.

Cadre directeur des méthodes présentées

Deux constatations sont à la base des méthodes présentées dans les deux premières parties de la thèse :

1°) la méthode d'analyse la plus couramment utilisée consiste à ramener les problèmes étudiés à des problèmes connus.

2°) Une grande partie de l'activité de programmation revient à appliquer la démarche suivante :

a) décomposer l'information en un ensemble d'informations plus élémentai
b) formuler le problème étudié en termes
- soit d'une recherche d'un élément de cet ensemble vérifiant une certaine propriété
- soit de l'application d'une même action élémentaire à tous les éléments de l'ensemble (dans un ordre précis ou non).

Remarque :
Notons que ces deux problèmes de base correspondent aux deux quantificateursexistentiel et universel de la théorie des ensembles.

De la première constatation, nous déduisons notre objectif : permettre le recours systématique aux solutions de problèmes déjà traités, en développant des méthodes d'analyse et des techniques de mise en oeuvre adéquates.

De la deuxième constatation, nous tirons le modèle des problèmes que nous voulons étudier : étant donné un ensemble d'informations, comment le

structurer pour pouvoir y rechercher un élément particulier ou énumérer tous ses éléments.

Principe général des méthodes présentées

Les méthodes que nous proposons s'inscrivent dans ce cadre directeur, en tirant parti des techniques classiques concernant les listes et les arbres ([Knu 69]), structures de données concrètes couramment utilisées pour représenter des ensembles. Elles reposent sur les principes simples suivants :

1°) Raffiner l'information en un ensemble d'informations élémentaires, et structurer cet ensemble en se servant de l'un des deux modèles suivants : la structure de "file" ou la structure d'"arbre". Ces deux modèles caractérisent les deux méthodes : dans le premier cas (traitement séquentiel) on est amené à définir un ordre total sur l'ensemble considéré, dans le deuxième cas (traitement arborescent), il s'agit d'un ordre partiel.

2°) Analyser le problème posé en se basant sur cette structure de l'information.

Cette démarche générale présente un double intérêt du point de vue de la construction des programmes :

. L'étude de l'information caractérisant le problème posé est guidée par la structure de référence (file ou arbre) : on connaît a priori l'ensemble des propriétés que doivent vérifier les composants de l'information pour pouvoir être organisés selon cette structure, et au besoin, on dispose de formalismes précis pour exprimer cette mise en forme.

. La découverte d'un algorithme et sa programmation dans un environnement informatique précis peuvent être entreprises de manière beaucoup plus systématique une fois que la structure a été choisie : les nombreux algorithmes traitant des files ou des arbres sont autant de modèles que l'on peut essayer d'appliquer.

Pour aider à l'application systématique de ces principes, nous proposons des outils de mise en oeuvre : modèles d'analyse et de description, schémas d'algorithmes :

. un modèle du processus de structuration (en file ou en arbre)

. un modèle de description de l'organisation logique de l'information

. un modèle d'analyse des traitements guidéepar la structure de référence

. des schémas d'algorithmes correspondant aux trois problèmes fondamentaux suivants :

1°) *"parcours"* : le traitement de l'ensemble est ramené à l'application d'une même action élémentaire à chacun des éléments de l'ensemble.

2°) *"énumération partielle"* : le traitement de l'ensemble est ramené à l'application d'une même action élémentaire à certains éléments de l'ensemble dotés d'une propriété caractéristique.

3°) *"recherche associative"* : le traitement de l'ensemble est ramené à l'application d'une action élémentaire à l'un des éléments de l'ensemble.

Présentation des méthodes

La première partie de cette thèse est consacrée au traitement séquentiel la seconde partie au traitement arborescent. La présentation des deux méthodes souligne leurs points communs :

1°) Définition récurrente de la structure de référence, présentation de modèles de structuration et de description, analyse des trois problèmes fondamentaux et construction des algorithmes récursifs correspondants (chapitres 1 et 5).

2°) Etude de la programmation des schémas de ces algorithmes dans un environnement informatique conventionnel (c'est-à-dire comportant la manipulation de variables à l'aide d'identificateurs et d'affectation) : cette étude met l'accent sur la description précise du processus de construction de programmes : chaque schéma présenté est ainsi à la fois un résultat que l'on peut appliquer directement dès lors que les conditions de son utilisation sont reconnues, et un modèle de démarche que

l'on peut consulter et imiter, lorsque cette identification n'est pas aussi précise. (chapitres 2 et 3 pour le traitement séquentiel, chapitre 6 pour le traitement arborescent).

3°) Application des deux méthodes à l'étude d'algorithmes récursifs (chapitres 4 et 7).

Pour illustrer l'application de ces méthodes, nous avons choisi des exemples relativement simples de manière à ce que leur présentation précise puisse rentrer dans le cadre de cette thèse. Il est clair que nos expérimentations ont aussi porté sur des problèmes plus importants comme le font apparaître les références bibliographiques.

Notations algorithmiques

Notre travail de recherche méthodologique ne s'est pas limité au développement d'idées sur la manière d'aborder un problème puis de le programmer. Nous nous sommes aussi interrogés sur le support d'expression des algorithmes : cette réflexion nous a conduit à développer l'idée centrale de notation algorithmique, intermédiaire entre les méthodes que l'on veut pratiquer et l'environnement informatique dans lequel on évolue. Ce rôle d'intermédiaire apparaît à deux niveaux :

. au niveau de l'application directe des méthodes par le programmeur : les notations algorithmiques permettent de développer une "méthode de comportement" du programmeur vis à vis de l'environnement informatique, et notamment des langages de programmation. Ce type de comportement est à la base d'une portabilité des programmes aussi bien que des programmeurs.

. au niveau de notre démarche méthodologique : la notation algorithmique nous permet de concrétiser nos idées sous forme d'un langage : le langage MEFIA, élément central d'un projet de logiciel d'aide à la programmation systématique, dont la présentation conclue la thèse.

Le travail que nous présentons ici n'est pas l'oeuvre d'un seul, mais plutôt le fruit d'une réflexion collective qui a fait participer tous les représentants des groupes concernés par la programmation : des universitaires surtout, chercheurs, enseignants et étudiants, mais aussi des personnes du monde industriel notamment par le biais de la formation permanente, et des non spécialistes de l'informatique, les plus concernés par cette discipline, mais les plus désarmés devant elle, notamment par le biais d'activités faisant intervenir des professeurs de l'enseignement secondaire.

Puisse ce travail convaincre que recherche et enseignement sont deux activités fondamentalement complémentaires.

CHAPITRE 0

NOTATIONS

Ce chapitre préliminaire est consacré à un bref résumé des notations utilisées et des conventions appliquées pour l'évaluation des algorithmes.

0.1 - NOTATION ALGORITHMIQUE

Pour décrire les algorithmes, nous utilisons un noyau de la notation MEFIA, formalisme de style ALGOL : nous présentons ici un résumé des seules constructions utilisées au cours de la thèse. Nous renvoyons le lecteur au chapitre 9 pour une présentation plus complète de la notation et pour une discussion approfondie des concepts sous jacents.

0.1.1. - Génération de valeurs

a) Constantes :

- On utilise les conventions standard pour dénoter les constantes (entier, réel, logique, caractères)
- On admet des n-uplets de constantes. Les constantes sont séparées par des virgules.

b) Expressions :

- On utilise le formalisme classique d'expression avec les conventions habituelles relatives aux priorités.
- On utilise des expressions conditionnelles dont la forme est directement déduite des instructions conditionnelles.
- Notons que nous utilisons les deux opérateurs logiques *et_conditionnel* (noté *et_c*) et *ou_conditionnel* (noté *ou_c*) définis comme suit :
  A et *B* étant des expressions à valeurs logiques :
  A et_c B : si A alors B sinon faux fsi
  A ou_c B : si ¬A alors B sinon vrai fsi

0.1.2. - Procédures

Le mécanisme de procédure suit les règles suivantes :

a) Un paramètre ne peut être modifié par la procédure (apparaître en partie gauche d'une affectation).

b) On distingue :

b.1 les actions :

. leur effet est caractérisé par une modification de variables

action nom (<liste de paramètres>) :
<corps de l'action>
faction nom

. les actions sont utilisées partout où l'on peut avoir une instruction (instruction d'appel de procédure classique comportant la liste des paramètres effectifs).

b.2 Les fonctions :

. Elles calculent une valeur ou un n-uplet de valeurs.

fonction nom (<liste de paramètres>)→<liste des types du résultat>
corps de la fonction
ffonction nom

. La valeur calculée par une fonction est décrite comme suit :

→<liste des valeurs du résultat>

. Si une fonction a un résultat simple (une seule valeur), elle peut être utilisée dans toute expression en place d'un opérande quelconque.

. Si une fonction a un résultat composé (un n-uplet de valeurs), elle ne peut être utilisée que dans une instruction d'affectation :

<liste de variables simples non indicées>:= nom (<liste de paramètres effectifs>)

c) Actions et fonctions peuvent être récursives.

0.1.3. - Instructions

a) Choix

. L'instruction *choix* permet d'établir une relation entre un ensemble de cas et un ensemble d'actions. Les cas envisagés doivent s'exclure mutuellement.

```
choix
    <expression logique 1>:<groupe d'instructions 1>
    ...
    <expression logique n>:<groupe d'instructions n>
fchoix
```

. L'exécution d'une telle instruction consiste à découvrir l'expression logique qui prend la valeur *vrai* et à exécuter le groupe d'instructions associé. Il y a erreur si aucune des expressions logiques ne prend la valeur *vrai*

. Les formes habituelles d'instructions conditionnelles sont admises (et considérées comme des "séquentialisations"de l'instruction *choix*) :

```
si <expression logique>
   alors <groupe d'instructions 1>
   sinon <groupe d'instructions 2>
fsi

si <expression logique> alors
   <groupe d'instructions>
fsi
```

b) Itération

. La forme principale d'itération est la suivante :

```
itérer
    <groupe d'instructions 1>
arrêt : <expression logique>
    <groupe d'instructions 2>
fitérer
```

en termes du schéma *tantque* classique, le schéma *itérer* a le sens suivant :

```
<groupe d'instructions 1>
tantque ¬<expression logique> faire
      <groupe d'instruction 2>
      <groupe d'instruction 1>
ftantque
```

. Nous utilisons aussi les schémas classiques :
tantque, répéter et *pour* (considérés comme des formes particulières du schéma *itérer.)*

0.1.4. - Mise en page

a) Commentaires

Un commentaire est encadré par des accolades :

{commentaire}

b) Indentation

A chacune des constructions proposées ci-dessus correspond une mise en page précise permettant de mettre en évidence la structure de l'algorithme.

c) Identificateurs

Nous réservons l'usage des majuscules pour désigner les objets qui ne sont pas définis dans le texte de l'algorithme

d) Délimiteurs

Le ; est le délimiteur d'instruction (indiquant une exécution en séquence). Il est systématiquement omis lorsqu'il se trouve en fin de ligne ou avant un mot clé.

0.2 - EVALUATION DES ALGORITHMES

0.2.1. - Rôle des évaluations

L'évaluation des algorithmes joue un triple rôle (cf. § 2.2.1) :

* vérification de la cohérence des algorithmes
* mesure du comportement des algorithmes
* estimation du coût de l'algorithme selon un modèle particulier d'évaluation.

0.2.2. - Présentation des évaluations

Nous portons en regard de chaque instruction élémentaire de l'algorithme (instruction de base de la notation, ou action non détaillée dans l'algorithme) le nombre de ses exécutions exprimé en fonction de paramètres du problème traité (la recherche et l'analyse de ces paramètres est d'ailleurs la question la plus délicate dans de telles évaluations).

L'algorithme est en général suivi d'un récapitulatif, intitulé "évaluation", dans lequel nous cumulons les chiffres correspondant à des instructions assimilables du point de vue des évaluations. Dans ce récapitulatif on trouve ainsi un certain nombre de rubriques au sens évident : affectations, opérations logiques, opérations arithmétiques. Sous l'intitulé général "contrôle", nous comptabilisons le coût occasionné par les schémas de composition (choix, itérations).

Remarque :

Nous prenons pour modèle d'évaluation, une traduction en termes de tests et de branchements : chaque itération (passage sur le test contrôlant l'itération) coûte un test et un branchement et chaque choix coûte x-1 tests et branchements, si x est le nombre de cas du choix. Sous l'intitulé "primitives" nous récapitulons l'ensemble des actions non détaillées dans l'algorithme .

PREMIERE PARTIE

LE TRAITEMENT SEQUENTIEL

INTRODUCTION

Le traitement séquentiel est une méthode de construction systématique de programmes que nous avons étudiée, développée et expérimentée à Grenoble (LuS75, LSV77, Luc78, Sch78a et b). Cette méthode est fondée sur les principes simples suivants :

1. Recherche d'une organisation logique de l'information permettant de la structurer en une suite d'informations plus élémentaires.

2. Recherche d'une formulation du problème posé, en termes d'un traitement de cette suite.

La première étape rejoint dans son principe des méthodes proposées par ailleurs : construction d'itérations par recherche d'invariants, ([Flo 67],[Hoa 69], [Dij 72 et 76]) [MaW 77] ; construction d'algorithmes récurrents ([Wir 73][Ars 77][AsW 77][Bau 78]); itérations en programmation déductive([Pai 77a], [BFH 79]).

La deuxième étape est basée sur l'identification d'un schéma général de traitement de suite qui corresponde au problème posé. Une part importante de notre travail méthodologique est la définition de schémas fondamentaux et leur formulation sous diverses formes. On peut ainsi les appliquer de manière systématique, quel que soit l'environnement de programmation et notamment quel que soit le degré d'évolution du langage utilisé.

Dans un premier chapitre, nous donnons les principes du traitement séquentiel : nous proposons quelques définitions et notations, nous présentons un modèle de construction d'algorithmes et nous construisons les algorithmes fondamentaux qui sont à la base de l'application du traitement séquentiel. Le deuxième chapitre est consacré à la programmation de ces algorithmes fondamentaux dans un langage de programmation "classique" faisant intervenir variables et affectations : on fait apparaître diverses manières de réaliser ces algorithmes, en fonction de questions précises liées au niveau de programmation choisi. Le chapitre 3 montre trois exemples d'application systématique du traitement séquentiel. Enfin, dans le chapitre 4, nous appliquons le traitement séquentiel à l'étude d'algorithmes récursifs.

CHAPITRE 1

PRINCIPES DU TRAITEMENT SEQUENTIEL

Nous montrons ici comment structurer une information en une *"file"* d'informations plus élémentaires et comment construire un algorithme à partir d'une telle interprétation de l'information. Enfin, nous construisons les algorithmes fondamentaux qui résolvent les problèmes de base que nous avons donnés dans l'introduction : énumération, énumération partielle et recherche.

La notion de *"file"* est l'élément central dans la maîtrise du traitement séquentiel. Elle s'inspire d'une structure de données classique dont le nom varie selon les auteurs : *"liste linéaire"* dans (Knu 69), *"liste"* dans (Pai 71), *"séquence"* dans (Hoa 72) et (Dah 77), *"file"* dans (CoV 74a), (Kun 76), (Wir 76) *"suite"* dans (MaB 78).

Nous utilisons le concept de *"file"* pour guider le processus de construction de programmes : il sert de modèle pour raffiner l'information caractérisant le problème analysé. Si ce raffinement est possible, on peut interpréter ce problème comme un traitement de cette file, et finalement déduire un algorithme de cette interprétation. La notion de *file* doit donc être définie de manière à répondre à deux objectifs :

. permettre la description du processus de structuration en file (processus d'analyse)

. permettre la description d'algorithmes de traitement de files (résultat de l'analyse).

On doit souligner qu'en aucun cas nous n'avons à décrire à ce niveau une création ou une modification de file. C'est pourquoi le concept de file que nous présentons ici est sensiblement différent de la structure de données dont il s'inspire. Nous utilisons l'expression *"file abstraite"* (caractéristique du traitement séquentiel) lorsqu'il peut y avoir ambiguité.

Remarque :
Si l'on considère une file représentée en mémoire, et par exemple la mise à jour de cette file par une opération d'insertion, cette mise à jour est elle-même un traitement séquentiel dont la "file abstraite" caractéristique est la sous-file se terminant au point d'insertion.

1.1 - GENERALITES SUR LES FILES

1.1.1. - Définition

Nous donnons la définition récurrente suivante :

Une file est un ensemble non vide sur lequel existe une partition entre :

. l'un des éléments, appelé le *"premier"* élément de la file

. les éléments restants, s'il en existe : ils constituent eux-mêmes une file, appelée *"sous-file"* issue du premier élément.

Analysant cette définition, on peut déduire deux propriétés des files (par simple raisonnement par récurrence sur l'ensemble des files ordonné selon la taille des files) :

. une file fait apparaître autant de partitions entre un premier élément et une sous-file, qu'il y a d'éléments dans la file : chaque élément est en fait le premier d'une sous-file.

. il existe un élément et un seul qui soit premier d'une sous-file de taille 1. Cet élément est appelé le *"dernier"* élément de la file.

Une conséquence intéressante de ces propriétés est que l'on peut établir une bijection entre l'ensemble des éléments de la file et l'ensemble de ses sous-files. On désignera notamment une sous-file par le nom de son premier élément.

La définition ci-dessus fait apparaître des relations entre les sous-files d'une file. Sur la base de la bijection que nous venons de souligner, nous pouvons faire apparaître des relations analogues entre les éléments de la file :

. E étant un élément autre que le dernier de la file, et SF la sous-file issue de E, on définit le *"successeur"* de E comme étant le premier élément de SF.

. inversement, E est le *"prédécesseur"* du premier élément de SF

Clairement, la relation *"successeur"* induit un ordre total sur l'ensemble des éléments de la file.

1.1.2. - Structuration d'une information en file

Le processus qui permet de structurer une information en file comporte deux étapes :

1 - Raffinement de l'information : on décompose l'information en un ensemble d'informations plus élémentaires.

2 - Structuration de l'information : l'ensemble ainsi obtenu est alors structuré en file.

Pour structurer un ensemble d'informations en file, nous appliquons le processus récursif suivant :

. désigner l'une des informations comme la première de la file que l'on construit.

. structurer l'ensemble des informations restantes, s'il y en a, en file.

1.1.3. - Description d'une file : un modèle de notations

Nous proposons ici des notations pour la description d'une file. On peut les considérer comme un résumé du processus de structuration en file.

Ces notations permettent de faire abstraction de la représentation effective de la file considérée : tous les éléments de la file peuvent être présents en mémoire (par exemple dans le cas de fichiers séquentiels, de listes linéaires ou même de vecteurs utilisés de manière séquentielle) ; mais on peut aussi considérer des files "calculées" pour lesquelles un seul élément est présent en mémoire à tout instant, les autres (la sous-file en étant issue) s'en déduisant par calcul ; enfin, on peut envisager des situations intermédiaires, où les éléments sont tous présents en mémoire, mais où l'ordre entre ces éléments est défini par calcul au fur et à mesure du traitement séquentiel.

Ce souci d'abstraction nous conduit à décrire une file par un nombre de primitives qui n'est pas minimum (contrairement à des définitions axiomatiques par exemple) : cette redondance permet une plus grande liberté dans l'établissement des algorithmes de traitement.

Un dernier rôle de ces notations est de nous permettre de définir les algorithmes généraux de traitement séquentiel en faisant totalement abstraction à la fois de l'origine de la file (ce qui dans le problème étudié a conduit la structuration en file) et de sa représentation (dont le choix est effectué après avoir défini les algorithmes de traitement solutionnant le problème posé).

Description d'une file

Les étapes de la description d'une file sont :

1 - la description de l'information caractérisant chaque élément de la file (résultat du raffinement)

2 - la description des relations entre les éléments de la file (résultat de la structuration en file).

Notations

. tout élément *E* de la file représente la sous-file dont il est le premier.

. *PREMIER* en abrégé *PRE*
désigne le premier élément de la file

. *SUCCESSEUR(E)* en abrégé *SUC(E)*
désigne la sous-file issue d'une file dont le premier élément est E

. *EXISTE_SUCCESSEUR(E)* en abrégé *E_S(E)*
est un prédicat indiquant si la file dont le premier élément est E comporte plus d'un élément *(E_S(E)=vrai)* ou un seul élément *(E_S(E)=faux)* .

Nous utilisons de plus les notations suivantes pour désigner l'accès aux éléments de la file inverse (i.e. pour décrire la file inverse) :

. *DERNIER* en abrégé *DER*
désigne le dernier élément de la file (celui qui n'a pas de successeur)

. *PREDECESSEUR(E)* en abrégé *PRED(E)*
désigne l'élément de la file qui a pour successeur (la file de premier élément) *E*.

. *EXISTE_PREDECESSEUR(E)* en abrégé *E_P(E)*
est un prédicat indiquant si l'élément *E* a un prédécesseur (i.e. si ce n'est pas le premier de la file).

Enfin, l'information propre attachée à un élément *E* est désignée par :

INFO(E)

Remarque :
Dans la suite nous utilisons le plus souvent les formes abrégées de ces notations. Les formes complètes servent essentiellement lors de l'expression orale.

1.1.4. - Files vides

On peut vouloir dans certains cas introduire la notion de *"file vide"*, qui correspond au cas (que nous avons écarté jusqu'à présent) où l'ensemble considéré est vide.

On peut avoir deux attitudes : considérer le cas de la file vide indépendamment du cas général, ou intégrer le cas de la file vide dans le cas général (la deuxième attitude étant plus sophistiquée mais conduisant à des algorithmes en général plus concis).

a) On considère le cas de la file vide indépendamment du cas général : la structuration en file n'est faite que si l'ensemble n'est pas vide, en appliquant ce qui précède. Le cas de l'ensemble vide reste un cas particulier, que l'on distingue à l'aide d'un prédicat

FILE_VIDE

qui permet de dissocier le cas où la file est vide *(FILE_VIDE=vrai)* du cas où elle ne l'est pas *(FILE_VIDE=faux)*.

b) On intègre le cas de la file vide au cas général : la notion de file vide est effectivement introduite en donnant la nouvelle définition suivante :

Une file est un ensemble qui :
- ou bien est vide
- ou bien n'est pas vide : il existe alors une partition entre
  . l'un de ses éléments, appelé le *"premier"* élément de la file
  . l'ensemble des éléments restants, qui constitue lui-même une file, appelée la *"sous-file"* issue du premier élément.

Cette définition permet de ne plus distinguer lors de la structuration, le cas particulier de la file vide. Le processus de structuration aboutit toujours à une file vide (alors que dans ce qui précède, il aboutit toujours à une file comportant un seul élément).

Du point de vue des notations, *PRE, SUC(E), PRED(E), INFO(E)* gardent les significations données ci-dessus. Mais la spécification d'une file doit maintenant comporter :

- la définition d'une propriété caractéristique des files vides, qui soit compatible avec la définition de l'information caractérisant un élément de la file,
- la définition d'un prédicat permettant de reconnaître une file vide :

  EST_VIDE(E)

  indique si la file désignée par *E* est vide *(EST_VIDE(E)=vrai)* ou non *(EST_VIDE(E)=faux)*.

Dans ces conditions, la notion de dernier élément est définie comme suit : le dernier élément d'une file est celui dont le successeur est la file vide (le dernier élément d'une file vide n'est toutefois pas défini).

Remarque :
Pour intégrer le cas de la file vide dans la spécification de la file, on utilise souvent une convention particulière qui a pour effet de se ramener au cas où la file n'est jamais vide : on caractérise la file vide à l'aide d'un élément particulier que nous notons *NIL* : *NIL* est un élément (qui répond aux spécifications générales des éléments de la file) choisi de manière à être distinct de tout élément pouvant appartenir à une file non vide. Suivant les cas, *INFO (NIL)* aura un sens ou pas.

1.1.5. - Exemples de descriptions de files

Les exemples qui suivent ont pour but de montrer comment nos notations permettent de faire abstraction des représentations. C'est pourquoi nous les présentons de manière ascendante : nous partons d'une représentation et donnons la description correspondante de la file dans nos notations.

a) Exemple 1

On se donne une suite de caractères implantée dans un tableau

tableau(1:N) car T

avec la convention suivante pour déterminer la fin de la suite : la suite se termine par le caractère (".").

Nous pouvons alors considérer les files suivantes :

. file de caractères

1. description de l'information :
   une position dans le tableau *T* suffit pour caractériser un élément de la file

2. description des relations :
   les primitives sont :

   PRE : *1*
   SUC(i) : *i+1*
   E_S(i) : *T(i)≠point* {*point* désigne le caractère "."}
   PRED(i) : *i-1*
   E_P(i) : *i≠1*

   enfin l'information propre à chaque élément est :

   INFO(i) : *T(i)*

Remarque :
La définition de la convention de fin de suite donnée dans l'énoncé ci-dessus, implique qu'on n'utilise pas ici la notion de file vide (une file comporte au moins le point final). Une autre interprétation de l'énoncé consiste à considérer que le point final est l'élément caractérisant la file vide).

. file des couples de caractères consécutifs dans le tableau T

1. description de l'information :
   un élément de la file est caractérisé par deux positions successives dans le tableau *T*. Un entier suffit donc pour caractériser un élément (un couple). La file est vide si le premier caractère du tableau est un point.

2. description des relations :

   FILE_VIDE : *T(1)=point*
   PRE : *1*
   SUC(i) : *i+1*
   E_S(i) : *T(i+1)≠point*
   INFO(i) : *T(i), T(i+1)*
   PRED(i) : *i-1*
   E_P(i) : *i≠1*

Remarque :
On peut considérer que la file vide est caractérisée par un couple dont le premier élément est le point et dont le deuxième élément est quelconque et peut même ne pas exister. On a alors :

EST_VIDE(i) : *T(i)=point*
PRE : *1*
SUC(i) : *i+1*
INFO(i) : *T(i), T(i+1)* {non défini si *EST_VIDE(i)=vrai*}

b) Exemple 2

On considère l'ensemble des carrés des N premiers entiers.

. une_première_manière de structurer cet ensemble en file est la suivante :

1. description de l'information :
   un élément de la file est caractérisé par un entier
2. description des relations : les primitives sont :

FILE_VIDE : $N = 0$
PRE : *1*
SUC(i) : *i+1*
E_S(i) : $i \neq N$
PRED(i) : *i-1*
E_P(i) : $i \neq 1$
INFO(i) : i^2

Remarque :
Si l'on veut traiter le cas $N=0$, on a deux possibilités : considérer qu'alors la file est vide, ou considérer que $N+1$ correspond à l'élément *NIL* (pour lequel *INFO* n'est pas définie)

. une_deuxième_manière de structurer l'ensemble des carrés en file est basée sur la remarque suivante :

$$\forall i : 1 \leq i \quad , \quad \sum_{j=1}^{i} (2j-1) = (2 \sum_{j=1}^{i} j) - i$$

$$= i(i+1) - i = i^2.$$

1. description de l'information :
   un élément de la file est caractérisé par deux entiers : un nombre impair *IP* et la somme *S* des nombres impairs de *1* à *IP*.

2. description des relations :

$$
\begin{aligned}
&PRE && : && 1\ ,\ 1\\
&SUC(ip,s) && : && ip+2,\ s+ip+2\\
&E\_S(ip,s) && : && ip \neq 2N-1\\
&PRED(ip,s) && : && ip-2,\ s-ip\\
&E\_P(ip,s) && : && ip \neq 1\\
&INFO(ip,s) && : && s
\end{aligned}
$$

1.1.6 - Analyse récurrente et traitement de files

La définition récurrente des files que nous avons donnée aux paragraphes 1.1.1 et 1.1.4. permet d'exprimer facilement le traitement d'une file de manière récurrente. De manière générale, on pourra mener deux types d'analyse récurrente suivant que l'on intègre le cas de la file vide ou pas :

a) premier type d'analyse : on ne considère pas le cas de la file vide.
L'analyse est basée sur les deux cas suivants :

cas n°1 : la file ne comporte qu'un seul élément
E_S(PRE)=faux : c'est le cas simple de l'analyse récurrente

cas n°2 : la file comporte plus d'un élément
E_S(PRE)=vrai : l'analyse consiste à se ramener à un traitement élémentaire du premier élément et au même problème sur la sous-file des éléments restants.

b) deuxième type d'analyse : on intègre le cas de la file vide.
L'analyse est basée sur les deux cas suivants :

cas n°1 : la file est vide

EST_VIDE(E)=vrai : on doit décider pour ce cas, d'une convention qui soit compatible avec le traitement général.

cas n°2 : la file n'est pas vide

EST_VIDE(E)=faux : l'analyse consiste à se ramener à un traitement élémentaire du premier élément et au même problème portant sur la sous-file des éléments restants.

1.1.7. - Exemple : mots palindromes

Un mot est *"palindrome"* si il est égal à son image miroir (exemple : laval). On considère un mot de taille *N* donné dans un tableau de caractères *(N > 0)* :

tableau (1:N) car T

Le problème consiste à déterminer si le mot donné est palindrome ou non. L'analyse qui suit est fondée sur l'idée simple suivante : un mot est palindrome si et seulement si ses deux extrêmités sont égales et si le sous-mot obtenu en enlevant ces deux extrémités au mot initial est lui aussi palindrome.

a) Construction de l'algorithme

Nous considérons la file suivante de sous-mots du mot donné :

. un sous-mot est caractérisé par un sous-tableau du tableau donné, soit par deux positions p et d dans ce tableau, avec $p \leq d$.

. étant donné un sous-mot, on lui associe la file de sous-mots définie récursivement comme suit :

- le premier sous-mot est le sous-mot donné lui-même
- les autres sous-mots, s'il en existe, sont ceux de la file associée au sous-mot (s'il existe) obtenu à partir du sous-mot donné en lui enlevant les extrémités.

Nous considérons alors la file de sous-mots associée au mot donné par ce processus récursif. On en déduit alors la définition des primitives de description de la file :

Description de la file :

1. un sous-mot est caractérisé par deux positions p et d dans le tableau donné $(p \leq d)$. La file est vide si N = 0.

2. les primitives sont :

PRE	:	*1,N*	
SUC(p,d)	:	*p+1,d-1*	
E_S(p,d)	:	*p+1 ≤ d-1*	
INFO(p,d)	:	*T[p..d]*	{désigne le sous mot *T(p),T(p+1),...,T(d)*}
FILE_VIDE	:	*N=O*	{la file est vide si le mot donné l'est}

Analyse récurrente :

On applique le premier type d'analyse récurrente à une sous-file de cette file, désignée par le premier de ses sous-mots, soit par deux positions p et d , avec $p \leq d$.

Cas n°1 : la file ne comporte qu'un seul élément

$p+1 > d-1$: le mot correspondant est palindrome si et seulement si ses extrémités sont égales

Cas n°2 : la file comporte plus d'un élément

$p+1 \leq d-1$: le mot correspondant est palindrome si et seulement si ses extrémités sont égales et la sous-file qui en est déduit par l'application *SUC* correspond aussi à un mot palindrome.

On déduit de cette analyse l'algorithme suivant :

Algorithme : mot palindrome - forme 1

```
fonction PAL(entier p,d) → logique
 {PAL a la valeur vrai si et seulement si le sous-mot défini
  par les positions p et d du tableau T est palindrome}
 choix
   p+1 > d-1  : →T(p)=T(d)
   p+1 ≤ d-1  :  choix
                   T(p)≠T(d) : →faux
                   T(p)=T(d) : →PAL(p+1,d-1)
                 fchoix
 fchoix
ffonction PAL
```

et on teste si le mot donné est palindrome par

```
choix
  N=0  :  "mot vide"
  N≠0  :  si PAL(1,N)
             alors "mot palindrome"        {"séquentialisation" d'une
             sinon "mot non palindrome"     construction choix}
          fsi
fchoix .
```

b) Autre construction

On peut obtenir un autre algorithme en intégrant le cas de la file vide dans le cas général. On a alors :

Description de la file :

On caractérise la file vide par deux entiers p et d, tels que $p > d$, soit :

EST_VIDE(p,d) : $p > d$

PRE, *SUC* et *INFO* conservent les sens précédents.

(*INFO*(p,d) n'est pas défini si $p > d$)

Analyse récurrente :

On prend pour convention que la file vide correspond à un mot palindrome. On a alors en appliquant le deuxième type d'analyse récurrente :

Cas n°1 : la file est vide
le mot correspondant est palindrome (convention)

Cas n°2 : la file n'est pas vide
le mot correspondant est palindrome si et seulement si ses extrémités sont égales et le sous-mot suivant est lui-même palindrome.

On déduit de cette analyse, l'algorithme suivant :

Algorithme : mot palindrome - forme 2

```
fonction PAL1(entier p,d) → logique :
  {PAL1 a la valeur vrai si et seulement si le sous-mot défini
   par les positions p et d du tableau T est palindrome. On peut
   avoir p > d : on considère qu'on a alors un mot palindrome}.
  choix
     p > d   : →vrai
     p ≤ d   : choix
                 T(p)≠T(d) : → faux
                 T(p)=T(d) : → PAL1(p+1,d-1)
               fchoix
  fchoix
ffonction PAL1
```

et on teste si le mot donné est palindrome par la séquence :

```
si PAL1(1,N)
   alors "mot palindrome"
   sinon "mot non palindrome"
fsi
```

1.1.8. - Conclusion

L'exemple qui précède nous a permis d'illustrer le processus de résolution d'un problème par la méthode de traitement séquentiel :

. Le raffinement de l'information traitée en un ensemble, sa structuration en file et la description de cette file en termes de primitives qui constituent autant d'objectifs vers lesquels tend le processus de structuration.

. La construction d'un algorithme à partir d'une analyse récurrente guidée par la structure de file donnée à l'information.

Nous avons de plus montré comment ce processus de construction peut être appliqué de deux manières différentes selon que l'on considère à part ou non le cas de la file vide. Dans les paragraphes qui suivent, nous construirons les algorithmes généraux de traitement de files sur la base du premier type d'analyse récurrente (où l'on considère la file vide à part). Lors de l'application de ces algorithmes dans un contexte particulier, on pourra toujours essayer, si on le désire, le deuxième type d'analyse (dont le succès dépend de la possibilité de trouver une convention permettant d'assimiler la file vide à une file non vide).

1.2 - ALGORITHMES DE TRAITEMENTS DE FILES

Nous appliquons maintenant les résultats précédents à la construction d'algorithmes généraux de traitement de files. Ce faisant, nous poursuivons deux objectifs :

. illustrer la méthode du traitement séquentiel

. élaborer un ensemble d'algorithmes que l'on puisse utiliser directement comme des schémas, dès lors que l'on reconnaît dans un problème les conditions de leur application.

Les algorithmes que nous étudions sont répartis en deux groupes :

. les algorithmes de parcours qui ramènent le traitement de la file à l'application successive d'un traitement élémentaire à chacun des éléments de la file

. les algorithmes de recherche qui ramènent le traitement d'une file au traitement élémentaire d'un seul de ses éléments.

De plus, nous envisageons le cas où l'on cherche à traiter non pas la file donnée, mais une de ses sous-files.

Remarque :

Nous faisons abstraction dans ce qui suit, de l'origine de la file (le processus de structuration) et de sa représentation, en utilisant les notations du § 1.1.3.

1.2.1. - Parcours d'une file

Le parcours d'une file est l'application d'une même action élémentaire à chacun des éléments de la file.

Nous utilisons les notations suivantes :

VISITER(E)

désigne l'application de l'action élémentaire pour l'élément *E* de la file. Nous parlerons de la "visite" d'un élément. L'effet de la visite d'un élément est caractérisé par un ensemble d'informations indépendantes de la file caractérisant le traitement séquentiel.

Le parcours de la file est initialisé par une action

INITIALISATION_PARCOURS en abrégé *INIT-P*

et est suivi d'une action de terminaison

TERMINAISON_PARCOURS en abrégé *TERM_P*

Nous définissons le parcours d'une file en nous basant sur la définition récurrente des files. Ce faisant, nous proposons aussi un modèle pour l'analyse de problèmes de parcours de files.

Construction de l'algorithme :

Elle se base sur l'analyse récurrente suivante (rappelons que nous prenons dans ce qui suit systématiquement le premier type d'analyse récurrente cf. § 1.1.6.).

Cas n°1 : la file a un élément : on visite l'élément

Cas n°2 : la file a plus d'un élément : on visite le premier élément et on parcourt la sous-file des éléments restants.

De cette analyse, on déduit l'algorithme suivant :

Algorithme : parcours d'une file

```
action parcours (file F)
  {parcours de la file F supposée non vide}
   VISITER(F)
   si E_S(F) alors parcours(SUC(F)) fsi
faction parcours
```

le parcours de la file donnée est réalisé par l'appel :

```
INIT_P
si ¬ FILE_VIDE alors parcours(PRE) fsi
TERM_P
```

1.2.2. - Parcours d'une sous-file

Nous considérons un cas particulier du problème précédent : de la file donnée, on ne désire parcourir qu'une sous-file, dont le premier élément est celui de la file donnée et dont le dernier élément est défini à l'aide d'un prédicat donné P.

Nous distinguons deux types de parcours de sous-files :

. le parcours d'une sous-file définie par *"inclusion"* : le prédicat P caractérise le dernier élément de la file.

. le parcours d'une sous-file définie par *"exclusion"* : le prédicat P caractérise le premier élément de la file donnée qui ne fait pas partie de la sous-file.

Dans les deux cas, si aucun élément de la file donnée ne vérifie le prédicat P, toute la file est parcourue.

a) Parcours d'une sous-file définie par inclusion

Nous appliquons le traitement séquentiel de la manière suivante : la file "abstraite" caractérisant le traitement séquentiel est la sous-file étudiée. Sa définition ne diffère de celle de la file donnée que par la primitive caractérisant le dernier élément :

Description de la file :

(pour distinguer les primitives décrivant la sous-file de celle décrivant la file, nous les suffixons à l'aide du caractère ')

PRE' : *PRE*

SUC'(E) : *SUC(E)*

E_S'(E) : *E_S(E) et ¬ P(E)*

Construction de l'algorithme :

On peut reprendre pas à pas l'analyse du § 1.2.1. : en fait nous sommes dans les conditions d'application de l'algorithme de parcours d'une file. De cette identification, nous déduisons l'algorithme suivant, en remplaçant dans l'algorithme de parcours d'une file, les primitives de description de la file par celles que nous venons de donner :

Algorithme : parcours d'une sous-file définie par inclusion

```
action parcours_inc (file F)
  {parcours d'une sous-file de F définie par inclusion à
   l'aide du prédicat P. F n'est pas vide}
   VISITER(F)
   si E_S(F) et ¬ P(F) alors parcours_inc(SUC(F)) fsi
faction parcours_inc
```

et le parcours de la sous-file est réalisé par la séquence :

```
INIT_P
si ¬ FILE_VIDE alors parcours_inc(PRE) fsi
TERM_P
```

Cas particulier :

On peut considérer le cas particulier où l'on sait que la file donnée comporte au moins un élément vérifiant le prédicat P. Dans ce cas, l'action de parcours devient :

```
action parcours_inc1(file F)
  {parcours d'une sous-file non vide, définie par inclusion.
   il existe un élément vérifiant le prédicat P}
   VISITER(F) ; si ¬ P(E) alors parcours_inc1(SUC(F)) fsi
faction parcours_inc1
```

Remarque : on peut constater l'analogie entre cet algorithme et l'algorithme de parcours d'une file. Nous reviendrons sur cette analogie au § 2.3.2.

b) Parcours d'une sous-file définie par exclusion

Nous appliquons le traitement séquentiel de la manière suivante :
On se ramène au parcours d'une sous-file définie par inclusion en considérant une nouvelle visite définie comme suit :

VISITER'(F) : si ¬ P(F) alors VISITER(F) fsi

Appliquant alors l'algorithme précédent, on obtient après simplification

Algorithme : parcours d'une sous-file définie par exclusion

```
action parcours_exc (file F)
   {parcours d'une sous-file définie par exclusion à l'aide du
    prédicat P. La file F est supposée non vide}.
   si ¬ P(F) alors
      VISITER(F) ; si E_S(F) alors parcours_exc(SUC(F)) fsi
   fsi
faction parcours_exc
```

et le parcours partiel de la file donnée est réalisé par la séquence :

```
INIT_P
si ¬ FILE_VIDE alors parcours_exc(PRE) fsi
TERM_P
```

Cas particulier :

Comme précédemment, on peut considérer le cas où l'on sait que la file comporte au moins un élément vérifiant P (si elle n'est pas vide).
L'action de parcours devient alors :

```
action parcours_exc1(file F)
   {parcours d'une sous-file définie par exclusion
    il existe un élément vérifiant le prédicat P}
   si ¬ P(F) alors VISITER(F) ; parcours_exc1(SUC(F)) fsi
faction parcours_exc1
```

1.2.3. - Recherche d'un élément dans une file

Nous étudions ici le problème de la *"recherche associative"* dans une file, défini comme suit :

Etant donné une file et une propriété P, le traitement de la file se ramène à deux cas :

. il existe un élément de la file vérifiant la propriété P :
on applique l'action

TRAITER_OUI en abrégé *T_OUI*

au premier élément E de la file qui vérifie P.

. il n'existe aucun élément dans la file vérifiant la propriété P :
on applique alors l'action

TRAITER_NON en abrégé *T_NON*

Construction de l'algorithme :

L'algorithme est décomposé en deux parties :

. la recherche du premier élément de la file vérifiant P (s'il existe)

. l'application du traitement adéquat en fonction du résultat de la recherche.

L'algorithme de recherche est construit par analyse récurrente :

Cas n°1 : la file comporte un seul élément :
cet élément est l'élément cherché s'il vérifie P
sinon il y a échec dans la recherche

Cas n°2 : la file comporte plus d'un élément
si le premier élément de la file vérifie P, c'est l'élément cherché
Dans le cas contraire, on se ramène au problème de recherche associative portant sur la sous-file des éléments suivant le premier élément.

De cette analyse on déduit l'algorithme suivant :

Algorithme : recherche associative dans une file

```
fonction r_a(file F) → logique, élément
   {recherche du premier élément de F vérifiant P
    F n'est pas vide. Le couple résultat est
    vrai, élément trouvé      si l'on trouve un élément vérifiant P
    faux, indéfini            si on n'en trouve pas}
   choix
    P(F)    : → vrai , F
   ¬P(F)    : choix
                 E_S(F) : → r_a(SUC(F))
                 ¬E_S(F) : → faux,F
              fchoix
   fchoix
ffonction r_a
```

le problème posé est alors résolu par la séquence :

```
choix
  FILE_VIDE    :  T_NON
  ¬ FILE_VIDE  :  (t,O) := r_a(PRE)
                  choix
                     t   : T_OUI(O)
                    ¬t   : T_NON
                  fchoix
fchoix
```

Cas particulier :

On sait que la file contient au moins un élément vérifiant le prédicat P.

Dans ce cas la fonction de recherche se simplifie :

```
fonction r_a1(file F) → élément
  {recherche associative dans une file non vide
   on sait que l'élément cherché existe}
  choix
    P(F)      → F
    ¬ P(F)  : → r_a1(SUC(F))
  fchoix
ffonction r_a1
```

et le traitement de la file est réalisé par la séquence :

```
choix
  FILE_VIDE    :  T_NON
  ¬ FILE_VIDE  :  T_OUI(r_a1(PRE))
fchoix
```

1.2.4. - Quelques exemples d'application de la recherche associative

Nous illustrons l'application de l'algorithme précédent sur quelques exemples dont la fréquence est suffisamment élevée pour qu'ils reçoivent le statut de schéma.

a) Traitement du dernier élément d'une file

On suppose que la primitive *DER* n'est pas définie. On veut visiter le dernier élément de la file. On est dans les conditions d'application de l'algorithme précédent :

. lorsque la file n'est pas vide, le dernier élément existe toujours
. il est caractérisé par la propriété :

$$P(E) : \neg E\_S(E)$$

La nature même de cette propriété conduit à une simplification évidente de l'algorithme général. On obtient finalement l'algorithme suivant :

Algorithme : traitement du dernier élément d'une file

```
fonction rech_der (file F) → élément
  {recherche le dernier élément d'une file non vide}
  choix
    E_S(F)    : → rech_der(SUC(F))
    ¬ E_S(F)  : → F
  fchoix
ffonction rech_der
```

le traitement de la file est alors réalisé par la séquence :

```
choix
  FILE_VIDE    : T_NON
  ¬ FILE_VIDE  : T_OUI(rech_der(PRE))
fchoix
```

c) Recherche d'un élément de clé donnée

On suppose que l'information propre associée à chaque élément de la file est structurée en champs et que l'un d'entre eux joue le rôle d'indicatif de recherche. On notera :

$CLE(E)$

l'indicatif associé à l'élément E.
Le problème consiste à rechercher le premier élément de clé égale à une valeur X donnée.

Construction de l'algorithme

On applique directement l'algorithme de recherche associative en prenant pour propriété :

$P(E) \quad : \quad CLE(E)=X$

nous ne donnons pas cet algorithme ici.

d) Recherche d'un élément de clé donnée dans une file triée

On reprend le problème précédent en supposant que la file donnée est triée selon les valeurs croissantes des indicatifs. (On suppose ces indicatifs munis d'un ordre total noté à l'aide des symboles utilisés couramment sur les entiers). On recherche pour une valeur de clé X donnée, le premier couple d'éléments consécutifs dans la file, soit $E1$ et $E2$ tels que :

$CLE(E1) \leq X < CLE(E2)$

Construction de l'algorithme :

On applique le traitement séquentiel à la file des couples d'éléments consécutifs de la file.

b) Recherche par comptage

L'élément cherché est le Nième de la file pour N donné.

On applique le traitement séquentiel de la manière suivante :

Description de la file abstraite :

La file abstraite caractérisant le traitement séquentiel est définie à partir de la file donnée, en associant à chaque élément de celle-ci son rang :

1. un élément de la file abstraite est caractérisé par un élément de la file donnée et un entier.

2. les primitives sont :

```
PRE'        : PRE, 1
SUC'(E,R)   : SUC(E), R+1
E_S'(E,R)   : E_S(E)
```

Construction de l'algorithme :

On applique un schéma de recherche associative à la file abstraite que nous venons de décrire, en prenant pour propriété :

```
P(E,R)  :  R=N
```

Algorithme : recherche dans une file par comptage

```
fonction rech_compt(file F, entier R) → logique, élément
  {recherche l'élément de rang N dans la sous-file F
   F n'est pas vide. Son premier élément a pour rang R
   le couple résultat est :
   vrai, élément de rang N      si cet élément existe
   faux, indéfini               sinon}
  choix
    R=N   :  → vrai, F
    R≠N   :  choix
               E_S(F)    : → rech_compt(SUC(F),R+1)
               ¬ E_S(F)  : → faux, F
             fchoix
  fchoix
ffonction rech_compt
```

et la recherche dans la file donnée est effectuée par l'appel :

```
choix
  FILE_VIDE    :  T_NON
  ¬ FILE_VIDE  :  (t,O):=rech_compt(PRE,1)
                  choix
                    t    : T_OUI(o)
                  ¬ t    : T_NON
                  fchoix
```

Description de la file des couples

1. un élément est caractérisé par deux éléments consécutifs de la file donnée.
   Pour que la file des couples ne soit pas vide, elle doit avoir au moins deux éléments.

2. les primitives sont :

```
PRE'          :  PRE, SUC(PRE)
SUC'(E1,E2)   :  E2, SUC(E2)
E_S'(E1,E2)   :  E_S(E2)
FILE_VIDE'    :  FILE_VIDE ou_c ¬ E_S(PRE)
```

Remarque :
ou_c désigne l'opérateur ou_conditionnel (cf. chapitre 0).

Algorithme :

On applique la recherche associative à la file des couples en prenant pour propriété :

$P(E1,E2)$: $CLE(E1) \leq X < CLE(E2)$

on obtient alors après simplification

```
fonction rech_tri(élément E1,E2) → logique, élément, élément
  {recherche,dans la file des couples commençant par E1, E2,
   le premier couple "contenant" X. On sait que X ≥ CLE(E1)
   le triplet résultat a pour valeur :
   vrai, couple solution      si la recherche aboutit
   faux, dernier, indéfini    sinon}
  choix
    X < CLE(E2)  : → (vrai,E1,E2)
    X ≥ CLE(E2)  : choix
                     E_S(E2)    : → rech_tri(E2,SUC(E2))
                     ¬ E_S(E2)  : → (faux,E2,E2)
                   fchoix
  fchoix
ffonction rech_tri
```

et la recherche s'effectue par la séquence :

```
choix
  FILE_VIDE    : T_NON
  ¬ FILE_VIDE  : si X < CLE(PRE) ou ¬ E_S(PRE)
                 alors T_NON
                 sinon (t,O1,O2):=rech-tri(PRE,SUC(PRE))
                       choix
                         t    : T_OUI(O1,O2)
                         ¬ t  : T_NON
                       fchoix
fsi
```

Remarque :

- l'exemple du § 1.1.7 peut être traité en appliquant un schéma de recherche : on recherche le premier sous-mot, s'il existe, dont les extrémités sont différentes.
- Nous étudierons dans la deuxième partie la réduction logarithmique de cet algorithme (cf. § 5.2.3.).

I.3 - CONCLUSION

Résoudre un problème en appliquant le traitement séquentiel consiste à :

1. Structurer l'information en file, après l'avoir raffinée en un ensemble d'informations plus élémentaires.

2. Guider la construction d'un algorithme par cette interprétation de l'information :

   - soit en appliquant un modèle général d'analyse de traitement de file,
   - soit en identifiant un problème général de parcours de file ou de recherche dans une file, et en appliquant le schéma correspondant.

Nous avons proposé des outils permettant d'appliquer ce principe de construction d'algorithmes de manière systématique :

- un modèle de processus de structuration en file
- des notations pour la description d'une file
- un modèle d'analyse récurrente des traitements de file
- des algorithmes généraux de parcours de files et de recherche dans une file.

Nous avons placé notre présentation dans un contexte algorithmique, utilisant notamment des formalismes récursifs correspondant à ce qui est disponible dans les langages de la famille ALGOL. Il est clair que le traitement séquentiel n'est pas lié à ce style de formalisme et peut être envisagé dans des environnements techniques ou culturels différents.
Par exemple :

- un langage tel que LUCID (AsW 77) nous semble parfaitement adapté à ce qui a été présenté, offrant de plus l'axiomatique nécessaire pour effecteur la preuve des algorithmes.

- le contexte de l'analyse syntaxique offre des formalismes (description de grammaires) des techniques (méthodes d'analyse, insertion d'actions sémantiques) et des outils (générateurs d'analyseurs) que l'on peut parfaitement utiliser pour la résolution des problèmes que nous avons présentés (GoH 74).

Nous suggérons que quel que soit l'environnement choisi, les principes du traitement séquentiel que nous venons de résumer et le souci constant d'une démarche systématique sont des éléments méthodologiques suffisants pour permettre de résoudre les problèmes considérés de manière méthodique et fiable.

CHAPITRE 2

PROGRAMMATION DU TRAITEMENT SEQUENTIEL

Nous poursuivons la présentation de la méthode de traitement séquentiel, en étudiant la programmation des algorithmes du § 1.2 dans un langage de programmation conventionnel (dont nous retiendrons pour caractéristique essentielle le fait d'être fondé sur la notion de variable et sur le recours à l'instruction d'affectation).

Nous voulons ainsi :

- continuer à décrire le processus de construction de programmes de traitement séquentiel, en caractérisant les étapes liées au niveau de programmation choisi.

- confirmer le caractère pratique de la méthode en élaborant un ensemble de schémas dans un environnement de programmation standard.

Pour passer du niveau algorithmique présenté au chapitre 1 au niveau de programmation choisi, le processus de construction comporte trois phases supplémentaires :

1. L'élaboration d'un répertoire d'actions élémentaires permettant de décrire le processus dynamique de progression le long d'une file. Pour rendre cette phase systématique, nous utilisons un modèle de "machine séquentielle", qui s'inspire notamment des répertoires classiques d'instructions de lecture de fichiers séquentiels (cf. par exemple [Wir 76]). De même que les notations du § 1.1.3., ce modèle de machine

séquentielle, nous permet par ailleurs de décrire les schémas de traitement séquentiel en faisant abstraction de la représentation de la file caractérisant le traitement.

2. La construction d'un algorithme itératif à la suite d'une analyse récurrente telle que nous l'avons présentée au § 1.1.6. Cette construction peut s'appuyer sur l'une ou l'autre des méthodes générales existantes que nous avons citées dans l'introduction de cette première partie. On peut aussi se baser sur des techniques générales de transformation de programmes récursifs en programmes itératifs (voir les références données au début du chapitre 4 et le chapitre 4 lui-même). Enfin, nous suggérons d'utiliser directement les schémas présentés dans ce chapitre, dès lors que les conditions de leur application sont reconnues.

3. La simplification du programme obtenu par des transformations simples, prenant en compte les particularités du problème étudié par rapport au schéma général de référence. Nous étudions certains de ces cas particuliers en détail tant pour illustrer cette phase de simplification, que pour fournir des schémas (ce qui est justifié par la fréquence ce ces cas).

2.1 - ACCES SEQUENTIEL A LA FILE : deux modèles de *"machines séquentielles"*

2.1.1. - Machines séquentielles

Une *"machine séquentielle"* est un répertoire d'actions élémentaires permettant de décrire le processus dynamique de progression le long d'une file (nous disons l'*"accès séquentiel"* à une file).
La description d'une machine séquentielle comporte :

1. La description des informations caractérisant l'état de la machine.

A tout moment de l'accès séquentiel à la file, on doit pouvoir connaître les informations suivantes :

. l'élément de la file sur lequel on est *"placé"* au moment considéré. Nous l'appelons l'*"élément courant"*.

. l'indication de la détection de la fin de la file. Nous parlons d'un *"voyant"* (de fin de file) qui est *"éteint"* (si l'on n'a pas encore atteint la fin de la file) ou *"allumé"* (si on a atteint la fin de la file).

2. La description des primitives d'accès séquentiel qui sont réparties en deux groupes :

. des primitives de consultation qui permettent d'examiner l'état de la machine, sans le modifier : quel est l'élément courant ? le voyant est-il allumé ?

. des primitives de modification de l'état de la machine : une action d'avancement le long de la file, une action de démarrage du processus de progression.

La manière de repérer le dernier élément d'une file joue un rôle important dans la mise en place des algorithmes de traitement séquentiel : un tel algorithme est toujours contrôlé par la progression le long d'une file, et la fin de l'algorithme correspond toujours à l'accès au dernier élément dans le cas d'un parcours, à l'élément cherché (ou au dernier élément) dans le cas d'une recherche.

Nous distinguons deux modèles de machine séquentielle selon la manière de spécifier l'arrivée sur la fin de la file (c'est-à-dire la relation entre l'action d'avancement et l'éclairage du voyant) :

1er cas : le voyant s'allume à l'instant du départ du dernier élément de la file.

Lorsque le voyant est allumé, on vient de quitter le dernier élément par une action d'avancement, et en toute généralité, l'élément courant est alors indéfini.

Remarque :

Ceci s'inspire directement de la spécification standard des instructions de lecture de fichiers séquentiels. La notion de "voyant allumé" correspond à la réception d'une interruption de fin de fichier.

Ce type de spécification est notamment utilisé :

. lorsque la file est décrite en intégrant le cas de la file vide (cf. 1.1.4. b).

. lorsque la file considérée est une sous-file définie par exclusion (cf. § 1.2.2.).

deuxième_cas : le voyant s'allume à l'instant de l'arrivée sur le dernier élément.

Lorsque le voyant est allumé, l'élément courant est effectivement le dernier de la file.

Remarque :

Ceci correspond au cas où le dernier élément de la file est doté d'une propriété caractéristique (par exemple lorsque l'on connaît le nombre d'éléments de la file).

Ce type de spécification est notamment utilisé :

. lorsque la file est décrite en distinguant le cas de la file vide (cf. 1.1.4.a).

. lorsque la file considérée est une sous-file définie par inclusion (cf. § 1.2.2.).

2.1.2. - Description_générale_des_deux_modèles_de_machine_séquentielle

Nous spécifions ici chacun des deux modèles de machine séquentielle : la spécification des primitives est donnée sous forme de pré-conditions et post-conditions exprimées en termes des informations caractérisant l'état de la machine (ce mode de spécification nous paraît bien adapté au niveau de programmation auquel nous nous plaçons et au pragmatisme que nous recherchons). La file dont on veut spécifier l'accès séquentiel est décrite à l'aide des notations des paragraphes 1.1.3. et 1.1.4.

a) premier modèle de machine séquentielle : le voyant s'allume en quittant le dernier élément.

1. Description de l'état :
C'est ici qu'on décrit la représentation de la file. Elle doit être compatible avec le fonctionnement de la machine impliqué par ce modèle de spécifications.

2. Description du répertoire :

. consultation de l'élément courant :

ELEMENT_COURANT en abrégé *EC*

désigne l'élément courant

. consultation du voyant :

FIN_DE_FILE_ATTEINTE en abrégé *FDF*

indique si le voyant est éteint *(FDF=faux)* ou allumé *(FDF=vrai)*.

. avancement le long de la file

AVANCER en abrégé *AV*

permet de donner accès au prochain élément de la file et provoque éventuellement l'éclairage du voyant. Ceci est spécifié comme suit : (e_o est une constante désignant la valeur de EC à l'appel de *AVANCER)*

état_initial :

FDF=faux et EC=e_o

état_final :

. $E\_S(e_o)$: *FDF=faux et EC=SUC(*e_o*)*
. $\neg E\_S(e_o)$: *FDF=vrai et EC=* valeur non définie

Remarque :
AVANCER ne peut être utilisé lorsque le voyant est allumé.

. démarrage :

DEMARRER en abrégé *DEM*

désigne le démarrage du processus d'accès séquentiel : *DEMARRER* donne accès au premier élément de la file (s'il existe). Son effet est spécifié comme suit :

état initial :

quelconque (la file doit être effectivement accessible)

état final :

FILE_VIDE : *FDF=vrai et EC=* valeur non définie

$\neg$*FILE_VIDE* : *FDF=faux et EC=PRE*

Remarque :
Nous avons spécifié cette action de manière à pouvoir intégrer le cas des files vides : lorsque la file est vide, DEMARRER initialise effectivement EC mais avec une valeur non définie (on ne peut présumer de cette valeur au niveau de l'utilisation de la machine séquentielle).

b) Deuxième modèle de machine séquentielle : le voyant s'allume en arrivant sur le dernier élément.

1. Description de l'état

Description de la représentation en accord avec le modèle.

2. Description du répertoire :

. primitives de consultation : cf. premier modèle

. avancement le long de la file :
La spécification de *AVANCER* est maintenant :

état initial :

$FDF=faux \text{ et } EC=e_o$

état final :

$(FDF= \neg E\_S(SUC(e_o))) \text{ et } EC=SUC(e_o)$

. démarrage :

état initial :

quelconque, file non vide

état final :

FDF= ¬E_S(PRE) et EC=PRE

Remarque :

On a la relation invariante : *FDF= ¬E_S(EC)*

2.1.3. - Exemple de description d'un accès séquentiel

Nous reprenons l'un des exemples du § 1.1.5. : la file considérée est la file des couples de caractères consécutifs dans un tableau de *N* caractères *(N > 0)*.. Nous avons décrit cette file au § 1.1.5. et nous pouvons maintenant appliquer les modèles précédents pour en décrire l'accès séquentiel.

Remarque :
Nous montrons dans cet exemple que les deux modèles peuvent être appliqués. Toutefois, nous ne choisissons pas l'une des solutions : ce choix ne peut être fait que dans un contexte plus large notamment en fonction des traitements effectués.

a) Appliquant le modèle 1, on obtient :

1. description de l'état :
La file de couples de caractères est représentée à l'aide du tableau de caractères qui apparaît dans l'énoncé même de l'exemple :

tableau (1:N) car T {*N > 0*
le dernier caractère est le premier point rencontré}

. l'application du modèle 1 implique que le voyant doit s'allumer lorsque l'on quitte le dernier couple (i.e. celui dont le deuxième élément est le point final) : ceci est réalisé simplement en considérant que le point est le début d'un couple fictif.

. Le couple courant est caractérisé par une position dans le tableau T :

entier poscour {position courante définit le couple courant $1 \leq poscour \leq N$}

2. Description du répertoire

EC : *T(poscour), T(poscour+1)* {non défini si *FDF=vrai*}
FDF : *T(poscour)=point*
AV : *poscour:=poscour+1*
DEM : *poscour:=1*

Remarque : On peut constater l'intégration du cas de la file vide.

b) Appliquant le modèle 2, on obtient :

1. Description de l'état :
Le seul changement par rapport à ce qui précède est le fait que le voyant s'allume lorsque l'on atteint le dernier couple, i.e. lorsque le deuxième caractère du couple courant est le point final
On a maintenant : $1 \leq poscour < N$
2. Description du répertoire :

EC : *T(poscour), T(poscour+1)*
FDF : *T(poscour+1)=point*
AV : *poscour:=poscour+1*
DEM : *poscour:=1*

Remarque :
Le cas de la file vide est pris en compte en testant si le premier caractère du tableau est un point.

FILE_VIDE : T(1) = point

2.1.4. - Construction d'une machine séquentielle

Cet exemple très simple n'avait pour objet que d'illustrer les différences entre les deux modèles proposés. Nous montrerons sur des exemples plus complets (notamment dans le chapitre 3) les éléments qui conduisent au choix d'un des deux modèles, et préciserons alors les aspects systématiques de la construction d'une machine séquentielle.

Une telle construction comporte deux étapes importantes :

1. Le choix d'un modèle de machine en fonction du contexte .
   Les éléments qui peuvent aider à ce choix sont par exemple :
   - une convention de fin de file particulière imposée par le problème
   - des critères de concision ou de coût des programmes écrits.

2. La construction proprement dite de la machine.
   Cette construction comporte dans l'ordre :
   - la description de l'état de la machine qui se résume par un ensemble de déclarations de variables et éventuellement par l'énoncé de relations existant entre ces variables (notamment énoncé de propriétés, de relations invariantes).
   - la réalisation des primitives du répertoire :

     . les primitives de consultation ne posent pas de problème particulier : on accède aux variables décrivant l'état de la machine.

     Remarque :
     On peut cependant apporter un soin particulier à la "protection" de cet état vis à vis de l'utilisateur de la machine : ne donner accès qu'aux variables donnant l'élément courant et l'état du voyant ; n'autoriser l'utilisation de ces variables qu'en lecture seulement.
     La réalisation de cette protection dépend évidemment du langage de programmation utilisé mais aussi de la complexité globale du programme en cours de construction (taille du programme, nombre de programmeurs y participant...).

     . les primitives de modification sont réalisées à partir des spécifications du modèle choisi et en fonction de la représentation choisie. En général il est préférable d'étudier l'avancement avant le démarrage.

     Remarque :
     En effet, il est fréquent que l'étude du démarrage soit une simple adaptation de l'étude de l'avancement au cas particulier où l'on cherche à acquérir le premier élément. Dans un certain nombre de cas ceci se traduit au niveau même des programmes, le démarrage étant réalisé en termes de l'avancement (nous étudions ce cas particulier au § 2.3.1).

2.1.5. - Conclusion

Les machines séquentielles constituent un guide pour réaliser le passage du niveau "abstrait" du chapitre 1 (on faisait abstraction du processus dynamique de progression le long de la file et de la représentation de la file ; la description de la file consistait à donner des relations entre ses constituants) au niveau "concret" de ce chapitre (on se place dans le contexte d'un environnement de programmation précis).

Nous avons tenu à donner une certaine généralité à notre présentation en choisissant des notations qui laissent une grande liberté de réalisation, notamment en fonction du langage utilisé :

. une machine séquentielle peut être décrite à l'aide d'une construction inspirée du concept de type abstrait (LiZ 74), et notamment à l'aide du mécanisme d'*"univers"* que nous présentons dans la troisième partie (cf. aussi [SGC 78]). On peut ainsi à la fois exprimer le caractère modélisant de la notion de machine séquentielle et assurer la protection de ce composant du programme vis à vis des autres composants.

. lorsque l'on ne dispose pas d'une telle construction, on peut associer une procédure à chacune des primitives du répertoire et refléter la notion de machine séquentielle à l'aide d'une mise en page et de commentaires adéquats.

. enfin, on peut se passer de toute construction associée aux abstractions manipulées, et simplement substituer dans les algorithmes de traitement de la file, les primitives d'accès séquentiel par leur définition (éventuellement en leur associant un commentaire rappelant leur rôle logique).

Remarque :
Soulignons que dans le cas où le langage utilisé fournit les constructions les plus élaborées, chacune des trois solutions envisagées ci-dessus, et toute solution intermédiaire, peut être appliquée, selon les critères qui régissent la construction du programme, les conditions de cette construction, et les conditions d'utilisation du programme.

2.2 - SCHEMAS DE TRAITEMENT SEQUENTIEL

Nous poursuivons maintenant la construction des algorithmes que nous avons présentés au § 1.2., en appliquant les principes de construction que nous venons d'étudier. Comme précédemment notre objectif est double :

. illustrer la méthode du traitement séquentiel
. élaborer un ensemble d'algorithmes que l'on puisse utiliser comme des schémas.

2.2.1. - Principes de construction et d'évaluation

a) Construction

Les schémas sont construits en partant des algorithmes du chapitre 1, et en examinant les deux types de machines séquentielles. Du point de vue des notations, nous utilisons des primitives d'itération qui sont présentées au chapitre 0 et discutées dans la troisième partie. De manière générale, nous cherchons à obtenir les schémas les plus concis possibles et notamment, nous voulons faire apparaître chaque primitive de la machine séquentielle une fois et une seule dans le schéma (essayant ainsi de concilier, sans nuire à notre démarche systématique, des critères de lisibilité, de portabilité, et éventuellement d'efficacité).

b) Evaluations

Chaque schéma est "évalué" de la manière suivante (cf. chapitre 0) : on associe à chaque instruction du schéma une expression à valeur entière (placée sur la droite en regard de l'instruction) qui indique le nombre de ses exécutions : cette expression est décrite en termes des paramètres suivants :

N	nombre d'éléments de la file
NS	nombre d'éléments de la sous-file considérée (le cas échéant - cf. § 2.2.2. et 2.2.3).
ε	1 si $N=0$ ou 0 si $N\neq 0$ (dans le cas où l'on traite à part le cas de la file vide).

Remarque :
Dans le cadre du traitement séquentiel, les calculs conduisant aux expressions associées à chaque instruction sont en général très simples. C'est pourquoi, nous ne donnons pas d'explication sur ces calculs.

Chaque schéma est suivi d'une rubrique "évaluation" qui résume le travail d'évaluation en cumulant certains chiffres :

1. On cumule le nombre d'exécutions de chaque primitive non définie dans le schéma (les paramètres du schéma) :

   . les primitives de la machine séquentielle

   . les primitives de traitement de la file.

   Ces chiffres représentent en quelque sorte le coût de l'algorithme (ce sont eux qui reflètent la "complexité" de l'algorithme, cf. par exemple [Knu 69], [AHU 74] ou [Baa 78]).

2. On cumule le nombre d'exécutions de toutes les autres instructions apparaissant dans le schéma :

   . "contrôle" : nous regroupons sous cette rubrique toutes les instructions de composition (choix, itération).

   . "opérations logiques" : nous ne tenons pas compte des opérations de complémentation($\neg$).

   . "opérations de comparaisons"

   . "opérations arithmétiques"

   . "variables"

   . "affectations"

Remarque :
Pour pouvoir réaliser ces cumuls, nous choisissons un modèle d'évaluation du coût de chaque instruction, basé sur la programmation des primitives dans un langage de bas niveau (langage machine) disposant de plusieurs registres (cf. chapitre 0).

Ces chiffres représentent le coût de l'algorithme dans le contexte du langage de programmation utilisé et du modèle d'évaluation des instructions choisi.

c) Rôle des évaluations

Les évaluations jouent troisrôles dans le processus de construction de programmes :

. elles permettent une certaine vérification du programme par un examen de la cohérence des chiffres obtenus.

Par exemple, lors d'un parcours de file accessible par une machine de type 1, le nombre de visites est égal au nombre d'avancements (cf. § 2.2.2.). Ce rôle apparaît plus nettement lorsque nous traitons des exemples de niveau de complexité supérieur à celui des schémas de base (cf. chapitre 3) et lorsque nous étudions les schémas de traitement d'arbres (cf. chapitre 6).

. elles constituent une "mesure du comportement" d'un algorithme : les chiffre indiquent d'où provient le coût de l'algorithme.

On peut ainsi, d'une part repérer les régions du programme les plus "coûteus (et sur lesquelles il convient de porter, le cas échéant, l'effort d'optimisation), d'autre part évaluer l'influence du style de réalisation (type de langage, technique de contrôle,...) sur le coût global .

. elles permettent de comparer pour un langage donné plusieurs solutions à un même problème.

Cette comparaison n'a pas forcément pour but de trouver la meilleure solution (en chiffres absolus qui sont toujours difficiles à calculer). Elle peut aussi bien permettre de se rendre compte que deux solutions sont de coûts équivalents ce qui laisse la possibilité de choisir en fonction d'autr critères.

2.2.2. - Parcours d'une file

Le parcours d'une file consiste à appliquer successivement à tous les éléments de la file, une action élémentaire *VISITER* . Il est initialisé par l'action *INIT_P* et est suivi de l'action *TERM_P* (cf. § 1.2.1.). Ces trois actions expriment la "sémantique" du traitement de la file. Leur effet est spécifié par rapport à un ensemble de variables dont le rôle est de traduire à tout instant du parcours de la file, l'effet cumulé des applications successives de VISITER. Nous utilisons le terme *"accumulateur"* pour désigner cet ensemble de variables.

Remarque :
VISITER ne peut pas modifier l'état de la machine séquentielle.

a) Construction des schémas de parcours

Nous avons deux schémas possibles selon que l'on choisit le premier ou le second modèle de machine séquentielle.

Dans les deux cas, la construction des itérations peut se faire à partir de l'invariant suivant (déduit directement des résultats du chapitre 1):

> Assertion invariante lors de l'appel de *VISITER* (avant l'appel) : "tous les éléments de la file précédant *EC* ont été visités. L'accumulateur traduit l'effet de ces visites antérieures. Si *EC* est le premier élément de la file, on vient d'appliquer *INIT_P*. *EC* est un élément qui doit effectivement être visité et qui ne l'a pas encore été. Enfin *EC* est l'élément qui suit le dernier élément que l'on a visité."

L'évaluation est réalisée en supposant l'algorithme correct : le nombre de visites est N. A partir de cette hypothèse, on déduit tous les autres chiffres.

Remarque :

> Les algorithmes qui suivent peuvent aussi être déduit des algorithmes du § 1.2.1. en appliquant de simples techniques de transformation "récursif-itératif".

b) Algorithmes: parcours d'une file

Algorithme P1 machine séquentielle de type 1		Algorithme P2 machine séquentielle de type 2	
INIT_P	1	*INIT_P*	1
DEM	1	*si ¬FILE_VIDE alors*	1
tantque ¬FDF faire	N+1	*DEM*	1-ε
VISITER(EC)	N	*itérer*	
AV	N	*VISITER(EC)*	N
ftantque		*arrêt : FDF*	N
TERM_P	1	*AV*	N-1+ε
		fitérer	
		fsi	
		TERM_P	1
Evaluation :		Evaluation :	
. primitives de traitement : *VISITER*:N , *INIT_P*:1 , *TERM_P*:1		. primitives de traitement : *VISITER*:N, *INIT_P*:1, *TERM_P*:1	
. primitives d'accès séquentiel : *DEM*:1, *AV*:N, *FDF*:N+1		. primitives d'accès séquentiel : *DEM*:1-ε, *AV*:N-1+ε, *FDF*:N *FILE_VIDE*:1	
. contrôle : N+1		. contrôle : N+1	

Remarque :

. Rappelons que le premier modèle de machine séquentielle tient compte du cas de la file vide.

. Si l'on sait que la file comporte au moins un élément, on peut utiliser une construction *répéter* à la place du *tantque* dans l'algorithme P1, et supprimer le test *FILE_VIDE* dans l'algorithme P2.

c) Utilisation des schémas :

A la suite du travail de construction décrit au chapitre 1, la programmation d'un parcours de file comporte le choix d'un schéma en fonction du choix d'un modèle d'accès séquentiel. Le programme obtenu peut être vérifié en examinant le nombre d'utilisation de chaque primitive, puis "simplifié" (rendu plus concis, amélioré,...) en tenant compte de conditions particulières (par exemple le fait de savoir que la file n'est pas vide, que nous avons souligné dans la remarque précédente). Nous examinerons certains de ces cas particuliers au § 2.3.

d) Conclusion

Les schémas de parcours de files résument un processus d'analyse dans lequel le problème envisagé a été réduit en termes d'une file et d'une action élémentaire de traitement. Ceci se traduit au niveau du programme par :

. La mise en évidence de deux groupes de primitives :

- les unes correspondent à l'accès séquentiel à la file
- les autres correspondent au traitement de cette file.

Cette répartition en deux groupes permet de dissocier au niveau du schéma un aspect "syntaxique" du problème (lié à la structure de file) et un aspect "sémantique" du problème (on retrouve une analogie avec la compilation, déjà soulignée en conclusion du chapitre 1).

. Le recours à des constructions linguistiques exprimant le mécanisme d'itération. Nous avons utilisé ici les constructions classiques *tantque, répéter, itérer*. Il est clair qu'il est facile de se placer à un niveau de programmation moins évolué où de telles primitives ne sont

pas disponibles (par une simple traduction en termes de tests et de branchements).

Remarque :

Nous restreignons l'utilisation de la construction *pour* au cas particulier où tout l'accès séquentiel est exprimé par le *pour* un élément de la file est caractérisé par un entier, la "valeur initiale" du *pour* définit le premier élément, la "valeur finale" définit le dernier élément, et le pas de progression définit l'avancement.
Dans ce cas, le schéma de parcours devient :

```
{i est l'entier caractérisant la file
 (variable de contrôle du pour)}

INIT_P
pour i:=valeur_initiale pas p jusquà valeur_finale faire
  VISITER(i)
fpour
TERM_P
```

Ce type de schéma nous paraît intéressant dans la mesure où toute la description de la file apparaît dans la construction d'itération. Nous pensons que c'est dans cet esprit qu'il faut rechercher des constructions linguistiques exprimant les itérations. Ceci apparaît déjà dans certains travaux (par exemple [Pai 77b],[LSA 77], [SWL 77]) [BoL 77]). Nous reprendrons cette discussion dans la troisième partie (§ 9.2.3.).

2.2.3. - Parcours d'une sous-file

L'exemple du parcours d'une sous-file nous permet comme dans le chapitre1 (cf. § 1.2.2.) de présenter une première illustration de l'application des algorithmes de parcours. Par ailleurs les schémas obtenus sont très fréquemment rencontrés (traitement de textes, traitement de fichiers séquentiels et notamment problèmes de ruptures).

Rappelons que la sous-file considérée est définie comme suit :

. son premier élément est le premier élément de la file

. son dernier élément est défini à l'aide d'un prédicat P, par inclusion ou par exclusion (cf. 1.2.2.). Si aucun élément de la file ne vérifie P, toute la file est parcourue.

. deux éléments consécutifs dans la sous-file, le sont dans la file. (nous avons décrit une telle sous-file au § 1.2.2.).

Le traitement qui doit être effectué est exprimé de la même manière que dans le cas des parcours de files.

Dans ce qui suit, nous traitons d'abord le cas des sous-files définies par exclusion, puis le cas des sous-files définies par inclusion. Pour chacun de ces cas, on construit deux schémas selon que la file donnée est accessible par une machine séquentielle de type 1 ou 2.

1. Sous-file définie par exclusion

Le dernier élément de la sous-file est soit le premier élément de la file (si aucun élément de la file ne vérifie P), soit l'élément précédant le premier élément de la file vérifiant P.

1. a) Construction de l'algorithme :

Nous appliquons le traitement séquentiel et cherchons à utiliser les schémas de parcours de files du § 2.2.2. La file abstraite caractérisant le traitement séquentiel est la sous-file.
Nous examinons la question du choix du modèle de machine séquentielle permettant de décrire l'accès séquentiel à la sous-file :

Cas n°1 :
La file donnée est accessible par une machine séquentielle de type 1 : il est clair que le dernier élément de la sous-file est connu lorsqu'on l'a quitté : l'algorithme PEX1 est construit en appliquant l'algorithme P1, avec la machine séquentielle suivante décrivant l'accès à la sous-file (machine de type 1) :

- état : il est déduit de la représentation de la file donnée
- répertoire (les primitives d'accès à la sous-file ont un nom suffixé par le caractère ');

AVANCER' : *AVANCER*
DEMARRER' : *DEMARRER*
EC' : *EC*
FDF' : *FDF ou_c P(EC')* (ou_c est l'opérateur ou_conditionnel - cf. chap. 0

Cas n°2 :

La file donnée est accessible par une machine séquentielle de type 2 :
Nous considérons que la sous-file est composée de deux parties :

- tous les éléments de la file jusqu'au premier élément (non compris) vérifiant *P(EC) ou FDF*.

- un élément supplémentaire lorsque la file ne comporte aucun élément vérifiant *P* : le dernier élément de la file (caractérisé par *FDF=vrai et P(EC)=faux*).

Ces deux parties sont traitées en séquence :

. On applique le traitement séquentiel à la première partie : il s'agit d'une sous-file dont l'accès séquentiel doit être décrit à l'aide d'une machine de type 1. Le répertoire est le suivant : (rappelons que maintenant *FDF=vrai* signifie que *EC* est le dernier de la file donnée) :

AVANCER'	:	*AVANCER*	
DEMARRER'	:	*DEMARRER*	{non défini si la file donnée est vide}
EC'	:	*EC*	
FDF'	:	*FDF ou P(EC')*	

en substituant ce répertoire aux primitives d'accès séquentiel dans l'algorithme P1, on obtient le traitement de la première partie de la sous-file (lorsqu'elle n'est pas vide)

. on traite le dernier élément s'il existe.

1. b) Algorithmes .

parcours d'une sous-file définie par exclusion

algorithme PEX1 file donnée accessible par une machine de type 1		Algorithme PEX2 file donnée accessible par une machine de type 2	
INIT_P	1	*INIT_P*	1
DEM	1	*si ¬FILE_VIDE alors*	1
tantque ¬FDF et_c ¬P(EC) faire	NS+1	*DEM*	1-ε
VISITER(EC)	NS	*tantque ¬FDF et ¬P(EC) faire*	NS+1-ε
AV	NS	*VISITER(EC)*	NS-ε1
ftantque		*AV*	NS-ε1
TERM_P	1	*ftantque*	
		si ¬P(EC) alors	1-ε
		VISITER(EC)	ε1
		fsi	
		fsi	
		TERM_P	1
Evaluation {ε1=1 si aucun élément de la file ne vérifie P, ε1=0 sinon} . accès séquentiel *DEM*:1, *AV*:NS, *FDF*:NS+1, *P*:NS+1-ε1 . traitement *INIT_P*:1, *TERM_P*:1, *VISITER*:NS . contrôle : NS+1 . opérations logiques:NS+1-ε1		Evaluation {ε1=1 si aucun élément de la file ne vérifie P, ε1=0 sinon} . accès séquentiel *DEM*:1-ε, *AV*:NS-ε1, *FDF*:NS+1-ε1-ε *FILE_VIDE*:1, *P*:NS+2-2ε-ε1 . traitement *INIT_P*:1, *TERM_P*:1, *VISITER*:NS . contrôle:NS+3-2ε-ε1 . opérations logiques:NS+1-ε1-ε	

Remarques:

. Dans l'algorithme PEX1, on peut remplacer l'opérateur *et_c* apparaissant dans le *tantque* par un opérateur *et* si l'on s'assure que lorsque le voyant est allumé, *EC* a effectivement une valeur (quelle qu'elle soit). Ceci est donc une manière de réaliser le *et_conditionnel*. Une autre manière consiste à utiliser une variable logique pour contrôler le *tantque*, cette variable étant mise à jour à chaque fois que l'on modifie *EC* .

. L'algorithme PEX2 implique que le prédicat *P* est testé deux fois pour le dernier élément de la sous-file. Si on veut éviter ceci, on peut utiliser une variable logique mémorisant le résultat du test.

1.c) Variante de l'algorithme PEX2

Nous proposons ici une variante de l'algorithme PEX2 qui correspond à un schéma utilisé fréquemment. La construction de l'algorithme est une nouvelle illustration de l'utilisation des machines séquentielles. La sous-file étant définie par exclusion, on ne peut savoir qu'un élément est le dernier de la sous-file qu'en le "quittant" : pour décrire l'accès séquentiel à la sous-file, nous nous basons sur le premier modèle de machine séquentielle.

- état : par rapport à la représentation de la file donnée, on ajoute une variable logique qui prend la valeur vrai lorsque le voyant s'allume :

```
logique fini
```

- répertoire :

```
EC'        : EC
FDF'       : fini=vrai
AVANCER'   : si FDF
                alors fini:=vrai
                sinon AVANCER ; fini:=P(EC)
             fsi
DEMARRER'  : DEMARRER ; fini:=P(EC)
             {non défini si la file donnée est vide}
```

On applique alors l'algorithme P1 et on obtient :

Algorithme :

```
{parcours d'une sous-file définie par exclusion
 file donnée accessible par une machine de type 2}
INIT_P                                          1
si ¬FILE_VIDE alors                             1
   DEM ; fini:=P(EC)        {DEMARRER'}         1-ε
   tantque ¬fini faire      {FDF'}              NS+1-ε
     VISITER(EC)                                NS
     si FDF                 {AVANCER'}          NS
        alors fini:=vrai                        ε1
        sinon AV ; fini:=P(EC)                  NS-ε1
     fsi
   ftantque
fsi
TERM_P                                          1
```

Evaluation : (ε1 a le même sens que dans PEX2) :
accès séquentiel :
DEM:1-ε, AV:NS-ε1, FDF:NS, FILE_VIDE:1, P:NS+1-ε-ε1
traitement :
INIT_P:1, TERM_P:1, VISITER:NS
contrôle : 2NS+2-ε
affectations : NS+1-ε.

2. Sous-file définie par inclusion

Le dernier élément de la sous-file est soit le dernier élément de la file, soit le premier élément de la file vérifiant le prédicat P .

2.a) Construction de l'algorithme

Comme ci-dessus, nous appliquons le traitement séquentiel à la sous-file. Nous distinguons les deux cas issus du choix de machine séquentielle pour la file donnée.

Cas n°1 :
La file donnée est accessible par une machine séquentielle de type 1. On se ramène au cas d'une sous-file définie par exclusion (algorithme PEX1) et on traite à part l'élément vérifiant la propriété P (s'il existe). On obtient ainsi l'algorithme PIN1.

Cas n°2 :
La file donnée est accessible par une machine séquentielle de type 2. On applique le traitement séquentiel à la sous-file : le dernier élément de la sous-file est connu dès que l'on est placé dessus. On doit donc utiliser le deuxième modèle de machine séquentielle pour décrire l'accès à cette sous-file :

état : aucune différence par rapport à la représentation de la file donnée.

répertoire :

EC'	:	*EC*
FDF'	:	*FDF ou P(EC)*
AVANCER'	:	*AVANCER*
DEMARRER'	:	*DEMARRER* (non défini si la file est vide).

On obtient l'algorithme PIN2, en substituant ce répertoire aux primitives d'accès séquentiel dans l'algorithme de parcours d'une file P2 (§ 2.2.2.).

Description de la file

1. un élément de la nouvelle file comporte un élément de la file donnée et un entier qui correspond au rang dans la file donnée.

2. les primitives sont :
   (E désigne un élément de la file donnée, R son rang).

$$
\begin{array}{lcl}
PREMIER' & : & PREMIER,\ 1 \\
SUC'(E,R) & : & SUC(E),\ R+1 \\
E\_S'(E,R) & : & E\_S(E) \\
PRED'(E,R) & : & PRED(E),\ R-1 \\
FILE\_VIDE' & : & FILE\_VIDE
\end{array}
$$

Traitement de la file

L'algorithme que l'on veut construire est un cas de parcours d'une sous-file de cette nouvelle file : la propriété qui définit la sous-file peut être exprimée de deux manières :

- par inclusion : $P(E,R)\ :\ R=N$
- par exclusion : $P(E,R)\ :\ R=N+1$

L'examen des schémas qui précèdent peut alors conduire à choisir

- une définition par "exclusion" lorsque la file donnée est accessible par une machine de type 1.
- une définition par "inclusion" lorsque la file est accessible par une machine de type 2.

(sur un critère de concision de la forme des algorithmes, les coûts étant équivalents).

2.b) Algorithmes

parcours d'une sous-file définie par inclusion

Algorithme PIN1 File donnée accessible par une machine de type 1		Algorithme PIN2 file donnée accessible par une machine de type 2	
INIT_P	1	*INIT_P*	1
DEM	1	*si ¬FILE_VIDE alors*	1
tantque ¬FDF et_c ¬P(EC) faire	NS+ε1	*DEM*	1-ε
VISITER(EC)	NS+ε1-1	*itérer*	
AV	NS+ε1-1	*VISITER(EC)*	NS
ftantque		*arrêt : FDF ou P(EC)*	NS
si ¬FDF alors	1	*AV*	NS-1+ε
VISITER(EC)	1-ε1	*fitérer*	
fsi		*fsi*	
TERM_P	1	*TERM_P*	1
Evaluation :(ε1 a le sens ci-dessus) accès séquentiel : *DEM*:1, *AV*:NS+ε1-1, *FDF*:NS+1+ε1 *P*:NS+ε1 traitement : *INIT_P*:1, *TERM_P*:1, *VISITER*:NS contrôle : NS+1+ε1 opérations logiques : NS+ε1		Evaluation : accès séquentiel : *DEM*:1-ε, *AV*:NS-1+ε, *FDF* :NS *FILE_VIDE*:1, *P*:NS traitement : *INIT_P*:1, *TERM_P*:1, *VISITER* :NS contrôle:NS+1 opérations logiques:NS	

3. Un exemple d'application : parcours d'une file à concurrence des N premiers éléments.

On reprend le problème de parcours : on doit visiter les N premiers éléments de la file. Si la file comporte moins de N éléments, elle est entièrement parcourue.

Construction de l'algorithme

Nous appliquons le traitement séquentiel de la manière suivante : pour assurer le comptage, nous associons à chaque élément de la file donnée son rang dans la file. Nous définissons donc une nouvelle file.

4. Conclusion : utilisation des schémas de parcours d'une sous-file.

La programmation d'un parcours de sous-file comporte :

. la description de la file dont on veut parcourir une sous-file, et le choix d'un mode d'accès séquentiel à cette file.

. la reconnaissance du type de définition de la sous-file (par inclusion ou par exclusion).

. l'application de l'un des schémas que nous venons d'étudier en fonction des réponses aux questions précédentes.

. éventuellement, la simplification du programme ainsi obtenu en tenant compte de détails spécifiques au problème étudié (qui dépassent le niveau d'abstraction des schémas ci-dessus).

2.2.4. - Recherche d'un élément dans une file

Nous construisons les schémas de programme répondant au problème de "recherche associative" d'un élément dans une file : on cherche à traiter par l'action *T_OUI* le premier élément de la file vérifiant une propriété donnée *P*. Si un tel élément n'existe pas, on applique l'action *T_NON* (cf. § 1.2.3.).

a) Construction d'un algorithme

Pour construire un algorithme, nous pouvons partir de l'étude du § 1.2.3. et par une technique analogue à celle que nous avons employée pour l'étude des parcours (§ 2.2.2.), obtenir un schéma.

Une autre manière de faire est d'appliquer les schémas de parcours de sous-files, en interprétant le problème de la manière suivante :

Comme au § 1.2.3., le problème est décomposé en deux parties, la recherche de l'élément vérifiant *P* et l'application du traitement en fonction du résultat de cette recherche. On considère alors qu'il s'agit d'un parcours d'une sous-file où :

. la fin de la sous-file est définie par exclusion

. le traitement est défini par les primitives suivantes (en suffixant les noms par le caractère ') :

VISITER' : action vide
INIT_P' : action vide
TERM_P' : *si* "trouvé" *alors T_OUI sinon T_NON fsi*

La manière de caractériser le résultat de la recherche, dépend de l'itération de recherche, c'est-à-dire du mode d'accès séquentiel à la file :

. machine séquentielle de type 1 :
l'échec dans la recherche est caractérisé par le fait que le voyant est allumé.

. machine séquentielle de type 2 :
la réussite dans la recherche est caractérisée par le fait que l'on termine par un élément vérifiant P.

b) Algorithmes :

recherche associative dans une file	
algorithme RA1 file donnée accessible par une machine de type 1	algorithme RA2 file donnée accessible par une machine de type 2
DEM 1 *tantque* ¬*FDF et_c* ¬*P(EC)* *faire* NS+1 *AV* NS *ftantque* *si FDF* 1 *alors T_NON* ε1 *sinon T_OUI(EC)* 1-ε1 *fsi*	*si FILE_VIDE* 1 *alors T_NON* ε *sinon* *DEM* 1-ε *tantque* ¬*FDF et* ¬*P(EC)* *faire* NS+1-ε-ε *AV* NS-ε1 *ftantque* *si P(EC)* 1-ε *alors T_OUI(EC)* 1-ε1-ε *sinon T_NON* ε1 *fsi* *fsi*
Evaluation :	Evaluation :
NS est le nombre d'éléments précédant l'élément recherché	
accès séquentiel : *DEM*:1, *AV*:NS, *FDF*:NS+2, *P*:NS+1-ε1 contrôle : NS+2 opérations logiques : NS+1 -ε1	accès séquentiel *DEM*:1-ε, *AV*:NS-ε1, *FDF*:NS+1-ε-ε1 *P*:NS+2-2ε-ε1, *FILE_VIDE*:1 contrôle : NS+3-2ε-ε1 opérations logiques : NS+1-ε1-ε

c) Exemple d'application :

Parmi les exemples donnés au § 1.2.4., nous traitons ici la recherche du dernier élément de la file, car son étude est un peu délicate (dans la mesure où la caractérisation de l'élément recherché est liée au mode d'accès à la file).

Analyse :

Une application directe de l'algorithme de recherche associative découle de l'idée suivante :

On recherche le premier élément n'appartenant pas à la file (!) : il n'existe pas et la recherche se termine toujours par un échec.
Cette idée n'est valable que si la file donnée est accessible par une machine séquentielle de type 2 : en effet dans ce cas, lorsque l'algorithme de recherche se termine, *EC* est effectivement le dernier élément de la file.

Par contre, si la file donnée est accessible par une machine séquentielle de type 1, lorsque la recherche est terminée, on a quitté le dernier élément. Cette remarque nous conduit alors à envisager une autre application du traitement séquentiel :

On considère la file des couples d'éléments successifs dans la file donnée, et on en recherche le dernier couple. Nous avons vu au travers de l'exemple donné au § 2.1.3. que l'accès séquentiel à une telle file peut être décrit par les deux modèles de machine : on décrit donc la file des couples d'éléments successifs dans la file donnée par une machine de type 2, et on recherche le dernier couple.

Construction de l'algorithme :

Cas n°1 : file accessible par une machine de type 1 :

accès séquentiel à la file des couples successifs (type 2) :

. état :

On doit constamment connaître deux éléments successifs de la file : on utilise une variable intermédiaire, notée *ep* pour contenir l'élément précédant l'élément courant *EC*

. répertoire :

FILE_VIDE'	:	*FILE_VIDE*
EC'	:	*ep,EC*
FDF'	:	*FDF*
AV'	:	*ep:=EC ; AV*
DEM'	:	*DEM ; ep:=EC ;* AV {non défini si *FILE_VIDE=vrai*}

Traitement :

On applique l'algorithme de recherche associative RA2 à cette file de couples avec la propriété *P : faux* (recherche du premier couple n'appartenant pas à la file).

Cas n°2 : file accessible par une machine de type 2 :

On applique directement l'algorithme de recherche associative RA2 à la file donnée, avec la propriété *P : faux*.

Algorithmes :

recherche du dernier élément d'une file	
Algorithme R-D1 file donnée accessible par une machine de type 1	algorithme R-D2 file donnée accessible par une machine de type 2
si FILE_VIDE 1 *alors T_NON* *sinon* *DEM;ep:=EC;AV* 1-ε *tantque ¬FDF faire* N *ep1:=EC;AV* N-1+ε *ftantque* *T_OUI(ep)* 1-ε *fsi* Evaluation : accès séquentiel : *DEM*:1, *AV*:N, *FDF*:N *FILE_VIDE*:1 contrôle : N+1 affectations:N	*si FILE_VIDE* 1 *alors T_NON* *sinon* *DEM* 1-ε *tantque ¬FDF faire* N *AV* N-1+ε *ftantque* *T_OUI(EC)* 1-ε *fsi* Evaluation : accès séquentiel : *DEM*:1, *AV*:N-1+ε, *FDF* :N *FILE_VIDE* :1 contrôle:N+1

Remarques :

- On voit bien que les deux algorithmes sont issus du même schéma de base (appliqué à deux files différentes).
- l'algorithme R-D1 peut être légèrement simplifié en intégrant l'appel initial de AV dans le corps de l'itération (cf. 2.3.1.).

2.3 - ETUDE DE CAS PARTICULIERS

Nous terminons cette étude de la programmation du traitement séquentiel en examinant deux cas particuliers dont la fréquence d'apparition dans les problèmes est suffisamment importante pour justifier l'écriture des schémas associés. Ils sont liés à des hypothèses supplémentaires sur la file caractérisant le traitement séquentiel :

. le premier cas est celui où l'accès séquentiel peut être initialisé de telle sorte que le démarrage se ramène directement à un avancement. Ceci est assez fréquent et s'inspire directement des protocoles standards d'accès aux fichiers séquentiels (l'action d'ouverture d'un fichier joue, entre autres, ce rôle d'initialisation de l'accès séquentiel). Du point de vue des schémas, on obtient de simples changements dans la forme (de manière à respecter notre souci de concision , cf. 2.2.1. a)).

. Le second cas concerne les schémas de parcours de sous-files et de recherche associative. Nous considérons le cas particulier où l'on sait a priori que la file traitée comporte au moins un élément vérifiant la propriété P (caractérisant la sous-file ou la recherche).

Ces deux cas particuliers débouchent sur des techniques classiques de représentation.

2.3.1. - Accès séquentiel initialisé par une action d'"ouverture"

a) Machine séquentielle

L'action d'ouverture est désignée par :

INITIALISATION_ACCES_SEQUENTIEL en abrégé *INIT_AS*

l'action de démarrage est décomposée comme suit :

DEMARRER : INIT_AS ; AVANCER

Spécification de *INIT-AS* :notion de *"sentinelle"*

Pour spécifier *INIT-AS* , nous recourrons à une notion de "sentinelle" que l'on doit pouvoir définir pour être dans les conditions du cas particulier.
On considère un objet noté

sentinelle en abrégé *sent*

Cet objet est de même nature que les éléments de la file, mais est différent de tous les éléments effectivement présents dans la file.
On doit de plus pouvoir étendre les relations *SUC* et *E_S* comme suit :

. *FILE_VIDE=vrai* : *E_S(sentinelle)=faux*
SUC(sentinelle) : non défini
. *FILE_VIDE=faux* : *E_S(sentinelle)=vrai*
SUC(sentinelle)=PREMIER

Remarque :
Une interprétation intuitive de la "sentinelle" est la suivante: tout se passe comme si la file considérée était bordée par un élément supplémentaire (la sentinelle), et toujours le même, quelle que soit la file effective (à la nature des éléments près) de telle sorte que l'on ait à considérer une file qui ne soit jamais vide (elle comporte au moins la sentinelle) et dont le premier élément (la sentinelle) soit facile à déterminer (son "nom" est connu a priori, quelle que soit la file donnée).

Dans ces conditions, les descriptions des deux modèles de machine séquentielle (§ 2.1.2.) peuvent être adaptées au cas particulier considéré, en rajoutant les spécifications suivantes :

Machine de type 1

- *INIT-AS*

état initial :

quelconque (cf. spécifications de *DEMARRER)*

état final :

{compatible avec l'état initial de *AVANCER*, de manière à ce que l'effet de *AVANCER* conduise à l'état final spécifié pour *DEMARRER*. Soit :}

FDF=faux et EC=sent

- *AVANCER*

doit être défini pour la *sentinelle* en accord avec le rôle de cet objet.

Machine de type 2

- *INIT-AS*

état initial :

quelconque

état final :

. *FILE_VIDE=vrai* : *FDF=vrai et EC=sent*
. *FILE_VIDE=faux* : *FDF=faux et EC=sent*

- *AVANCER*

doit être défini pour la sentinelle en accord avec le rôle de cet objet.

Remarque :
On constate que l'introduction de l'action *INIT-AS* permet d'intégrer le cas de la file vide, par une spécification particulière de l'état final. (ceci est cohérent avec l'interprétation intuitive de la sentinelle donnée ci-dessus).

b) Application aux schémas de traitement séquentiel

On peut remplacer dans tous les schémas donnés précédemment *DEMARRER* par sa définition en termes de *INIT-AS* et *AVANCER* . Pour éviter que l'action *AVANCER* n'apparaissent deux fois dans les schémas (ce qui découle de cette substitution), on peut changer la forme des schémas, en recourrant à une construction d'itération adéquate. Par exemple dans le cas des schémas de parcours d'une file, on obtient :

parcours d'une file
accès séquentiel initialisé

algorithme P1bis		algorithme P2bis	
file donnée accessible par une machine de type 1		file donnée accessible par une machine de type 2	
INIT_P	1	*INIT_P*	1
INIT_AS	1	*INIT_AS*	1
itérer		*tantque ¬FDF faire*	N+1
AV	N+1	*AV*	N
arrêt : FDF	N+1	*VISITER(EC)*	N
VISITER(EC)	N	*ftantque*	
fitérer		*TERM_P*	1
TERM_P	1		

les évaluations sont inchangées

Remarques:

- Nous ne donnons pas les schémas correspondant aux parcours de sous-file et à la recherche. Nous y ferons référence, le cas échéant, en suffixant le nom de l'algorithme par "bis".

- Soulignons l'analogie entre la "simplification" présentée ici et les techniques de représentation de listes linéaires (cf. [Knu 69], [Wir 76] par exemple) utilisant des "doublets" fictifs pour ancrer les listes manipulées (l'intérêt de la technique étant justement d'éviter le cas particulier de la liste vide).

2.3.2. - Cas particulier pour le parcours d'une sous-file et la recherche

Le cas particulier que nous envisageons ici est celui où un parcours de sous-file, ou une recherche est défini sur une file dont on sait à priori, qu'elle contient au moins un élément vérifiant la propriété *P*.

La conséquence de cette hypothèse est de permettre de simplifier (par un raisonnement évident) les schémas. Outre une simplification de la forme, on obtient une "amélioration" du coût de l'algorithme dans la mesure où un certain nombre de tests sont évités.

A titre d'exemple, nous donnons les schémas de parcours de sous-files :

On a maintenant le même schéma indépendamment du modèle de machine

2.3.3. - Raffinement des programmes

Comme l'illustrent les cas particuliers que l'on vient d'étudier, l'application de la méthode de traitement séquentiel conduit éventuellement à affiner, dans une dernière phase, le programme qu'on a pu construire en appliquant les techniques systématiques que nous venons de décrire et d'illustrer. Cet affinement du programme est fait en appliquant un certain nombre de transformations qui tiennent compte à la fois des critères que le programme final doit satisfaire et des spécificités du problème étudié (par rapport à l'abstraction à laquelle se placent les techniques employées pour produire le programme). A ce stade on peut utiliser les nombreuses techniques de transformation (et notamment d'optimisation) qui ont pu être étudiées (cf. [Knu 74][Bau 76] [BPP 76] [Mon 76] [Weg 76] [Cha 77] [Ars 79]).

2.4. CONCLUSION : UN RESUME DU TRAITEMENT SEQUENTIEL

Nous résumons ici la méthode du traitement séquentiel en proposant un récapitulatif d'une part des modèles et outils proposés, et d'autre part des étapes du processus de construction de programmes qu'implique l'application systématique de la méthode. Ces récapitulatifs sont organisés sur la base des deux principes de base du traitement séquentiel.

2.4.1. - Récapitulatif des modèles et outils proposés

A. recherche d'une organisation logique de l'information permettant de la structurer en une file d'informations élémentaires (cf. introduction).

Pour aider à cette organisation, nous avons proposé plusieurs étapes et donné les modèles de réalisation et de description associés :

1. un modèle du processus de structuration en file (§ 1.1.2).
2. un modèle de description de la file (§ 1.1.3) comportant
   - la description de l'information élémentaire (résultat du raffinement)
   - la description des relations (résultat de la structuration).

parcours d'une sous-file
il existe un élément vérifiant la propriété P

algorithme PEX3 sous-file définie par exclusion		algorithme PIN3 sous-file définie par inclusion	
INIT_P	1	*INIT_P*	1
DEM	1	*DEM*	1
tantque ¬P(EC) faire	NS+1	*itérer*	
VISITER(EC)	NS	*VISITER(EC)*	NS
AV	NS	*arrêt : P(EC)*	NS
ftantque		*AV*	NS-1
TERM_P	1	*fitérer*	
		TERM_P	1
Evaluations : accès séquentiel : *DEM*:1, *AV*:NS, *FDF*:0, *P*:NS+1 contrôle : NS+1		Evaluations : accès séquentiel : *DEM*:1, *AV*:NS-1, *FDF*:0, *P*:NS contrôle : NS	

Remarques :

- l'hypothèse implique que la file n'est pas vide.
- on peut comparer les évaluations avec les chiffres obtenus précédemment : le gain correspond aux tests relatifs à la fin de la file.
- on peut constater l'analogie entre ces schémas et les schémas de parcours de file. L'algorithme PEX3 correspond à l'algorithme P1: il peut être construit directement en appliquant P1 au traitement de la sous-file définie par exclusion. Inversement le parcours d'une file accessible par une machine de type 1 peut être interprété comme un parcours de sous-file définie par exclusion (le premier élément non visité est celui pour lequel le voyant est allumé).
  De même l'algorithme PIN3 correspond à l'algorithme P2 : on peut construire l'un à partir de l'autre sur la base d'interprétations évidentes du problème de départ.
- C'est sur la base de l'analogie précédente qu'est construite la progression d'exercices qui nous permet de présenter le traitement séquentiel aux étudiants (LuS 75) : les exemples proposés correspondent toujours à des sous-files de files concrètes (files de caractères notamment).
- Nous ne donnons pas ici l'algorithme de recherche associative RA3, d'un élément dans une file où on sait le trouver. Il se déduit directement de l'algorithme PEX3. Le gain en tests par rapport aux algorithmes RA1 et RA2 justifie la technique classique dite de "sentinelle" (cf. par exemple Wir 76) : on représente les listes linéaires dans lesquelles sont effectuées des recherches, en les bordant à droite (en queue) par un "doublet sentinelle". La recherche d'un élément dans la liste est alors précédée par le stockage dans le "doublet sentinelle" de la valeur cherchée.

3. un modèle d'accés séquentiel à cette file, appelé "machine séquentielle", comportant :
   - la définition de deux types d'accés séquentiel (§ 2.1.1.)
   - la définition d'un cas particulier important (§ 2.3.1.).

4. un modèle de description des machines séquentielles (§ 2.1.2.) comportant :
   - la description de l'état de la machine
   - la description du répertoire d'actions élémentaires permettant la progression le long de la file.

B. recherche d'une formulation du problème posé, en termes d'un traitement de cette file.

Pour aider à cette formulation, nous avons proposé

1. un modèle d'analyse récurrente (§ 1.1.6.).

2. un ensemble de schémas fondamentaux répartis en trois groupes :
   - parcours de files (§ 1.2.1. et § 2.2.2.)
   - parcours de sous-files (§ 1.2.2. et § 2.2.3.) distinguant les sous-files définies par exclusion et les sous-files définies par inclusion.
   - recherche "associative" dans une file (§ 1.2.3. et § 2.2.4.)

2.4.2. - Etapes du processus de construction de programmes

Le tableau suivant donne les diverses étapes que nous avons fait apparaître. Chaque étape est caractérisée par un nom et le résultat auquel elle aboutit.

A. Organisation logique de l'information		
	1. raffinement	Description de l'information élémentaire
	2. structuration	Description des relations
	3. choix d'un type d'accès séquentiel	Description d'une machine séquentielle - état - répertoire
	4. Examen de cas particuliers	Raffinement de la machine séquentielle
B. Formulation du problème en termes d'un traitement de file		
	1. Analyse récurrente (choix entre deux type d'analyse par cas	Algorithme récursif
	1'.Identification de problèmes	Choix d'un schéma
	2. Examen de cas particuliers	Raffinement du schéma
	3. Evaluation	mesure du comportement de l'algorithme

CHAPITRE 3

APPLICATION SYSTEMATIQUE DU TRAITEMENT SEQUENTIEL

Nous proposons dans ce chapitre trois exemples d'application du traitement séquentiel. Chacun de ces exemples a un triple objectif :

. illustrer l'application directe des outils présentés au cours des chapitres précédents.

. Montrer comment on peut proposer systématiquement plusieurs solutions à un problème posé, en se basant sur les deux types de choix que fait intervenir le traitement séquentiel :

- choix d'une file et du traitement associé
- choix d'une implantation de la file.

Les deux premiers exemples illustrent plus particulièrement la manière d'obtenir plusieurs solutions en changeant l'interprétation du problème posé en termes du traitement d'une file. Le troisième exemple illustre les diverses manières de programmer un traitement séquentiel en appliquant les résultats du chapitre 2, une fois qu'on a choisi l'interprétation du problème en termes d'un traitement d'une file.

. Etre représentatif d'une classe importante d'applications : les démarches illustrées dans chaque exemple peuvent être utilisées directement dans des contextes nombreux et variés.

3.1 - PREMIER EXEMPLE : ENUMERATION PARTIELLE D'UNE FILE

Ce premier exemple reste dans le style des schémas du chapitre 2 : on étudie un problème général de traitement de file et on construit plusieurs solutions en appliquant le traitement séquentiel. Nous voulons de plus illustrer une démarche qui consiste à rechercher le maximum de solutions à un problème posé. Le traitement séquentiel suggère pour cela

. de varier l'interprétation initiale du problème en termes du traitement d'une file.
. d'étudier divers cas particuliers.

Le problème est le suivant :

Etant donné une file, on désire appliquer une action V à chacun de ses éléments vérifiant une propriété P .
On suppose que la file est accessible à l'aide d'une machine séquentielle de type 1 (cf. § 2.1.2.).

Remarque :
Tous les résultats que nous obtenons peuvent être facilement adaptés au cas où la file est accessible par une machine séquentielle de type 2 (cf. § 2.1.2.).

Nous résolvons ce problème en appliquant le traitement séquentiel de trois manières différentes :

. application directe de l'algorithme de parcours à la file donnée : l'action *VISITER* est l'application conditionnelle de V suivant que l'élément considéré vérifie ou non la propriété P.

. application de l'algorithme de parcours à une file déduite de la file donnée : la file comportant uniquement les éléments (de la file donnée) vérifiant la propriété P.

. application d'un algorithme de parcours d'une sous-file à chacun des groupes d'éléments consécutifs de la file donnée qui vérifient tous la propriété donnée.

Nous évaluons le coût de chacune des solutions obtenues en termes des quantités suivantes :

N : nombre d'éléments de la file donnée
PV : nombre d'éléments vérifiant P
PF : nombre d'éléments ne vérifiant pas P
GV : nombre de groupes d'éléments consécutifs vérifiant tous P
GF : nombre de groupes d'éléments consécutifs ne vérifiant pas P.

Remarque :
PV+PF = N

3.1.1. - Application directe d'un algorithme de parcours à la file donnée

L'action *VISITER* est définie comme suit

VISITER(E) : si P(E) alors V(E) fsi

On applique le schéma P1 de parcours d'une file (cf. 2.2.2.) :

Algorithme : énumération partielle - version 1

INIT_P	1
DEM	1
tantque ¬FDF faire	N+1
si P(EC) alors	N
V(E)	PV
fsi	
AV	N
ftantque	
TERM_P	1

Evaluation :
accès séquentiel
DEM:1 , *AV*:N , *FDF*:N+1 , *P*:N
contrôle : 2N+1

3.1.2. - Définition de la file des éléments vérifiant la propriété *P*

a) Analyse

1 - Définition de la file

. un élément de la file est un élément de la file donnée vérifiant *P*

. les primitives sont :

- *PREMIER* le premier de la file qui vérifie *P*. On le calcule à l'aide d'un schéma de recherche associative (§ 2.2.4

- *SUCCESSEUR* : le successeur d'un élément est le prochain élément vérifiant *P*. Son existence et sa valeur sont déterminées par application d'un schéma de recherche associative.

2. traitement

L'algorithme est alors un simple parcours de cette file : à chacun de ses éléments est appliqué *V* .

b) Construction de l'algorithme

1. Choix d'un type d'accès séquentiel

Comme on ne peut savoir si un élément de la file est le dernier qu'en cherchant à trouver son successeur éventuel, on se base sur le premier modèle de machine séquentielle.

2. Description de l'accès séquentiel

. Etat

correspond à la représentation de la file donnée

. Répertoire

```
EC'        : EC
FDF'         FDF
AVANCER'   : {recherche associative du prochain élément vérifiant
              P. On applique RA1}
             AV
             tantque ¬FDF et_c ¬P(EC) faire AV ftantque
             {non trouvé si FDF=vrai}
DEMARRER'  : {recherche associative du premier élément
              vérifiant P}
             DEM
             tantque ¬FDF et_c ¬P(EC) faire AV ftantque
             {non trouvé si FDF=vrai}
```

3. Choix d'un schéma

On applique l'algorithme P1 de parcours de la file ci-dessus.
Le traitement est défini par :

VISITER : V

4. Simplification

L'application du schéma P1 conduit à un algorithme qui comporte deux fois l'itération interne de recherche associative du prochain élément vérifiant *P*.

Ceci se simplifie aisément, en utilisant une autre primitive d'itération.

c) Algorithme : énumération partielle - version 2

```
INIT_P                                          1
DEM                                             1
itérer tantque ¬FDF et_c ¬P(EC) faire           N+1
        AV                                      PF
      ftantque
arrêt : FDF                                     PV+1
        V(EC)                                   PV
        AV                                      PV
fitérer
TERM_P                                          1
```

évaluation
accès séquentiel :
DEM:1, *AV*:N, *FDF*:N+PV+1, *P*:N
contrôle : N+PV+2
opérations logique : N

d) Cas particulier :

On considère l'hypothèse simplificatrice où *FDF* implique *P(EC)* (i.e. on se met dans les conditions du § 2.3.2.).

Algorithme : énumération partielle - version 2 bis

```
{FDF=vrai implique P(EC)=vrai}
INIT_P ; DEM
itérer tantque ¬P(EC) faire AV ftantque
arrêt : FDF
        V(EC) ; AV
fitérer
TERM_P
```

Evaluation : On a supprimé les opérations logiques par rapport à la version 2.

3.1.3. - Regroupement des éléments consécutifs vérifiant P

a) Analyse

On considère la file des groupes d'éléments consécutifs vérifiant *P* Chacun de ces groupes est "désigné" par son premier élément. Intuitivement le traitement consiste à atteindre chacun de ces groupes dans l'ordre où ils se trouvent dans la file donnée et pour chaque groupe à appliquer l'action *V* à tous ses éléments. L'algorithme que nous construisons est une application répétée d'une séquence comportant le parcours d'une sous-file définie par exclusion (§ 2.2.3), et une recherche associative (§ 2.2.4.).

b) Construction

Nous ne détaillons pas ici la construction systématique de cet algorithme : nous le ferons à l'occasion du troisième exemple de ce chapitre (§ 3.3.). Nous nous contentons ici de donner diverses formes de solutions basées sur un regroupement des éléments vérifiant *P*.

c) Algorithmes :

énumération partielle			
Version 3		version 4	
INIT_P	1	*INIT_P*	1
DEM	1	*DEM*	1
itérer		*itérer*	
tantque ¬FDF et_c ¬P(EC) faire	GV+PF+1	*tantque ¬FDF et_c P(EC) faire*	GF+PV+1
AV	PF	*V(EC) ; AV*	PV
ftantque		*ftantque*	
arrêt : FDF	GV+1	*arrêt : FDF*	GF+1
répéter V(EC) ; AV	PV	*répéter AV*	PF
jusqu'à FDF ou_c ¬P(EC)	PV	*jusqu'à FDF ou_c P(EC)*	PF
fitérer		*fitérer*	
TERM_P	1	*TERM_P*	1
Evaluation :		Evaluation :	
. accès séquentiel :		. accès séquentiel :	
DEM:1, *AV*:N, *FDF*:N+GV+1		*DEM*:1, *AV*:N, *FDF*:N+GF+1	
P:N+GV+1		*P*:N+GF+1	
. contrôle : N+2GV+2		. contrôle : N+2GF+2	
. opérations logiques : N+GV+1		. opérations logiques : N+GF+1	

Remarques :

- on peut constater l'analogie entre les versions 3 et 4 : la version 4 est obtenue en inversant les rôles dans le contrôle de l'algorithme, des éléments vérifiant *P* et de ceux ne le vérifiant pas.
- les versions 3 et 4 admettent des "simplifications" dans la mesure où l'on peut appliquer une technique de sentinelle : ou bien on peut assurer que *FDF=vrai* implique *P(EC)* , ou bien on peut assurer *FDF=vrai* implique ¬*P(EC)* . Dans chacun de ces cas on peut simplifier soit la recherche associative, soit le parcour de sous-file. (cf. § 2.3.2.).

Enfin nous pouvons donner une forme moins symétrique mais qui est analogue à la version 2 :

Algorithme : énumération partielle - version 5

INIT_P	1
DEM	1
itérer tantque ¬FDF et_c P(EC) faire	N+1
V(EC) ; AV	PV
ftantque	
arrêt : FDF	PF+1
AV	PF
fitérer	
TERM_P	1

Evaluation : cf. version 2 en remplaçant PV par PF

Cas particulier : ici encore, on peut chercher à appliquer la technique de sentinelle pour simplifier l'itération interne. Ceci sera possible si on peut assurer :

version 5 bis

FDF=vrai implique *P(EC)=faux*

on obtient alors une version 5 bis que nous ne donnons pas ici.

3.1.4. - Discussion

Outre l'utilisation directe des outils proposés au chapitre 2, cet exemple nous a permis de montrer comment rechercher diverses solutions à un problème posé en variant l'interprétation du problème en termes d'un traitement de file.

Comparons les solutions obtenues :

. clairement, la première solution semble être la plus simple tant du point de vue de son analyse (et donc de sa preuve) que de sa forme. Par ailleurs si l'on reste au niveau d'abstraction du problème posé (i.e. on ne rentre pas dans le détail de l'action v), seules les versions 2 bis et 5 bis sont plus "performantes" que la version 1.

. les autres versions font toutes intervenir un "regroupement" soit des éléments vérifiant P, soit des éléments ne vérifiant pas P. Du point de vue de l'idée qui y conduit, elles sont tout aussi "naturelles" que la version 1, surtout si l'on se trouve dans un contexte qui suggère un regroupement plutôt qu'un traitement "élément par élément" (ceci dépend de l'énoncé du problème).

. toutefois, l'intérêt effectif des versions 2 à 5 n'apparaît que si la définition de l'action v fait intervenir une distinction entre les éléments selon leur position dans la file (par exemple, on distingue le traitement du premier élément d'un groupe d'éléments vérifiant la propriété P - ceci est notamment le cas dans de nombreux traitements de textes faisant intervenir une notion de "mots"). Dans ce cas, les arguments que nous avons donnés ci-dessus, s'appliquent plutôt aux versions faisant intervenir un regroupement qu'à la version 1.
(on trouvera de nombreux exemples dans [LuS 75], vol. 1, Chap. 2).

Pour terminer, soulignons que l'on peut développer le même type d'analyse pour résoudre un problème légèrement plus général que celui que nous avons traité : il s'agit d'appliquer une action V1 à tous les éléments vérifiant une propriété P et une action V2 à tous les éléments qui ne la vérifient pas.

3.1.5. - Conclusion

La discussion qui précède montre la difficulté que nous avons à enfermer dans des schémas une certaine expérience de programmation : dès que l'on arrive à un certain degré de complexité, on est pris entre deux extrêmes :

. ou bien préciser trop de détails et perdre l'idée même de schéma général

. ou bien rester à un trop grand niveau d'abstraction et ne pas faire apparaître la majeure partie des problèmes rencontrés lors de la construction d'un programme relevant de ce schéma.

Nous reviendrons sur cette question d'ordre méthodologique dans une conclusion ultérieure.

3.2 - DEUXIEME EXEMPLE : FUSION DE FICHIERS

Dans ce deuxième exemple, nous voulons montrer que l'utilisation du traitement séquentiel dans une application particulière, (ici le traitement de fichiers séquentiels) peut passer par une re-définition des notations en fonction de cette application. L'exemple est de plus une nouvelle occasion de montrer la construction de plusieurs solutions basées sur diverses interprétation du problème posé en termes de traitement de files. Enfin, l'exemple, joue un rôle important dans l'application considérée.

Le problème posé est le suivant :

Etant donné deux fichiers séquentiels F1 et F2 comportant des enregistrements de même nature et ordonnés selon la même relation d'ordre, réaliser une fusion de ces deux fichiers, c'est-à-dire produire un troisième fichier F3, trié lui-aussi et comportant tous les enregistrements des deux fichiers donnés (on comparera deux enregistrements en utilisant les symboles habituels des opérateurs de comparaisons d'entiers).

3.2.1. - Notations

Pour résoudre notre problème, nous reprenons dans [Sch 79 a], des notations permettant de décrire des algorithmes de traitement de fichiers (ces notations sont une application directe au cas des fichiers, des notations données au paragraphe 2.1).

Convention de fin de fichier

Nous supposons que la fin du fichier est détectée à l'aide d'un mécanisme d'interruption classique : l'interruption à lieu lors d'une tentative de lecture après le dernier enregistrement du fichier (ou lors d'une tentative de lecture dans un fichier vide). Cette interruption est "récupérée" au niveau programme par la convention suivante : à tout fichier en cours d'exploration est associé un prédicat, noté *FDF* , qui prend la valeur *vrai* lorsque l'interruption a lieu et a la valeur *faux* au cours de l'exploration du fichier (cela correspond exactement aux conventions d'un langage comme PASCAL, et peut être facilement implémenté dans un langage comme PL/1,...).

Notations : (dans ce qui suit *F* désigne un fichier et *X* un enregistrement).

a) Lecture d'un fichier :

. situation par rapport à la fin

fdf est un prédicat indiquant si la fin de fichier est atteinte :

fdf(f) : *vrai* si la fin est atteinte, *faux* sinon.

. avancement dans le fichier

lire est une action qui avance le long du fichier et fournit le prochain enregistrement s'il existe. S'il n'existe pas, la fin de fichier est signalée.

lire(f,x) : deux effets différents suivant l'état initial.

- on n'est pas au départ positionné sur le dernier élément, alors on avance d'un cran sur le fichier et à la fin *X* contient le nouvel enregistrement.
- on est au départ positionné sur le dernier élément, alors l'état du mécanisme de lecture est modifié de telle sorte que le prédicat *fdf* prenne la valeur *vrai* . La valeur de *X* à la fin est quelconque.

On ne peut pas appeler *lire* si *fdf=vrai*

. initialisation de la lecture :
ouvrir est une action qui initialise le mécanisme de lecture de telle sorte que le prochain appel de l'action *lire* donne accès au premier élément du fichier. Après l'appel d'*ouvrir* le prédicat *fdf* a la valeur *faux*.

ouvrir(f,L) : initialise le mécanisme de lecture avant une lecture du fichier *f* (le paramètre *L* sert à distinguer l'ouverture en lecture de l'ouverture en écriture).

toute exploration d'un fichier doit commencer par l'appel de *ouvrir*

. terminaison de la lecture :
fermer est une action qui termine toute opération de lecture ou d'écriture d'un fichier.

fermer(f)

b) Ecriture sur un fichier

. adjonction d'un enregistrement à la suite
écrire est une action qui permet de placer un nouvel enregistrement à la suite des enregistrements que l'on vient de placer (juste après le dernier que l'on a placé).

écrire(f,x) : l'enregistrement *x* est placé sur le fichier *f* sur la prochaine position par rapport au dernier élément que l'on a écrit.

écrire peut provoquer une erreur de saturation si il n'y a plus de place.

. initialisation de l'écriture :
ouvrir est une action qui initialise le mécanisme d'écriture de telle sorte que le prochain appel de *écrire* permette de placer un enregistrement en première position du fichier

ouvrir(f,E)

. terminaison de l'écriture

fermer(f) : assure que le fichier que l'on vient de créer se termine de manière cohérente par rapport à la convention de fin de fichier.

Evaluations

On évaluera les algorithmes en fonction des quantités suivantes: N1, N2 et N3 désignent respectivement les tailles (nombre d'enregistrements) des fichiers F1, F2 (données) et F3 (résultat). Q1 est le nombre d'éléments de F1 supérieurs au plus grand élément de F2. Q2 est le nombre d'éléments de F2 supérieurs au plus grand élément de F2. (Q1=0 ou Q2=0). On pose Q = Q1+Q2

3.2.2. - première approche : file de couples

a) Analyse

On considère que le fichier résultat est obtenu en traitant une file de couples définie comme suit :

. un couple est composé d'un enregistrement du premier fichier et d'un enregistrement du deuxième fichier soit *x1,x2*.

. étant donné un couple *x1*, *x2*, le prochain couple à traiter est formé comme suit : si *x1* est le plus petit des deux, *x1* est remplacé par le prochain enregistrement du premier fichier (s'il existe).

si *x2* est le plus petit des deux enregistrements, *x2* est remplacé par le prochain enregistrement du deuxième fichier (s'il existe).

. le traitement d'un tel couple consiste à recopier sur le fichier résultat le plus petit des deux enregistrements.

. le traitement s'arrête dès que l'on atteint la fin de l'un des deux fichiers. Il faut alors le compléter par la recopie de la fin du fichier qui n'est pas terminé (le cas échéant).

b) Construction de l'algorithme

1. Choix du type d'accès séquentiel

On ne peut détecter si un couple est le dernier qu'en cherchant à constituer le suivant. On utilise donc le modèle 1.

2. Description de l'accès séquentiel

Etat :

On utilise deux variables x1 et x2 pour représenter le couple courant.

Répertoire :

```
EC'       : x1,x2
FDF'      : fdf(f1) ou fdf(f2)
AVANCER'  : choix
              x1<x2 : lire (f1,x1)
              x1≥x2 : lire (f2,x2)
            fchoix
DEMARRER' : ouvrir (f1,L) ; ouvrir (f2,L)
            lire (f1,x1) ; lire (f2,x2)
```

3. Description du traitement

Il s'agit d'un parcours de la file de couples (algorithme P1) avec :

```
INIT_P          : ouvrir (f3,E)
VISITER(x1,x2)  : choix
                    x1≤x2 : écrire (f3,x1)
                    x1>x2 : écrire (f3,x2)
                  fchoix
TERM_P          : {on termine éventuellement la recopie d'une
                   des files}
                  tantque ¬fdf(f1) faire écrire(f3,x1);
                   lire(f1,x1) ftantque
                  tantque ¬fdf(f2) faire écrire(f3,x2) ;
                   lire(f2,x2) ftantque
                  fermer(f1) ; fermer(f2) ; fermer(f3)
```

4. Simplification

L'application du schéma P1 conduit à un algorithme comportant deux *choix* successifs contrôlés par les mêmes conditions. Ceci est simplement transformé en combinant ces deux instructions *choix* en une seule.

c) Algorithme : fusion - version 1

```
ouvrir(f3,E) ; ouvrir(f1,L) ; ouvrir(f2,L)
lire(f1,x1) ; lire(f2,x2)                    {lecture 1er couple}
tantque ¬fdf(f1) et ¬fdf(f2) faire
  {tous les éléments de f1 et f2 précédant x1 et x2 sont dans f3.
   f3 est trié. Le dernier élément de f3 est inférieur ou égal à
   x1 et x2}.
   choix
   x1<x2 : écrire(f3,x1) ; lire(f1,x1)
   x1≥x2 : écrire(f3,x2) ; lire(f2,x2)
   fchoix
ftantque
tantque ¬fdf(f1) faire écrire(f3,x1) ; lire(f1,x1) ftantque
tantque ¬fdf(f2) faire écrire(f3,x2) ; lire(f2,x2) ftantque
fermer(f1) ; fermer(f2) ; fermer(f3)
```

Evaluation :
contrôle : 2N3-Q+3
opérations logiques : N3-Q+1
comparaisons : N3-Q

d) Variante :

Une variante de cet algorithme est basée sur une autre définition du dernier couple à traiter :

. On admet que l'un des enregistrements du couple peut être absent parce que le fichier correspondant est épuisé.

. On étend la relation d'ordre entre les couples à ce cas.

Algorithme fusion - version 2

```
ouvrir(f3,E) ; ouvrir(f1,L) ; ouvrir(f2,L)
lire(f1,x1) ; lire(f2,x2)
tantque ¬fdf(f1) ou ¬fdf(f2) faire
  {même invariant que précédemment}
  si ¬fdf(f1) et-c(fdf(f2) ou-c x1<x2)
      alors écrire(f3,x1) ; lire(f1,x1)
      sinon écrire(f3,x2) ; lire(f2,x2)
  fsi
ftantque
fermer(f1) ; fermer(f2) ; fermer(f3)
```

Evaluation :
contrôle : 2N3+1
opérations logiques et, ou : 3N3+1
{si l'on séquentialise l'expression logique contrôlant le *si..alors..sinon* interne, cette quantité devient 2N3+N1-Q+1}
comparaisons : N3-Q.

3.2.3. - Deuxième approche : parcours de l'un des deux fichiers

a) Analyse

On considère que l'on applique un traitement séquentiel à l'un des deux fichiers : recopier le premier (par exemple) fichier en insérant les éléments du second au bon endroit.
La construction de l'invariant part de l'idée suivante : au moment de recopier un élément de f1, tous ceux de f2 qui doivent le précéder l'ont été.

b) Construction

1. accès séquentiel
   défini et décrit au § 3.2.1. : accès à la file *f1*

2. traitement
   On applique un parcours de file (algorithme P1) :
   le traitement d'un élément de *f1* consiste à :
   - recopier tous les éléments de *f2* qui sont plus petits que lui et qui n'ont pas encore été recopiés
   - recopier l'élément de *f1*

   (ceci est traduit par l'invariant porté dans l'algorithme).

c) Algorithme:fusion - version 3

```
ouvrir(f3,E) ; ouvrir(f2,L) ; ouvrir(f1,L)
lire(f1,x1) ; lire(f2,x2)
tantque ¬fdf(f1) faire
  {tous les éléments précédant x1 et x2 sont dans f3. f3 est trié.
   Le dernier élément qui a été recopié est celui prédécant x1 (sauf
   au départ)}
   tantque ¬fdf(f2) et_c x2≤x1 faire
     écrire(f3,x2) ; lire(f2,x2)
   ftantque
   {tout élément de f2 inférieur ou égal à x1 a été recopié}
   écrire(f3,x1) ; lire(f1,x1)
ftantque
tantque ¬fdf(f2) faire écrire(f3,x2) ; lire(f2,x2) ftantque
fermer(f1) ; fermer(f2) ; fermer(f3)
```

Evaluation :
contrôle : 2N1+N2+2
opérations logiques : N1+N2-Q2
comparaisons : N3-Q

d) Variante : version 4

On peut écrire une version 4 en changeant les rôles des fichiers 1 et 2 dans la version 3.

3.2.4. - Application d'une technique de sentinelle

Une technique classique permet de s'assurer que l'exploration des deux fichiers se termine simultanément.

On modifie la spécification de l'ordre de lecture de la manière suivante lorsque *lire(f,x)* provoque l'interruption de fin de fichier, on assure que *x* reçoive une valeur supérieure à toute valeur possible du fichier *f* . Ceci n'est pas réalisable dans tous les langages de programmation (il faut pouvoir programmer la "récupération" de l'interruption). Une alternative consiste à marquer tous les fichiers par une valeur logique supérieure à toute valeur possible.

Dans ce qui suit, on suppose que, quel que soit le moyen employé pour y aboutir, toute exploration des fichiers donnés se termine par la lecture de l'enregistrement de valeur maximum, noté *fictif*.

fusion Technique de sentinelle	
Version 2 bis à partir de la version 2	Version 3 bis
{on applique une technique de sentinelle} *ouvrir(f1,L);ouvrir(f2,L);ouvrir(f3,E)* *lire(f1,x1);lire(f2;x2)* *tantque x1≠fictif ou x2≠fictif faire* *choix* *x1<x2:écrire(f3,x1); lire(f1,x1)* *x1≥x2:écrire(f3,x2); lire(f2,x2)* *fchoix* *ftantque* *écrire(f3,fictif)* *fermer(f1);fermer(f2);fermer(f3)* Evaluation : contrôle : 2N3+1 opérations logiques :N3+1 comparaisons : 3N3+2	{on applique une technique de sentinelle *ouvrir(f3,E);ouvrir(f1,L);ouvrir(f2,L)* *lire(f1,x1) ; lire(f2,x2)* *tantque x1≠fictif faire* *tantque x2≤x1 faire* *écrire(f3,x2); lire(f2,x2)* *ftantque* *écrire(f3,x1); lire(f1,x1)* *ftantque* *tantque x2≠fictif faire* *écrire (f3,x2) ; lire (f2,x2)* *ftantque* *écrire(f3,fictif)* *fermer(f1);fermer(f2);fermer(f3)* Evaluation : contrôle : 2N1+N2+2 comparaisons : 2N1+N2+2

3.2.5. - Discussion

Comme précédemment, on s'est attaché à développer plusieurs solutions au problème posé, en l'interprétant de plusieurs manières en termes d'un traitement de files. Nous avons de plus examiné des conditions particulières d'implantation des files données permettant d'obtenir des nouvelles solutions.

Nous pouvons comparer les diverses versions obtenues :

. du point de vue de la lisibilité, les versions nous paraissent équivalentes On peut toutefois souligner la concision de la version 2 bis.

. si l'on veut comparer les coûts des divers algorithmes, on doit d'abord souligner :

  - qu'ils font tous intervenir le même nombre d'opérations d'entrée-sortie.
  - que la majeure partie du coût provient de ces opérations d'entrée-sortie.

On peut cependant raffiner cette analyse et étudier les coûts un peu plus en détail : nous comparons les algorithmes en pondérant les opérations de la manière suivante (en termes d'une unité de temps fonction de la machine physique utilisée).

. contrôle : 1
. opération logique : 1
. comparaison : 2 (en supposant que les enregistrements sont comparés par l'examen de clés numériques).
. entrée-sortie : x

Dans ces conditions, nous pouvons résumer les évaluations dans le tableau suivant :

version	1	2	3	4	2 bis	3 bis
contrôle	2N3-Q+3	2N3+1	2N1+N2+2	2N2+N1+2	2N3+1	2N1+N2+2
opérations logiques	N3-Q+1	3N3+1	N1+N2-Q2	N1+N2-Q1	N3+1	0
comparaisons	N3-Q	N3-Q	N3-Q	N3-Q	3N3+2	2N1+N2+2
entrée sortie	2N3	2N3	2N3	2N3	2N3	3N3
total (en unités de temps)	5N3-4Q+4 +2N3x	7N3-2Q+1 +2N3x	4N3+N1-Q2 -2Q+2 +2N3x	4N3+N2-Q1 -2Q+2 +2N3x	9N3+6 +2N3x	3N3+3N1+6 +2N3x

En supposant les deux fichiers d'entier de tailles sensiblement égales et en négligeant Q par rapport à N3,on obtient :

version	1	2	3 , 4	2 bis	3 bis
total	$\frac{10N3}{2}$+2N3x	14 $\frac{N3}{2}$ +2N3x	9 $\frac{N3}{2}$ +2N3x	18 $\frac{N3}{2}$ +2N3x	9 $\frac{N3}{2}$ +2N3x

Ceci donne une indication pour une comparaison des coûts des diverses versions.

On peut constater ainsi que l'application de la même technique (sentinelle) conduit d'une part à l'une des deux meilleures solutions (3 bis), d'autre part à la plus mauvaise (2 bis). On peut de plus s'interroger sur la pertinence de ces différences en comparaison du coût global. Estimant le coût des entrées-sorties à 2N3 x, on voit que ces différences restent significatives pour des valeurs relativement faibles de x : par exemple, pour x=5, on obtient un gain de 32 % entre les deux solutions extrêmes, ce gain étant ramené à 4 % pour x=100.

3.2.6. - Conclusion

L'analyse des coûts ne permet pas de conclure (au niveau général où nous nous plaçons) de manière nette ni pour rejeter telle ou telle solution ni pour en sélectionner une. Toutefois, on ne peut affirmer a priori que les différences seront négligeables (notamment notre analyse ne tient aucunement compte de facteurs de blocages ou de techniques utilisant des tampons de taille adéquate situés en mémoire centrale).

Notre souci dans cette discussion est de souligner une fois de plus l'importance, d'un point de vue méthodologique, des techniques qui permettent de produire systématiquement diverses solutions à un problème donné.

Comme précédemment, nous terminerons en remarquant l'intérêt général du problème que nous avons traité : le principe même de la fusion est à la base même de nombreux algorithmes dans le domaine de la gestion de fichiers notamment. Dans [Sch 79 a] nous montrons comment les algorithmes classiques de mise à jour de fichiers, d'inventaires, de facturations peuvent être ramenés à l'algorithme de fusion, ou du moins peuvent être analysés en appliquant le même type de démarche que celle que nous suivons constamment ici.

3.3 - TROISIEME EXEMPLE : TAILLE MOYENNE DES IDENTIFICATEURS D'UN PROGRAMME

Nous voulons, par ce troisième exemple, donner un aperçu de l'application pratique du traitement séquentiel sur un problème concret. A partir du problème de l'analyse lexicographique dans un compilateur (cf.[GoH 74] par exemple), nous avons défini un énoncé, qui bien que simplifié, (pour permettre une présentation dans cette thèse), reflète suffisamment la complexité d'un analyseur lexicographique.

Le problème étudié est le suivant :

On considère un texte correspondant à un programme PL/1 correct. On veut réaliser un programme qui calcule la longueur moyenne des identificateurs de ce texte.

Remarques :

- un programme PL/1 peut comporter des commentaires (délimités par "/*" et "*/") et des littéraux (notations de constantes chaînes, délimités par des apostrophes) : les caractères appartenant à ces commentaires et à ces littéraux ne peuvent pas contribuer à la formation d'un identificateur du texte PL/1.
- Un identificateur est une suite alphanumérique commençant par une lettre.
- Les mots clés PL/1 font partie des identificateurs considérés.

Nous fixons le principe de l'analyse sans détailler les réflexions qui l'ont guidée, préférant insister ici sur les divers choix possibles qui subsistent au niveau de la description de l'accès séquentiel aux files considérées.

3.3.1. - Principe de la construction

1) Analyse

a) On considère la file des identificateurs du texte PL/1 : d'un tel identificateur on ne retient que l'information "longueur de l'identificateur" (en d'autres termes on considère en fait la file des longueurs d'identificateurs). L'algorithme est un traitement séquentiel simple de cette file : calculer la moyenne d'une suite de nombres.

b) Les identificateurs sont reconnus dans la file des caractères *"effectifs"* du texte source.

. un caractère est *"effectif"* s'il n'appartient ni à un commentaire, ni à un littéral

. tout commentaire ou littéral joue le rôle d'un caractère effectif séparateur (espace par exemple).

c) Les caractères effectifs sont reconnus dans le texte source par élimination des commentaires et des littéraux.

Remarque :
Il y a évidemment d'autres possibilités d'analyse. On peut par exemple fonder la construction de l'algorithme sur l'idée d'un traitement séquentiel d'une file de couples de caractères effectifs consécutifs où même directement de la file des caractères effectifs (automate d'états finis). L'analyse ci-dessus a été retenue ici parce qu'elle correspond le mieux aux solutions apportées au problème général dont nous avons tiré notre exemple simplifié.

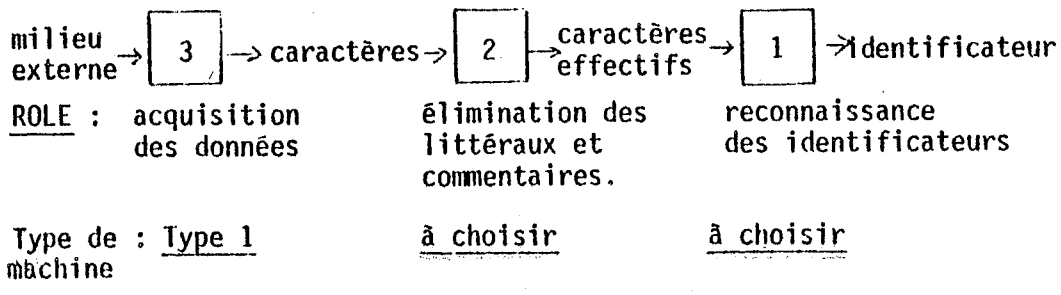
2) Etapes de la construction

On a ainsi mis en évidence trois niveaux d'abstraction :

- identificateurs
- caractères effectifs
- caractères source

et nous allons en déduire les étapes de la construction de notre programme : dans un premier temps, on construit un programme supposant que l'on dispose d'une file (de longueurs) d'identificateurs, accessible à l'aide d'une machine séquentielle (appelée "machine 1"). Dans un deuxième temps, on réalise la machine séquentielle 1 en supposant qu'en fait on dispose de la file de caractères effectifs : celle-ci est rendue accessible par une machine séquentielle (appelée machine 2). Finalement, la machine séquentielle 2 est réalisée en supposant que l'on dispose de la file des caractères sources, accessible par une machine séquentielle (appelée machine 3).

Ce principe de construction du programme est illustré par le schéma suivant :



Nous supposons que les caractères du texte source sont acquis à l'aide d'instructions de lecture dont les spécifications correspondent à notre premier modèle de machine séquentielle (on "récupère" une interruption à la suite d'une lecture suivant le dernier caractère des données).

Notations :

Nous reprenons dans la suite de cet exemple les notations données au § 2.1.2. Pour distinguer les primitives des diverses machines qui interviennent, nous les suffixons par le numéro de la machine auxquelles elles correspondent. Nous appliquons ce principe de notation à la machine 3, sans détailler sa réalisation évidente en termes d'ordres de lecture classiques.

Le schéma ci-dessus fait apparaître les choix que nous avons à faire : quel modèle de machine choisir pour les machines 1 et 2. Nous allons développer les 4 solutions issues des choix possibles, et nous tâcherons de les juger ("lisibilité", "coût").

3.3.2. - Premier niveau d'abstraction : moyenne des longueurs d'identificateurs

a) Construction

Il s'agit d'un simple parcours de file. Le traitement est défini comme suit :

```
accumulateur :
   entier nb      {nombre d'identificateurs rencontrés}
   entier t       {somme des longueurs des identificateurs
                   rencontrés.}
répertoire :
   INIT_P : nb:=0 ; t:=0
   VISITER(L) : nb:=nb+1 ; t:=t+L
   TERM_P : écrire (t/nb)
```

On applique le schéma P1 ou le schéma P2 selon le modèle de définition de la machine 1.

solution au niveau d'abstraction de la machine 1	
Algorithme A1	Algorithme A2
{machine 1 de type 1} *nb:=0 ; t:=0* *DEM1* *tantque ¬FDF1 faire* *nb:=nb+1 ; t:=t+EC1* *AV1* *ftantque* *choix* *nb=0 : écrire("pas d'idf")* *nb≠0 : écrire(t/nb)* *fchoix*	{machine 1 de type 2} *choix* *FILE_VIDE : écrire("pas d'idf")* *¬FILE_VIDE:* *nb:=0 ; t:=0* *DEM1* *itérer nb:=nb+1 ; t:=t+EC1* *arrêt : FDF1* *AV1* *fitérer* *écrire(t/nb)* *fchoix*

3.3.3. - Deuxième niveau d'abstraction : constitution des identificateurs à partir des caractères effectifs.

La machine 1 est réalisée en termes des primitives de la machine 2 : on calcule les longueurs des identificateurs successifs à partir de la donnée de la file des caratères effectifs.

Le principe de l'acquisition des identificateurs est basé sur le fait que deux identificateurs sont séparés par un groupe de séparateurs, et qu'un identificateur est un groupe de caractères alphanumériques commençant par une lettre : l'acquisition d'un identificateur à l'aide de AV1 provoque l'acquisition d'un certain nombre de caractères effectifs à l'aide de AV2. Ceux-ci correspondent aux caractères de l'identificateur et aux caractères séparateurs le précédant (ou le suivant). Ceci est une technique générale que nous avons étudiée par ailleurs [LuS 75].

L'analyse fait apparaître deux primitives :

- une action qui permet de rechercher le premier caractère d'un identificateur à partir d'une position dans la file des caractères effectifs :

  CHERCHE_IDF

- une action qui "lit" un identificateur et calcule sa longueur, sachant que l'on est positionné au départ sur le premier caractère de l'identificateur :

  CALCUL_LONG

La combinaison de ces deux actions permet d'acquérir un groupe de séparateurs et l'identificateur associé et l'algorithme que nous construisons va engendrer une alternance d'appels de ces deux actions. On peut donc en préciser les spécifications :

. *CHERCHE_IDF*

état_initial : On se trouve au début de la file de caractères effectifs, ou sur un caractère suivant un identificateur.

état_final : on se trouve sur le premier caractère d'un identificateur ou à la fin de la file de caractères effectifs. Il s'agit d'une recherche associative (§ 2.2.4.).

. *CALCUL_LONG*

état_initial : On se trouve sur le premier caractère d'un identificateur

état_final : On se trouve sur le caractère suivant l'identificateur ou à la fin de la file de caractères effectifs, et on a calculé la longueur de l'identificateur. Il s'agit d'un parcours d'une sous-file définie par exclusion (§ 2.2.3.).

Nous développons maintenant les deux solutions correspondant aux deux choix possibles pour la spécification de la machine 1.

Remarque :

Dans ce qui suit, nous utilisons deux prédicats portant sur des caractères :

. *ALPHANUM(C)* est *vrai* si *C* est alphanumérique, *faux* sinon

. *ALPHA(C)* est *vrai* si *C* est alphabétique, *faux* sinon

a) La machine 1 est définie selon le premier modèle de machine séquentielle

Cette spécification implique que dans le cas général, on ne peut savoir si un identificateur est le dernier qu'en cherchant à trouver son successeur. Par conséquent l'état final de *AV1* est tel qu'on se trouve sur le caractère effectif suivant l'identificateur que l'on vient de lire. Lorsque le texte se termine par un identificateur, on peut alors se trouver à la fin de la file des caractères effectifs : cette fin sera détectée par le prochain appel de *AV1*. Cette spécification correspond donc à une progression le long de la file de caractères qui fait lire dans l'ordre un groupe de séparateurs puis un identificateur. La réalisation de *AV1* et de *DEM1* tient compte de l'existence possible de séparateurs en début et en fin de la file de caractères effectifs.

Réalisation de la machine 1 :

. Etat : . l'élément courant *EC1* est un entier qui correspond à la longueur de l'identificateur courant. Il est représenté par une variable entière notée simplement *L* dans les algorithmes qui suivent.

. la détection de la fin de la file d'identificateurs a pour effet de positionner une variable logique *final1*

. primitives : Elles sont rédigées en termes des primitives de la machine 2. On a donc deux réalisations possibles de la machine 1 selon que la machine 2 est spécifiée selon le modèle 1 ou le modèle 2 de machine séquentielle. Le principe général de *AV1* est le suivant :

```
{on est sur le caractère effectif suivant l'identificateur
 courant ou c'est la fin}
si "c'est la fin"
   alors final1:=vrai
   sinon CHERCHE_IDF
         si "on a trouvé un identificateur"
            alors CALCUL_LONG
            sinon final1:=vrai
         fsi
fsi
```

En ce qui concerne *DEM1*, nous retenons la réalisation suivante :

DEM1 : DEM2 ; final1:=faux ; AV1

Remarque :
Ceci est une simplification dans l'hypothèse où le premier caractère "effectif" est un séparateur. Cette hypothèse ne nous gêne pas dans la mesure où l'on peut toujours "simuler" ce séparateur au niveau de la réalisation de la machine 2.

On a ainsi mis en évidence une action d'initialisation de l'accès séquentiel à la file des identificateurs :

INIT_AS1 : DEM2 ; final1:=faux

La structure de l'algorithme A1 est modifié en tenant compte de cette remarque (cf. § 2.3.1.).
On obtient ainsi les deux algorithmes A11 (machine 1 de type 1 et machine 2 de type 1) et A12 (machine 1 de type 1 et machine 2 de type 2).

Remarque :
Nous avons porté en face de chacune des instructions de ces algorithmes, le nombre de leurs exécutions, avec les conventions suivantes

N : nombre d'identificateurs

X1 : nombre total des caractères des identificateurs

X2 : nombre de "caractères effectifs" séparateurs (1 commentaire ou un littéral compte pour 1 caractère effectif séparateur)

X : nombre total de caractères effectifs (X=X1+X2)

$\varepsilon 1$: 0 si le dernier caractère est un séparateur
1 sinon

$\varepsilon 2$: 0 ou 1 et $\varepsilon 1 + \varepsilon 2 = 1$

$\varepsilon 3$: 1 si le texte se termine par un identificateur de longueur 1
0 sinon

b) La machine 1 est définie selon le second modèle de machine séquentielle

Cette spécification implique que l'on sait, à l'issue d'un appel de AV1, si on vient de lire ou non le dernier identificateur : à chaque identificateur est associé le groupe de séparateurs qui le suit, et l'état final de AV1 est tel que l'on se trouve dans le cas général sur le premier caractère de l'identificateur suivant.

Réalisation de la machine 1 :

. Etat : . comme précédemment, l'identificateur courant est représenté par un entier *L*.

. l'identificateur courant est le dernier de la file dans la mesure où on a atteint la fin de la file de caractères effectifs. Si la machine 2 est de type 1, ceci se traduit par *FDF2=vrai*. Si la machine 2 est de type 2, ceci se traduit par *FDF2=vrai et ALPHA(EC2)=faux*

. Primitives : comme précédemment, on a deux réalisations possibles suivant les spécifications de la machine 2.

- le schéma général de *AV1* est le suivant :

```
{EC2 est le premier caractère de l'identificateur à lire}
CALCUL_LONG
si "ce n'est pas la fin" alors CHERCHE_IDF fsi
```

- en ce qui concerne *DEM1* , on a le schéma général suivant :

```
        {on suppose que la file de caractères effectifs commence
         toujours par un séparateur}
DEM2 ; CHERCHE_IDF
si "c'est la fin"
   alors "la file est vide
   sinon AV1
fsi
```

Si l'on replace ces définitions de AV1 et DEM1 dans l'algorithme A2, on peut en simplifier la forme comme suit :

Algorithme A2bis

```
        nb:=0 ; t:=0
        itérer CHERCHE_IDF
        arrêt :"c'est la fin"
               CALCUL_LONG
               nb:=nb+1 ; t:=t+L
        fitérer
        si nb=0
           alors écrire ("pas d'idf")
           sinon écrire (t/nb)
        fsi
```

C'est à partir de cette forme de l'algorithme que nous obtenons les algorithmes A21 et A22.

c) Algorithmes

Algorithme A11 (machine 1 de type 1, machine 2 de type 1)

```
1     nb:=0, t:=0
1     DEM2 ; final1:=faux
      itérer
N+1           {FDF2 ou ¬alphanum(EC2)}
N+1       si FDF2
ε1           alors final1:=vrai
             sinon {RECHIDF1 : recherche associative, algorithme RA1}
                    répéter
X2+1                   AV2
X2+1                jusqua FDF2 ou alpha (EC2)
N+ε2                si FDF2
ε2                     alors final1:=vrai
                       sinon {CALCUL_LONG1 : parcours d'une sous-file
                                             algorithme PEX1-bis}
N                                 ℓ := 1
X1                                itérer AV2
X2                                arrêt FDF2 ou ¬alphanum(EC2)
X1-N                                   ℓ :=ℓ+1
                                  fitérer
                    fsi
          fsi
N+1   arrêt : final1
N         nb:=nb+1 ; t:=t+ℓ
      itérer
1     choix
          nb=0 : écrire ("pas d'identificateur")
          nb≠0 : écrire t/nb
      fchoix
```

Evaluation :

machine 2 : *DEM2* :1, *AV2*:X+1, *EC2*:X+1, *FDF2*:2N+X+2+ε2
prédicats : *ALPHANUM*:X1, *ALPHA*:X2+1
opérations logiques : X+1
comparaisons : 1
opérations arithmétiques : "+":N+X1, "/":1
affectations : 2N+X1+4
contrôle:3N+X+2+ε2

Algorithme A12 (machine 1 de type 1, machine 2 de type 2)

```
1       nb := 0 ; t := 0
1       DEM2 ; final1:=faux
        itérer
N+1         {AV1  : FDF2 ou ¬alphanum(EC2)}
N+1         si FDF2
ε1             alors final1:=vrai
N+ε2           sinon {RECHIDF2 : recherche associative RA2}
                      {alphanum(EC2) et ¬FDF2}
X2+1-ε2               répéter AV2
X2+1-ε2               jusqua FDF2 ou alpha(EC2)
N+ε2                    si ¬alpha(EC2)
ε2                         alors final1:=vrai
                           sinon {CALCUL_LONG2 : parcours de sous-file définie
                                                 par exclusion - PEX2}
N                                 ℓ :=1
N                                 si ¬FDF2 alors
                                     itérer
                                          AV2
X1-ε1                                arrêt : FDF2 ou ¬alphanum(EC2)
X1-ε1                                        ℓ :=ℓ+1
X1-N-ε1+ε3                           fitérer
                                     si alphanum(EC2) alors
N-ε3                                    ℓ :=ℓ+1
ε1-ε3                                fsi
                                  fsi
                        fsi
            fsi
               {FDF2 ou ¬alphanum(EC2)}
N+1     arrêt : final
N           nb :=nb+1 ; t :=t+ℓ
        fitérer
1       si nb=0
            alors écrire ("pas d'identificateur")
            sinon écrire (t/nb)
        fsi
```

Evaluation

machine 2 : *DEM2*:1, *AV2*:X, *EC2*:2N+X+1-ε1-ε3, *FDF2*:2N+X+1
prédicats : *ALPHANUM*:N+X1-ε1-ε3, *ALPHA* :N+X2+1
opérations logiques : X
comparaison : 1
opérations arithmétiques : "+" : N+X1, "/":1
affectations : 2N+X1+3
contrôle : 5N+X+ε2-ε3

Algorithme A21 (machine 1 de type 2, machine 2 de type 1)

```
1       nb:=0 ; t:=0
1       DEM2
        itérer
N+1        { FDF2 ou ¬alphanum(EC2)}
           si ¬FDF2 alors
           {CHERCHE_IDF2 : recherche associative-RA2}
              répéter
X2+1             AV2
X2+1          jusqua FDF2 ou alpha(EC2)
           fsi
N+1     arrêt  : FDF2
           {CALCUL_LONG : parcours d'une sous-file définie par exclusion-PEX1 bi
N           ℓ:=1
            itérer
X1               AV2
X1          arrêt : FDF2 ou ¬alphanum(EC2)
X1-N              ℓ:=ℓ+1
            fitérer
N           nb:= nb+1 ; t:= t+ℓ
        fitérer
1       choix
           nb=0 : écrire ("pas d'identificateur")
           nb≠0 : écrire (t/nb)
        fchoix
```

Evaluation :
machine 2 : *DEM2*:1, *AV2*:X+1, *EC2*:X+1, *FDF2*:2N+X+3
prédicats : *ALPHANUM* :X1, *ALPHA*:X2+1
opérations logiques : X+1
comparaisons : 1
opérations arithmétiques : "+":N+X1, "/":1
affectations : 2N+X1+2
contrôle : 2N+X+4

Algorithme A22 : (machine 1 de type 2, machine 2 de type 2)

```
                {le premier caractère effectif est un séparateur}
1               nb:=0 ; t:=0
1               DEM2 ; final1:=faux
                itérer
                   {FDF2 ou ¬alphanum(EC2)}
N+1                si FDF2
ε1                    alors final1:=vrai
                      sinon {RECHIDF2 : recherche associative-algorithme RA2}
                            {¬alphanum(EC2) et ¬FDF2}
X2+1-ε2                      répéter AV2
X2+1-ε2                      jusqua FDF2 ou alpha(EC2)
N+1-ε1                       si ¬alpha(EC2) alors
ε2                              final1:=vrai
                             fsi
                   fsi
N+1             arrêt : final1
                   {CALCUL_LONG2 : parcours d'une sous-file définie par exclusion
                    algorithme PEX2}
N                   ℓ:=1
N                   si ¬FDF2 alors
                        itérer
                             AV2
X1-ε1                   arrêt : FDF2 ou ¬alphanum(EC2)
X1-ε1                         ℓ:=ℓ+1
X1-N-ε1+ε3              fitérer
                        si alphanum (EC2) alors
N-ε3                         ℓ
ε1-ε3                   fsi
                    fsi
N                   nb   nb ; t:=t+ℓ
                fitérer
1               si nb=0
                    alors écrire ("pas d'identificateurs")
                    sinon écrire (t/nb)
                fsi
```

Evaluation :
machine 2 : *DEM2*:1, *AV2*:X, *EC2*:3N+X+2-ε3, *FDF2*:2N+X
précidats : *ALPHANUM*:N+X1-ε1-ε3, *ALPHA*:N+X2+1
opérations logiques : X
comparaisons : 1
opérations arithmétiques : "+", N+X1, "/":1
affectaions : 2N+X1+4
contrôle : 5N+X+4-ε1-ε3

d) Comparaison des algorithmes

Du point de vue de la "forme", les algorithmes construits avec une machine 2 de type 1 semblent plus simples que les autres. Du point de vue des coûts, les différences tiennent en deux points :
. l'utilisation des primitives *ALPHA* et *ALPHANUM*.
. le nombre de tests de contrôle
et viennent corroborer l'impression de structure plus compliquée des algorithmes A12 et A22 par rapport aux autres.
Par conséquent, si le choix de la machine 2 n'influe pas sur le coût global, on préfèrera l'algorithme A21. Par contre, s'il est nécessaire de recourir à une machine 2 de type 2, A12 et A22 sont équivalents tant du point de vue des coûts que de la lisibilité.

3.3.4. - Troisième niveau d'abstraction : élimination des commentaires et des littéraux

La machine 2 produit des caractères "effectifs" à partir du texte source : son rôle est de reconnaître les commentaires et les littéraux et de les assimiler à un séparateur. Comme nous l'avons dit ci-dessus, nous avons deux choix possibles pour sa réalisation : nous étudions ces deux solutions et les comparons à la lumière des résultats précédents.

a) élimination des commentaires et des littéraux

La syntaxe PL/1 est telle que ces éléments lexicographiques sont reconnus dès que l'on trouve une certaine configuration de caractères. Leur élimination est réalisée à l'aide des deux primitives suivantes :

```
ELIMINER_COMMENTAIRE :
      {¬FDF3 et EC3="*" et on vient de trouver un "/"}
      {le texte PL/1 est supposé correct}
      {il s'agit d'une recherche associative du couple "*/" dans la file
       des couples de caractères consécutifs. Le texte étant supposé correct,
       on sait que cette recherche sera positive.
       On applique RA3}
      AV3
      répéter prec:=EC3 ; AV3
      jusqua prec="*" et EC3="/"
      AV3
      {on est sur le caractère suivant le commentaire, ou c'est la fin}.
```

Remarque :
Si le texte PL/1 n'est pas correct, la détection de l'erreur "délimiteur de fin de commentaire manquant" est simplement réalisée en appliquant le schéma de recherche associative dans sa forme générale.

```
ELIMINER_LITTERAL :
    {¬FDF3 et EC3=apostrophe - le texte PL/1 est supposé correct}
    {on applique le schéma de recherche associative RA3}
    {même remarque que ci-dessus, en ce qui concerne le traitement d'erreurs}
    répéter AV3 jusqua EC3=apostrophe ; AV3
    {on est sur le caractère suivant le littéral, ou c'est la fin}
```

b) Réalisation d'une machine 2 de type 1

- état : . l'élément courant est un caractère. Nous le représentons à l'aide d'une variable de type caractère appelée *courant2*
  . le "voyant" est représenté par une variable logique appelée *final2* : *final2=vrai* indique qu'on a atteint la fin de la file de caractères effectifs ("voyant allumé").

- primitives :
  . la consultation de l'état de la machine est simplement réalisée par une utilisation des variables *courant2* et *final2*
  . la modification de l'état de la machine est réalisée comme suit :

```
. AV2 :
  {FDF3 ou "EC3 est le premier caractère source non traité"}
  choix
    ¬FDF3 : final2:= vrai
     FDF3 : choix
                EC3='/' : AV3
                          si ¬FDF3 et EC3='*'
                             alors éliminer-commentaire
                                   courant2:=blanc
                             sinon courant2:='/'
                          fsi
                EC3=apostrophe : éliminer-littéral
                                 courant2:=blanc
                autrement:courant2:=EC3
                          AV3
            fchoix
  fchoix
```

. *DEM2*
{on veut toujours avoir un caractère séparateur au départ}
DEM3
courant2:=blanc ; final2:=faux

Evaluation : nous nous plaçons dans le contexte de l'algorithme A21, et évaluons le coût du aux primitives de la machine 2. Nous notons ND, NC et NL le nombre, respectivement, d'opérateurs "/", de commentaires et de littéraux.
affectations : X+3
comparaisons : 2X (en supposant les *choix* séquentialisés dans l'ordre du texte)
contrôle : 3X+1
opérations logiques : NC+ND
primitives : *ELIMINER_COMMENTAIRE*:NC, *ELIMINER _LITTERAL*:NL

c) Réalisation d'une machine 2 de type 2

On met à profit le fait que la machine 3 est toujours en avance d'un caractère sur la machine 2 (i.e. à l'issue de *AV2* , *EC3* est le caractère suivant) : le caractère effectif courant est le dernier si et seulement si on a atteint la fin du texte source. La réalisation de *AV2* est simplifiée en conséquence : le test initial sur *FDF3* disparaît.

. *AV2 :*
{¬*FDF3* et "*EC3* est le premier caractère du prochain caractère effectif"}.
choix EC3="/":... (cf. ci-dessus)
EC3=apostrophe:...
autrement :...
fchoix
{*courant2* est le dernier caractère effectif si et seulement si *FDF3=vrai*}

. *DEM2 :*
DEM3 ; courant2:=blanc

Evaluation :
Par rapport à la solution précédente, on gagne 1 test de contrôle par appel de *AV2*
Dans ces conditions, ceci ne compense l'avantage de la solution A21 par rapport aux solutions A12 et A22 que si $X \leq 4N$.

d) Algorithme final
En conclusion, l'algorithme A21 semble être le meilleur dans la majeure partie des cas.
L'algorithme final que nous présentons correspond à la version A21, avec quelques modifications supplémentaires : notamment, on incrémente directement le compteur général *t* au lieu de calculer la taille de chaque identificateur.

Algorithme : taille moyenne des identificateurs d'un texte PL/1

```
nb:=0 ; t:=0
DEM3 ; courant2:=blanc ; final2:=faux
itérer
      {final2 ou ¬ALPHANUM(courant2)}
      si final2 alors
         {RECHIDF :¬final2 et ¬ALPHANUM}
         al := faux
         répéter{AV2}
            choix
               FDF3:final2:=vrai
               ¬FDF3:choix
                         EC3='/' : traiter_slash
                         EC3= apostrophe : traiter-apostrophe
                         autrement : courant2:=EC3 ; AV3
                                     al:=apha(courant2)
                     fchoix
            fsi
          jusqua final2 ou al
      fsi
arrêt : final2
       {CALCUL_LONG  : ¬final2 et ALPHA(courant2)}
        nonal:= faux
        répéter
             t:=t+1
             {AV2}
             si FDF3
                alors final2 := vrai
                sinon choix
                         EC3='/' : traiter-slash ; nonal := vrai
                         EC3=apostrophe ; traiter-apostrophe; nonal := vrai
                         autrement : courant2 := EC3 ; AV3
                                     nonal :=non-alphanum (courant2)
                     fchoix
             fsi
        jusqua final2 ou nonal
        nb :=nb+1
fitérer
choix
      nb=0 : écrire ("pas d'identificateur")
      nb≠0 : écrire (t/nb)
fchoix
```

```
avec

   . traiter-slash
          AV3
          si ¬FDF3 et EC3="*"
              alors
                    AV3
                    répéter
                          prec:=EC3 ; AV3
                    jusqua prec='*" et EC3='/'
                    AV3
                    courant2:=blanc
              sinon courant2:='/'
          fsi

   .  traiter-apostrophe
          répéter
                    AV3
          jusqua EC3=apostrophe
          AV3 ; courant2:=blanc
```

3.3.5. - Discussion

L'exemple que nous venons de présenter, nous a permis de montrer les aspects méthodiques de l'application du traitement séquentiel : l'application systématique des résultats du chapitre 2 permet d'une part de guider l'analyse du problème et de mettre en évidence les choix que l'on peut faire, d'autre part d'apporter des solutions fiables par l'utilisation quasi immédiate des schémas de traitement séquentiel. Nous avons pu ainsi construire plusieurs solutions au problème posé, les évaluer et les comparer.

Cet exemple est de plus représentatif de nombreux problèmes pris dans les domaines les plus variés : nous avons appliqué la même démarche avec succès à d'autres traitements de texte (problème du télégramme [HeS 72] problème de "références croisées" [Sch 77a], éditeur de texte, analyse lexicographique,...), mais aussi à des problèmes de gestion [Sch 79a], d'analyse de données numériques.

3.4 - CONCLUSION

Les exemples présentés ont permis de montrer l'application directe de ce qui a été proposé dans les deux premiers chapitres : reconnaissance des algorithmes fondamentaux et utilisation des règles de programmation. Chaque exemple est en lui même un élément supplémentaire dans notre présentation du traitement séquentiel :

. Le problème de l'énumération partielle conduit à un ensemble de schémas moins faciles à utiliser directement que les schémas du chapitre 2, parce que moins élémentaires (on peut d'ailleurs interpréter tous les algorithmes fondamentaux du chapitre 1 comme des cas particuliers du problème d'énumération partielle).

. Le deuxième exemple , la fusion,laisse entrevoir comment l'on peut adapter notre propos à une application particulière : ce travail (détaillé dans [Sch 79a]) comporte la définition de notations qui reflètent le traitement séquentiel et les notations couramment utilisées dans l'application, et l'élaboration de schémas correspondant aux problèmes les plus fréquemment rencontrés dans l'application.

. Le calcul de la taille moyenne des identificateurs est l'exemple le plus concret. Comme nous l'avons souligné, il est représentatif de nombreux problèmes pris dans des domaines divers. Nous avons pu montrer que dans un problème d'une complexité certaine, nous pouvons avoir une attitude méthodique et systématique, et ceci grâce à l'ensemble des résultats que fournit le traitement séquentiel (que ce soit du point de vue de la démarche ou du point de vue de l'utilisation des schémas).

Nous avons fait apparaître que le traitement séquentiel permet de construire méthodiquement et systématiquement plusieurs solutions à un problème posé. Ceci est une qualité :

. Au niveau de l'élaboration de schémas généraux. Nous avons pu constater que, à un tel niveau d'abstraction, il est difficile de définir une solution optimale, même si l'on peut donner des évaluations (qui sont des éléments de comparaison et qui reflètent le comportement des algorithmes).

. A un niveau pratique, dès lors que l'on cherche pour un problème spécifique, la solution la plus efficace (quel que soit le critère). L'obtention systématique de solutions variées est un élément indispensable pour une telle étude.

Inversement, nous devons souligner que d'autres contraintes pratiques (temps disponible pour la réalisation du programme, coût de la mise au point) peuvent interdire de rechercher aussi systématiquement les solutions d'un problème. Clairement, c'est à ce niveau que l'"expérience" du programmeur doit intervenir. Le traitement séquentiel est alors utilisé a postériori pour justifier le choix d'une voie particulière dans la construction du programme. Nous essayons toutefois dans notre travail méthodologique de dépasser ce facile recours à l'expérience ou à l'intuition (notions empiriques par excellence), en proposant des lignes générales de conduite qui permettent d'aborder un problème sans se heurter à une explosion combinatoire du nombre de solutions, puis de construire une solution de manière systématique. Une telle démarche est proposée dans [Sch 78 b] pour traiter les problèmes complexes de traitement séquentiel. cf. aussi [Jack 75], [Hug 79], [Fin 79]).

CHAPITRE 4

ANALYSE RECURRENTE ET TRAITEMENT SEQUENTIEL

L'application du traitement séquentiel repose sur la définition d'une file et d'un traitement de cette file. A partir de ces données, on construit systématiquement un ou plusieurs algorithmes. Dans ce qui précède, nous avons particulièrement insisté sur une façon de faire cette construction : dans le cadre général d'une méthode d'analyse descendante, on cherche constamment à identifier des problèmes connus sur les files et à appliquer les algorithmes correspondants. Clairement, lorsque une telle identification semble impossible, on peut toujours recourir à d'autres méthodes d'analyse telles que manipulation d'assertions, ou analyse récurrente. Ce type d'analyse a souvent pour effet de mettre en évidence la file ou le traitement que l'on n'avait pu découvrir directement. Nous les considérons donc comme tout à fait complémentaire des méthodes que nous présentons.

C'est à l'analyse récurrente que nous nous intéressons ici. Nous avons déjà vu au chapitre 1 comment l'utiliser pour construire un algorithme de traitement séquentiel, notamment en se basant sur la définition récurrente d'une file. On est alors conduit naturellement à utiliser une formulation récursive pour décrire les algorithmes. Nous posons la question du passage à une formulation itérative d'un tel algorithme, quelle qu'en soit la raison (contraintes de l'environnement de programmation notamment, rendant la récursivité inefficace ou même l'interdisant).

Cette question a vite été soulevée à la suite de l'introduction des formalismes récursifs ([Mac 60] ,[Mac 61]) et de premiers résultats sont présentés dans [Coo 66]. Plus récemment, le développement de la réflexion sur la programmation a conduit à des études plus systématiques du passage "récursif-itératif" ([Rob 78]) : en tant que tel ([Vei 76], [PaP 76], [Bir 77],[ArK 79]) ou dans un contexte plus général de construction de programmes par transformations successives ([Bau 76], [Gri 76a] ,[Ars 77], [BuF 77]) ou de manipulation automatique de programmes ([Dar 72] ,[DaB 76] , [Cha 77] , [PPW 78] [Ars 79]).

Nous allons voir comment le traitement séquentiel peut être appliqué pour transformer certains algorithmes récursifs

(ceux dans lesquels un appel peut engendrer au plus un appel récursif) en algorithmes itératifs. Nous compléterons la présentation de ce procédé de transformation dans la deuxième partie, en application du traitement arborescent (cf. chapitre 7).

Remarque :
Le procédé a été présenté pour la première fois dans [Sch 76] où l'on trouvera de nombreux exemples.

4.1 - FORME GENERALE DES ALGORITHMES CONSIDERES

Nous considérons des algorithmes récursifs correspondant au schéma général suivant :

```
action AC(param P)
  choix
       A(P):ALPHA(P)
       ¬A(P):BETA(P) ; AC(T(P)) ; GAMMA(P)
  fchoix
faction AC
```

et nous considérons la séquence d'appel suivante :

PROLOGUE ; AC(X) ; EPILOGUE

Dans ce schéma, nous employons les conventions suivantes :

- *P* désigne l'ensemble des paramètres de l'action *AC*
- *param* désigne l'ensemble des types de ces paramètres
- *A* est un prédicat dont la valeur ne dépend que de l'examen du paramètre *P*
- *T* est une fonction qui permet de calculer une nouvelle valeur de type *param* à partir de la donnée de *P*.
- *ALPHA*, *BETA* ,*GAMMA* sont des actions dont l'effet dépend de *P* mais qui ne le modifient pas. Leur effet est spécifié en fonction d'un ensemble de variables externes à l'action.

Remarque :
Rappelons que le "passage des paramètres" d'une *action* ou d'une *fonction* est toujours fait par "valeur" dans notre notation.

4.2 - FILE ASSOCIEE AL'APPEL D'UN ALGORITHME RECURSIF

Nous considérons l'appel initial *AC(X)* : il engendre un nombre fini d'appels de *AC* . C'est cet ensemble d'appels que nous organisons en file :

. à l'appel initial correspond le premier élément de la file
. les autres appels sont organisés de la même manière en file : ils constituent la sous-file des appels issus du premier appel.
Cette sous-file est vide si *A(X)=vrai* (*AC(X)* étant le premier appel).

Cette définition récurrente de la file montre qu'un élément de cette file peut être caractérisé par la valeur du paramètre passé lors de l'appel correspondant. Nous pouvons donc définir la file comme suit :

. un élément est une valeur de type *param*
. les primitives qui définissent l'accès à la file sont :

PRE : *X*
SUC(P): *T(P)*
E_S(P): $\neg A(P)$

au niveau d'abstraction où nous nous plaçons, on ne peut dire si la file inverse est définie ou pas, c'est-à-dire si on peut donner une signification directe ou non aux primitives *DERNIER* et *PRED* (cf. § 1.1.3.). La définition de *PRED* est liée à l'existence de la fonction inverse T^{-1} . Nous traiterons ce problème en détail un peu plus loin.

4.3 - TRAITEMENT SEQUENTIEL ASSOCIE A UN ALGORITHME RECURSIF

Nous interprétons l'effet de l'appel *AC(X)* (quel que soit *X* dans le domaine de validité fixé par l'action *AC*) comme étant la succession de deux parcours de file.

. Le premier est un parcours de la file des appels dans lequel on a :

INIT_PARCOURS : *vide*
VISITER(P) : *si A(P) alors ALPHA(P) sinon BETA(P) fs*
TERM_PARCOURS : *vide*

. Le second est un parcours de la file inverse des appels dans lequel on a :

```
INIT_PARCOURS  : vide
VISITER(P)     : si ¬A(P) alors GAMMA(P) fsi
TERM_PARCOURS  : EPILOGUE
```

L'algorithme itératif s'écrit (on applique le schéma P2) :

Algorithme : traitement séquentiel équivalent à l'appel AC(X) (forme générale)

```
{Y est une variable de type param}
PROLOGUE
Y:=X
tantque ¬A(Y) faire BETA(Y) ; Y :=T(Y) ftantque
ALPHA(Y)
tantque E_P(Y) faire Y:=PRED(Y) ; GAMMA(Y) ftantque
EPILOGUE
```

Remarque :

. La preuve de cette équivalence peut être faite par récurrence sur la taille de la file (i.e. le nombre d'appels récursifs)

. le nombre d'exécutions de la première itération est le même que celui de la deuxième. On peut tirer parti de cette remarque en proposant un schéma dans lequel :
- on compte le nombre d'itérations du premier parcours
- on contrôle le deuxième parcours à l'aide de ce compteur

```
PROLOGUE ;
Y:=X ; C:=1;
tantque ¬A(Y) faire BETA(Y) ; Y:=T(Y) ; C:=C+1 ftantque
ALPHA(Y) ;
tantque C≠1 faire C:=C-1 ; Y:=PRED(Y) ; GAMMA(Y) ftantque
EPILOQUE
```

. Nous supposons que le cas des algorithmes comportant des variables locales est inclu dans le schéma général précédent par la convention suivante : on peut supprimer les variables locales par l'introduction de piles gérées de manière classique au niveau des actions BETA et GAMMA.

. On peut enfin considérer des actions récursives sans paramètres. Dans ce cas, le contrôle de terminaison dépend directement de l'état du programme dans lequel se situe l'appel récursif. L'équivalence devient :

```
action AC
  si A alors ALPHA sinon BETA ; AC ; GAMMA
faction
```

l'appel : *PROLOGUE; AC; EPILOGUE* a le même effet que :

```
PROLOGUE
C:=1 ; tantque ¬A faire BETA ; C:=C+1 ftantque
ALPHA
tantque C≠1 faire GAMMA ; C:=C-1 ftantque
EPILOGUE
```

4.4. - REALISATION DE LA PRIMITIVE PREDECESSEUR

Nous examinons ici le cas où la file inverse n'est pas définie : l'application T^{-1} n'est pas définie. Pour réaliser la primitive PRED, nous avons recours à une pile, gérée à l'aide des primitives classiques *VIDER_PILE*, *EMPILER*, *DEPILER*. Le schéma devient alors le suivant :

Algorithme : traitement séquentiel équivalent à l'appel *AC(X)* - version 2

{la fonction inverse T^{-1} n'est pas définie, on utilise une pile pour la réaliser}.

```
PROLOGUE
VIDER_PILE ; Y:=X
tantque ¬A(Y) faire BETA(Y) ; EMPILER(Y) ; T:=T(Y) ftantque
ALPHA(Y)
tantque ¬PILE_VIDE faire Y:=DEPILER ; GAMMA(Y) ftantque
EPILOGUE
```

4.5 - CAS PARTICULIERS

Nous proposons ici les schémas correspondants à quelques cas particuliers particulièrement fréquents :

a) Cas où la liste inverse est entièrement définie :

la fonction T^{-1} est définie, et de plus on connaît le dernier élément de la file. Enfin l'action *BETA* est vide : la première itération disparait.

Algorithme : traitement séquentiel équivalent à l'appel *AC(X)* - version 3

{BETA est vide, PRED et DER sont définies}

```
PROLOGUE
Y:=DER ; ALPHA(Y)
tantque E_P(Y) faire Y:=T^-1(Y) ; GAMMA(Y) ftantque
EPILOGUE
```

b) cas où GAMMA est vide :

l'algorithme se réduit au premier parcours de la file. On a l'équivalence classique suivante :

algorithme : traitement séquentiel équivalent à l'appel *AC(X)* - version 4

```
{GAMMA est vide}
PROLOGUE
Y:=X ; tantque ¬A(Y) faire BETA(Y) ; Y:=T(Y) ftantque
ALPHA(Y)
EPILOGUE
```

4.6 - FONCTIONS RECURSIVES

Nous considérons des fonctions récursives ayant la forme générale suivante (la plupart des fonctions récursives ne comportant qu'un seul appel récursif, se ramènent à cette forme) :

```
fonction F(param P) → résult :
   choix
       A(P) : → H(P)
       ¬A(P): → G(P,F(T(P)) )
   fchoix
ffonction F
```

et nous considérons la séquence d'appel suivante :

```
PROLOGUE ; R:=F(X) ; EPILOGUE
```

Les conventions sont les suivantes :

- *param*, *P*, *A*, *T* ont la même signification qu'en 4.1
- *result* est le type du résultat (on admet des fonctions ayant un n-uplet de résultats).
- *H* est une fonction à un paramètre de type *param* calculant une valeur de type *result*
- *G* est une fonction à deux paramètres, le premier de type *param* , le second de type *result*, qui calcule une valeur de type *result*.

Pour transformer cette fonction sous forme itérative, nous commençons par définir une action *AF* qui pour X donné calcule *F(X)* et range cette valeur dans une variable *Z* :

```
action AF(param P)
  {calcule F(P) et range cette valeur dans une variable Z }
  choix
       A(P)   : Z:=H(P)
       ¬A(P) : AF(T(P)) ; Z:=G(P,Z)
  fchoix
faction AF
```

la séquence d'appel est alors :

```
PROLOGUE ; AF(X) ; R:=Z ; EPILOGUE
```

La transformation de l'action *AF* découle alors des résultats précédents dans la mesure où l'on identifie facilement le schéma général du § 4.1 : on peut donner la signification suivante aux divers éléments du schéma :

ALPHA(P) : *Z:=H(P)*
BETA(P) : *vide*
GAMMA(P) : *Z:=G(P,Z)*
EPILOGUE : *R:=Z ; EPILOGUE*

remarque :
Nous n'abordons pas ici la question des transformations particulières des fonctions vérifiant certaines propriétés sémantiques (notamment l'associativité de la fonction G, ou des formes particulières de cette fonction). On trouvera une étude systématique de ces transformations particulières notamment dans [Coo 66], [Dar 72], [BPP 76], [Ars 77]

4.7. EXEMPLE :

Nous traitons ici l'exemple du calcul du reste et du quotient de la division entière de deux entiers positifs.

a) Analyse

Une première analyse est basée sur les relations suivantes :

$\{A,B > 0\}$

$A < B$: $reste(A,B) = A$
$quotient(A,B) = 0$

$A > B$: $reste(A,B) = reste(A-B,B)$
$quotient\ (A,B) = 1+quotient(A-B,B)$

à partir de cette analyse on peut construire un algorithme simple que nous omettons ici, mais dont nous retenons seulement le coût : si Q est le quotient, le calcul nécessite Q soustractions.

Une deuxième analyse est fondée sur l'idée d'améliorer ce résultat. On cherche à réduire le coût de manière logarithmique.

Pour diminuer le coût du calcul, on cherche à le ramener au même calcul portant sur deux autres nombres C,D mais de coût moindre. On déduit de la première analyse que le coût diminue si le diviseur augmente. On cherche ainsi à établir une relation entre le reste R et le quotient Q de la division de A par B, et le reste R1 et le quotient Q1 de la division de A par 2B.

On peut écrire :

(1) $A = BQ+R$ (3) $A = (2B)Q1+R1$

(2) $O \leq R < B$ (4) $O \leq R1 < 2B$

(3) s'écrit

(3') $A = B(2Q1)+R1$

(4) se décompose en

(4') $O \leq R1 < B$

ou (4") $B \leq R1 < 2B$

. (3') et (4') définissent le reste et le quotient de la division de A par B.
Donc :

si $O \leq R1 < B$ on a $Q = 2Q1$
$R = R1$

. de (3') et (4") on tire :

$A = (2Q1+1)B+(R1-B)$
$O \leq R1-B < B$

et on peut encore identifier R et Q soit

si $B \leq R1 < 2B$ on a $Q = 2Q1+1$
$R = R1-B$

Ceci conduit alors à l'algorithme suivant :

b) Algorithme :

```
fonction RQ(entier A,B) → entier, entier :
  {calcule le reste et le quotient de la division entière de A par B.
   A et B positifs}
  choix
      A < B : → (A,O)
      A ≥ B : entier R,Q
              (R,Q) := RQ(A,2*B)
              choix
                  R < B : → (R,2*Q)
                  R ≥ B : → (R-B,1+2*Q)
              fchoix
  fchoix
ffonction RQ
```

c) Transformation

La transformation de la fonction *RQ* est une application de la version 1 donnée au § 4.3. En effet on écrit tout d'abord l'action *ARQ* :

```
action ARQ (entier A,B) :
  {calcule le reste et le quotient de la division de A par B et range
   ces valeurs dans deux variables ZR et ZQ. A,B > 0}
  choix
       A < B : ZR:=A ; ZQ:=0
       A ≥ B : ARQ(A,2*B)   {reste et quotient de A par 2B dans ZR et ZQ
               choix
                    ZR < B: ZQ:=2*ZQ
                    ZR ≥ B: ZQ:=1+2*ZQ ; ZR:=ZR-B
               fchoix
  fchoix
faction
```

on identifie alors les éléments du schéma de la manière suivante :

```
param       : entier, entier
A(X,Y)      : X < Y
ALPHA(X,Y)  : ZR:=X ; ZQ:=0
BETA(X,Y)   : vide
GAMMA(X,Y)  : si ZR < Y
                 alors ZQ:=2*ZQ
                 sinon ZQ:=1+2*ZQ ; ZR:=ZR-B
              fsi
T(X,Y)      : X,2*Y
T-1(X,Y)    : X,Y/2
```

on obtient alors l'algorithme itératif suivant en appliquant le schéma général du § 4.3.

Algorithme : reste et quotient de deux entiers A et B donnés

```
X:=A ; Y:=B
tantque X ≥ Y faire Y:=2*Y ftantque
ZR:=X ; ZQ:=0
tantque Y≠B faire
  B:=B/2
  Q:=2*Q ; si ZR ≥ Y alors ZQ:=ZQ+1 ; ZR:=ZR-Y fsi
ftantque
```

Remarques :

- Nous obtenons ici l'algorithme habituellement présenté, soit pour illustrer les techniques de manipulation d'assertions ([Dij 72] p. 13 , [Wir 73 a] p. 36) ou plus généralement la construction des programmes ([Ars 77] p. 37 et p. 73, [Bau 76] p. 3 et p. 7)
- Nous avons choisi cet exemple pour illustrer notre technique de transformation. Nous reviendrons au chapitre 5 (§ 5.3) à sur le problème de la réduction logarithmique du coût d'un algorithme et étudierons alors cet exemple à nouveau.

4.8. - CONCLUSION

Dans le cadre très général d'un processus de construction de programmes par transformations successives, nous avons évoqué le problème de la transformation "récursif-itératif". Notre approche est la suivante :

Considérant la forme récursive donnée comme une spécification du problème à programmer, nous avons appliqué le traitement séquentiel en :

. interprétant le problème comme le traitement d'une file abstraite définie formellement à partir de l'algorithme récursif donné
. appliquant les résultats du chapitre 2 pour produire un algorithme itératif.

Cette approche "globale" est adaptée à un souci de transformation manuelle. Nous l'opposons aux techniques fondées sur l'axiomatique des constructions de base et directement guidées par le processus dynamique engendré par l'algorithme récursif (cf. exemple [Vei 76] , [PaP 76], [ArK 79]) qui sont plutôt adaptées à un souci de transformation automatique.

CONCLUSION : ASPECTS PRATIQUES ET METHODOLOGIQUES DU TRAITEMENT SEQUENTIEL

Les travaux que nous avons présentés dans cette première partie doivent être examinés selon deux points de vue :

. un point de vue pratique : nous donnons une méthode de construction de programmes et des schémas généraux.
. un point de vue méthodologique : nous illustrons une démarche générale d'étude et de recherche de méthodes de construction.

Sur un plan pratique, nous avons décrit un modèle de processus de construction de programmes (chapitres 1 et 2) et nous avons illustré l'application de ce processus à divers niveaux :

. sur quelques exemples simples tout au long de la présentation de la méthode dans les chapitres 1 et 2
. sur un exemple concret (§ 3.3.) représentatif de nombreux problèmes réels (que nous avons étudiés par ailleurs).
. lors de l'élaboration de schémas généraux (§ 2.2. et 3.1.) ou particuliers à un type d'application informatique (§ 3.2.).

L'intérêt de la méthode proposée a ainsi pu être amplement démontré notamment par les aspects systématiques qu'elle comporte.

Finalement nous avons présenté un résultat significatif dans le domaine de la production de programmes par transformations successives : l'application du traitement séquentiel est à la base d'un procédé général de transformation "récursif-itératif" (chapitre 4) qui regroupe dans un cadre précis les considérations qui conduisent aux solutions connues de ce problème.

Un autre résultat concret de cette première partie est l'ensemble des schémas de traitement séquentiel que nous avons fournis : leur définition, leur classification et leur étude à divers niveaux d'abstraction en font des outils directement utilisables dans le processus de programation. Soulignons à nouveau l'intérêt général des évaluations qui accompagnent ces schémas (bien qu'elles aient été réalisées, pour le

besoin de l'exemple, dans un contexte particulier) : elles permettent à la fois de confirmer la cohérence des schémas, d'expliciter leur comportement et d'estimer leur coût.

Le dernier aspect pratique que nous voulons souligner ici tient à la présentation même de notre méthode : nous avons eu le souci constant de concilier un certain réalisme et la plus grande généralité. Ceci apparaît à deux niveaux :

. Nous avons choisi des notations (§ 1.1.3. et 2.1.2.) qui puissent être facilement adaptées aux formalismes les plus divers (comme elles le sont à notre formalisme de description d'algorithmes), qu'ils soient récents ou plus conventionnels.

. Nous avons étudié le traitement séquentiel à divers niveaux d'abstraction : le chapitre 1 est plutôt représentatif d'un niveau d'analyse (particulièrement adapté aux "formalismes applicatifs") et le chapitre 2 est plutôt représentatif d'un niveau de programmation plus conventionnel (mieux adapté aux "formalismes procéduraux").

Sur un plan méthodologique, le traitement séquentiel est la première illustration d'une démarche générale d'étude et de recherche de méthodes, dont on peut donner ici quelques lignes directrices :

1. discrétiser l'activité de programmation ; c'est l'idée même de "processus" de construction de programmes (qui sous-entend l'idée de production de programmes par transformations successives).

2. placer le processus de construction de programmes dans le cadre précis d'une "structure de donnée abstraite" définie à partir d'une structure de donnée concrète : les résultats connus sur cette dernière sont autant d'éléments qui permettent d'élaborer ce cadre directeur.

3. organiser le processus de construction de programmes en étapes très précises et fournir pour chacune d'elles des modèles pour leur réalisation et des notations pour leur description.

4. traiter des problèmes généraux en appliquant les principes de construction ainsi établis, et amorcer l'élaboration d'une base de schémas représentatifs de l'expérience collective ou d'une expérience particulière (que ce soit à des fins didactiques ou pour fournir des outils directement utilisables dans le processus de construction).

5. Intégrer la méthode étudiée au sein des méthodes et techniques générales existantes : analyse descendante, manipulation d'assertions, analyse récurrente, techniques de raffinement et d'optimisation...

Nous allons illustrer cette démarche à nouveau dans la deuxième partie en proposant une nouvelle méthode conçue d'après les mêmes idées directrices.

DEUXIEME PARTIE

LE TRAITEMENT ARBORESCENT

INTRODUCTION

Nous présentons maintenant une deuxième méthode de construction systématique de programmes, le *"traitement arborescent"*, dont les bases sont les mêmes que celles du traitement séquentiel :

1. recherche d'une organisation logique de l'information permettant de la structurer en une arborescence d'informations plus élémentaires.

2. recherche d'une formulation du problème posé en termes d'un traitement de cet arbre.

Comme dans le cas du traitement séquentiel, le processus de construction de programmes comporte ainsi une étape de raffinement et de structuration de l'information, et une étape d'analyse du traitement guidées par la connaissance d'algorithmes généraux sur les arbres.

Si l'on veut comparer les deux méthodes, le traitement arborescent est plus puissant, mais plus complexe que le traitement séquentiel :

- il est plus puissant car la structure d'arbre n'impose pas un ordre total sur les composants de l'information : il est moins contraignant de munir une information d'une structure d'arbre (ordre partiel) que d'une structure de file (ordre total).

- il est plus complexe car les algorithmes de traitement d'arbres sont plus nombreux et plus sophistiqués que les algorithmes de traitement de files. En effet, l'information n'étant pas totalement ordonnée, la structure d'arbre laisse une plus grande liberté dans la manière d'accéder aux informations que la structure de file.

Ces différences sont confirmées lorsque l'on examine les constructions linguistiques couramment associées au traitement des files et des arbres : l'itération pour les premières, la récursivité pour les seconds. De même que l'un des effets de l'application du traitement séquentiel est une plus grande sureté dans l'écriture des programmes itératifs, l'application du traitement arborescent contribue à une meilleure maîtrise de la récursivité.

Dans le chapitre 5, nous donnons les principes du traitement arborescent : nous proposons quelques définitions et notations, nous présentons un modèle de construction d'algorithmes et nous étudions les algorithmes fondamentaux qui sont à la base de l'application du traitement arborescent.
Le chapitre 6 est consacré à l'application du traitement séquentiel aux traitements d'arbres : écriture d'algorithmes itératifs de traitement d'arbres et définition d'un modèle d'analyse de problèmes basé sur la référence à ces algorithmes. Enfin nous montrons dans le chapitre 7 comment le traitement arborescent peut servir à l'étude d'algorithmes récursifs : que ce soit pour les évaluer, pour produire des algorithmes itératifs équivalents ou pour les optimiser à partir de propriétés des arbres qui interviennent.

CHAPITRE 5

PRINCIPES DU TRAITEMENT ARBORESCENT

Nous montrons ici comment structurer une information en un *"arbre"* d'informations plus élémentaires et comment construire un algorithme à partir d'une telle interprétation de l'information. Dans un deuxième temps, nous construisons les algorithmes qui permettent de résoudre nos trois problèmes de base : énumération, énumération partielle d'un arbre et recherche d'une information dans un arbre. Enfin, nous terminons le chapitre par une application du traitement arborescent à l'étude de l'accélération logarithmique d'une classe d'algorithmes.

Comme la notion de "file" dans le traitement séquentiel, la notion d'"arbre" est l'élément central dans la maîtrise du traitement arborescent. Elle s'inspire d'une structure de données classiques (cf. notamment le chapitre consacré aux arbres dans [Knu 69]), et nous l'utilisons pour guider le processus de construction de programmes (on trouvera une démarche voisine dans [Mah 73] et [MaB 78]) : le concept d'*"arbre"* sert de modèle pour raffiner l'information caractérisant le problème analysé. Si ce raffinement réussit, on peut interpréter ce problème comme un traitement de cet arbre, et construire un algorithme sur la base de cette interprétation.

Notre définition de la notion d'*"arbre"* répond ainsi aux deux objectifs suivants :

. permettre la description du processus de structuration en *"arbre"*

. permettre la description d'algorithmes de traitement d'arbres.

En aucun cas, nous n'avons à décrire une création ou une modificatio d'arbre. C'est à ce niveau que le concept d'arbre que nous présentons ici, diffère sensiblement de la structure de données dont il s'inspire. Nous

utilisons l'expression *"arbre abstrait"*(caractéristique du traitement arborescent) lorsqu'il peut y avoir ambiguité.

Remarque :
Soulignons l'analogie des démarches suivies pour définir et présenter le traitement arborescent et le traitement séquentiel.

5.1 - GENERALITES SUR LES ARBRES

5.1.1. - Définitions

Nous donnons la définition récurrente suivante :

Un arbre est un ensemble non vide, structuré comme suit :

- un des éléments est désigné particulièrement comme étant la *"racine"* de l'arbre.
- il existe une partition sur les éléments restants (s'il y en a) et chaque classe de cette partition est elle-même un arbre : on parle des *"sous-arbres"* de la racine.

Dans cette définition, l'ensemble est considéré comme la réunion de l'un de ses éléments (la racine) et de sous-ensembles qui lui sont directement associés (les sous-arbres). Outre les relations que ceci implique, on doit souligner le fait qu'on fixe ainsi un certain niveau d'abstraction qui met en évidence un élément et les sous-ensembles associés (on fait abstraction de la composition de ces sous-ensembles). Un niveau d'abstraction plus bas (un niveau de détails plus fins) permet de considérer à son tour chacun des sous-arbres : ils sont eux-mêmes composés d'une racine et de sous-arbres...

Ainsi tout élément de l'arbre est racine d'un sous-arbre : on a une relation biunivoque entre l'ensemble des éléments considéré et l'ensemble des sous-arbres que comporte l'arbre. On peut ainsi désigner un sous-arbre par le nom de sa racine, et établir des relations entre les éléments de l'arbre.

Nous recourons ici à la terminologie classique sur les arbres ("noeud", "fils", "frère", "père", "feuille", "profondeur", "degré d'un noeud",...). En général, nous utilisons les termes selon le sens donné dans [Knu 69] (pp. 305-313). Précisons trois définitions :

. un arbre est *"ordonné"*, si l'on établit une relation d'ordre entre les sous-arbres d'un noeud.

. une "forêt" est un ensemble d'arbres, non vide et ordonné. Une forêt est désignée par le nom de la racine de son premier arbre.
C'est ainsi que nous parlons, dans le cas d'un arbre ordonné, de la forêt des sous-arbres d'un noeud.

. un *"arbre binaire"* est un ensemble fini (éventuellement vide) qui est composé d'une racine et de deux sous-arbres (éventuellement vides). L'un des sous-arbres est appelé le *"sous-arbre gauche"* , l'autre le *"sous-arbre droit"*.

Remarques :

- cette terminologie (sous-arbre gauche, sous-arbre droit) définit une relation entre les fils d'un noeud. En ce sens, tout arbre binaire est ordonné.
- nous appelons *"arbre binaire homogène"* , un arbre binaire dans lequel tout noeud non terminal (qui n'est pas une feuille) a exactement deux fils. Nous étudions plus loin (§ 5.1.4.) le cas particulier des *"arbres binaires complets"*.
- les arbres binaires jouent un rôle important dans notre méthode dans la mesure où le traitement d'un arbre non binaire peut être ramené systématiquement au traitement d'un arbre binaire, par une application particulière du traitement arborescent (§ 5.1.5).

5.1.2. - Structuration d'une information en arbre

Le processus qui permet de structurer une information en arbre comporte deux étapes :

1°) raffinement de l'information : on décompose l'information en un ensemble d'informations plus élémentaires (les noeuds de l'arbre).

2°) structuration de l'information : l'ensemble ainsi obtenu est alors structuré en arbre (ou en arbre binaire).

a) Structuration en arbre - cas général

Pour structurer un ensemble non vide d'informations en arbre, nous appliquons le processus récursif suivant :

1. on désigne l'une des informations comme la racine de l'arbre que l'on construit.

2. on répartit les informations restantes en groupes disjoints

3. on structure en arbre chacun de ces groupes.

Remarque :
Le processus s'arrête à l'étape 1, dès que l'ensemble considéré n'a qu'un seul élément.

b) Structuration en arbre binaire, première forme

La structuration d'un ensemble non vide en arbre binaire est particulière :

1. on désigne l'une des informations comme la racine de l'arbre binaire que l'on construit

2. on répartit les informations restantes en un ou deux groupes disjoints.

3. on désigne l'un des groupes comme le "gauche", l'autre comme le "droit" (s'il n'y a qu'un groupe, il reçoit le qualificatif "gauche" ou "droit").

4. on structure en arbre binaire le ou les groupes restants.

Remarque :
Le processus s'arrête à l'étape 1 dès que l'ensemble considéré n'a qu'un seul élément.

c) Structuration en arbre binaire, deuxième forme

Une deuxième manière de décrire le processus de structuration en arbre binaire fait intervenir la notion d'*"arbre binaire vide"*. Pour structurer un ensemble éventuellement vide en arbre binaire :

1. on fait correspondre à l'ensemble vide, un arbre binaire vide

2. on désigne l'une des informations comme la racine de l'arbre

3. on répartit les informations restantes en deux groupes (éventuellement vides)

4. on désigne l'un des groupes comme le "gauche", l'autre comme le "droit"

5. on structure en arbre binaire, les deux groupes ainsi constitués.

Remarque :
Le processus s'arrête à l'étape 1 dès que l'ensemble considéré est vide.

5.1.3. - Description d'un arbre : un modèle de notations

Nous proposons des notations pour la description d'un arbre. On peut les considérer comme un résumé du résultat du processus de structuration en arbre. Elles permettent par ailleurs de décrire des algorithmes en faisant abstraction de la représentation exacte de l'arbre (notamment on ne suppose pas a priori que tous les noeuds de l'arbre existent en mémoire à un instant donné).

Comme pour les files (§ 1.1.3.) nous donnons un nombre de primitives qui n'est pas minimal : cette redondance laisse une plus grande liberté dans la construction des algorithmes.

Description d'un arbre

Les étapes de la description d'un arbre sont

1°) la description de l'information caractérisant chaque noeud de l'arbre (résultat du raffinement)
2°) la description des relations entre les noeuds de l'arbre (résultat de la structuration en arbre).

Notations :

Nous proposons deux ensembles de notations, selon que l'arbre est considéré comme un arbre binaire ou non.

a) Cas d'un arbre binaire

. tout noeud de l'arbre désigne le sous-arbre dont il est racine

. *RACINE* en abrégé *RAC*

désigne la racine de l'arbre considéré

. *GAUCHE(N)* en abrégé *G(N)*

désigne la racine du sous-arbre gauche du noeud *N* (sil existe)

. *DROIT(N)* en abrégé *D(N)*

désigne la racine du sous-arbre droit d'un noued *N* (s'il existe)

. *PERE(N)*

désigne le père du noeud *N*

EXISTE_GAUCHE(N) en abrégé *E_G(N)*

est un prédicat indiquant si *N* a un sous-arbre gauche non vide

(E_G(N)=vrai) ou non *(E_G(N)=faux)*.

. *EXISTE_DROIT(N)* en abrégé *E_D(N)*

est un prédicat indiquant si *N* a un sous-arbre droit non vide

(E_D(N)=vrai) ou non *(E_D(N)=faux)*.

. *FEUILLE(N)*

est un prédicat indiquant si *N* est une feuille

Remarque : *FEUILLE(N)=¬E_G(N)et¬E_D(N)*

. *ESTDROIT(N)*

est un prédicat indiquant si le noeud *N* est racine d'un sous-arbre droit.

Remarque : *ESTDROIT(N)=vrai* implique *N=DROIT(PERE(N))*

. *FRERE(N)*

désigne le frère du noeud *N*,c'est-à-dire

. *¬ESTDROIT(N)* : *FRERE(N)=DROIT(PERE(N))*
. *ESTDROIT(N)* : *FRERE(N)=GAUCHE(PERE(N))*

. *INFO(N)*

désigne l'information propre attachée au noeud *N*.

Remarque :
Nous faisons explicitement référence au sous-arbre vide en introduisant un noeud particulier, le noeud racine du sous-arbre vide Ce noeud est noté *NIL* . Dans ce cas, on a :

. *¬ E_G(N)* : *G(N) = NIL*
. *¬E_D(N)* : *D(N) = NIL*

b) Cas général (arbre non binaire)

Les notations que nous proposons pour le cas général sont basées sur l'interprétation suivante de l'arbre :
à tout noeud de l'arbre est associée la forêt de ses sous-arbres et inversement tout noeud de l'arbre est la racine d'un arbre faisant partie d'une forêt (celle des fils de son père).

Cette interprétation implique qu'on ordonne les fils d'un noeud (même si la logique du problème ne l'impose pas), mais elle nous permet de recourir facilement au traitement séquentiel pour traiter les arbres. La forêt des fils d'un noeud est en effet une file de noeuds (ou de sous-arbres suivant le niveau où on se place) définie comme suit :

. un élément est un noeud (un sous-arbre)

. la file des fils (sous-arbres) du noeud N est définie par :

$FILE\_VIDE_N$	: *FEUILLE(N)*	
PRE_N	: *FILS(N)*	
$SUC_N(X)$	: *FRERE(X)*	{X est un fils de N}
$E\_S_N(X)$	: *E_F(X)*	{X est un fils de N}

avec les conventions suivantes :

. *FILS_AINE(N)* en abrégé *FILS(N)*

désigne le fils ainé (le premier fils) de N , si N n'est pas une feuille

. *FRERE_CADET(N)* en abrégé *FRERE(N)*

désigne le frère cadet (le prochain frère) de N , si N a un frère

. *EXISTE_FRERE_CADET(N)* en abrégé *E_F(N)*

est un prédicat indiquant si N a un frère cadet *(E_F(N)=vrai)* ou si N est le dernier frère dans la file de frères *(E_F(N)=faux)*

. *FEUILLE(N)*

est un prédicat indiquant si N a des fils *(FEUILLE(N)=faux)* ou non *(FEUILLE(N)=vrai)*

. *FRERE_PREC(N)*

désigne le frère ainé de N :*FRERE(FRERE_PREC(N))=N*

Remarques :

. les notations *RACINE, PERE, INFO* et *NIL* gardent le sens donné pour les arbres binaires.

. soulignons à nouveau que le fait de se référer à une information par une des notations précédentes n'implique rien sur la manière dont l'arbre est représenté. Par exemple, à l'information *PERE* peut correspondre le sommet d'une pile dans une réalisation, et un pointeur dans une autre réalisation.

5.1.4. - Exemples de description d'arbres

Nous illustrons ici l'utilisation des notations précédentes par l'intermédiaire de deux exemples :

. le premier présente les arbres binaires complets qui jouent un rôle assez important qui apparaîtra dans la suite de cette présentation (§ 5.3.).

. le second est extrait d'une application du traitement arborescent à la résolution d'un problème (énumération des parties d'un ensemble cf.[LuS 79]).

a) Exemple 1 : arbres binaires *"complets"*

a.1. Définition :

Pour définir les arbres binaires complets, nous considérons les deux numérotations suivantes des noeuds d'un arbre binaire :

. la numérotation par *"niveau"* : *NUM_NIV(N)* désigne le numéro associé à un noeud dans cette numérotation (la racine a pour numéro 1, les noeuds sont numérotés par niveau croissant, de gauche à droite dans chaque niveau).

. la numérotation définie par les relations de récurrence suivantes : *(NUM(N)* désigne le numéro associé à *N* dans cette numérotation).

. *NUM(RAC)=1*
. *E_G(N)=vrai* : *NUM(G(N))=2*NUM(N)*
. *E_D(N)=vrai* : *NUM(D(N))=2*NUM(N)+1*

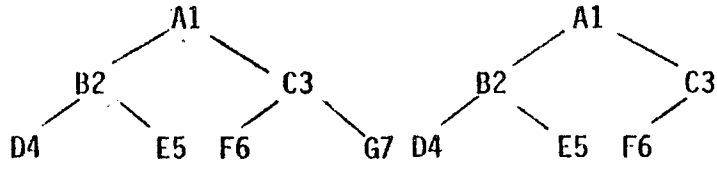
Dans ces conditions, un arbre binaire est complet, si et seulement si les deux numérotations ci-dessus associent les mêmes numéros aux noeuds de l'arbre, c'est-à-dire :

NUM(N) = NUM_NIV(N) pour tout noeud *N* de l'arbre

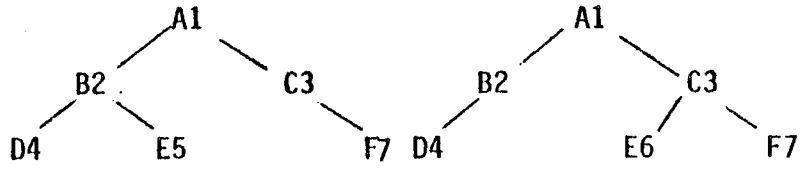
Exemples :

(dans les dessins ci-dessous, nous figurons la numérotation par niveau à l'aide de lettres de l'alphabet, et la deuxième numérotation à l'aide d'entiers. L'arbre binaire est complet si l'entier correspond au rang de la lettre dans l'alphabet).

Les deux arbres binaires suivants sont complets :



mais les suivants ne le sont pas :



Remarque :
Un arbre binaire complet peut avoir au plus un noeud non terminal ayant un seul fils. Dans un tel cas, ce fils unique est un fils gauche et le noeud est le dernier (dans l'ordre par niveau) non terminal. C'est pourquoi, on peut en général appliquer aux arbres binaires complets les résultats obtenus pour les arbres binaires homogènes.

a.2. Utilisation de la numérotation *NUM* pour décrire l'arbre

La numérotation *NUM* ci-dessus permet d'associer à chaque noeud d'un arbre binaire complet un numéro unique. On peut alors décrire un arbre complet de *N* noeuds de la manière suivante :

1. description de l'information
   un noeud de l'arbre est caractérisé par son numéro *I* dans la numérotation
2. description des relations
   les primitives définissant l'arbre binaire sont alors :

```
RAC          : 1
G(i)         : 2*i
D(i)         : 2*i+1
E_G(i)       : 2*i≤N
E_D(i)       : 2*i+1≤N
PERE(i)      : i div 2          {div est la division entière }
ESTDROIT(i)  : impair i         {impair i=vrai si i est impair}
FRERE(i)     : choix
                 impair i : i-1
                 pair i   : i+1
               fchoix
```

a. 3. Représentation d'un arbre binaire complet à l'aide d'un vecteur

La numérotation *NUM* permet de représenter un arbre binaire complet à l'aide d'un vecteur de taille *N* . Si *V* est ce vecteur, il suffit de mettre en position *I* l'information associée au noeud de numéro *I* :

INFO(I) : V(I)

Remarque :

Cette représentation reste possible dans le cas des arbres binaires non complets. Toutefois un certain espace sera perdu : toutes les positions du vecteur correspondant à des numéros associés à des sous-arbres vides ou inexistants dans l'arbre.

b) Exemple 2 : ensemble des combinaisons de N objets

On donne N objets numérotés de 1 à N (et désignés par ces entiers, et on cherche à organiser l'ensemble des combinaisons de K objets pris parmi les N objets donnés, pour K allant de 1 à N (en d'autres termes, l'ensemble des parties de l'ensemble donné).

Remarque :

Une première solution est une organisation en file, qui permet l'application du traitement séquentiel : l'ensemble des combinaisons comporte 2^N-1 combinaisons. A chaque combinaison, on associe l'entier compris entre 1 et 2^N-1 vérifiant la propriété suivante : on considère la forme binaire de cet entier

$$\sum_{i=0}^{N-1} b_i$$

la combinaison associée est alors définie simplement par :

l'objet i appartient à la combinaison si et seulement si : $b_{i-1}=1$

Nous cherchons ici à structurer l'ensemble des combinaisons en arbre :

. raffinement : à chaque combinaison, de K objets nous associons une forme canonique consistant du K-uplet des numéros de ces objets en ordre croissant.

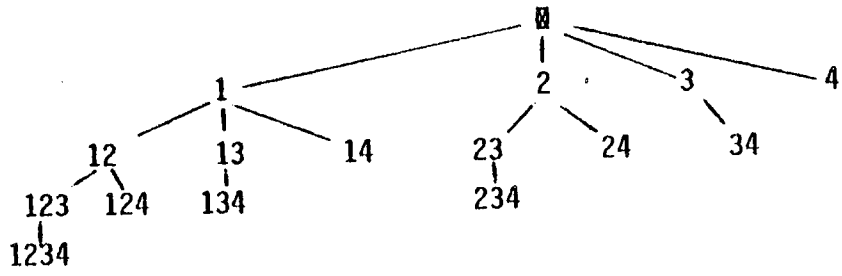
. l'ensemble est alors structuré en arbre :

- la racine de l'arbre est un noeud correspondant à la combinaison de 0 objet pris parmi les N donnés (c'est-à-dire à la partie vide).
- on répartit les combinaisons selon la valeur du premier objet (dans la forme canonique) : on obtient ainsi N groupes disjoints

de combinaisons, qui forment N sous-arbres.

- chacun de ces sous-arbres est à son tour partitionné en fonction de la valeur du second objet (dans la forme canonique). On obtient ainsi C_N^2 groupes.
- ...

par exemple, pour N=4, on obtient l'arbre suivant :



Pour définir cet arbre, nous cherchons à définir les relations entre les noeuds en termes des notations précédentes :

1. description de l'information

Un noeud est caractérisé par un k-uplet c et sa longueur κ
Nous utilisons les notations suivantes pour manipuler les K-uplets :
C étant un k-uplet, x étant un objet et i une position du k-uplet,

$c \dagger x$	désigne le k-uplet obtenu à partir de c en rajoutant x à sa fin
$\dagger c$	désigne le k-uplet obtenu à partir de c en enlevant son dernier élément
$c \oplus x$	désigne le k-uplet obtenu à partir de c en remplaçant son dernier élément par x
c_i	désigne le ième élément du K-uplet

2. Description des relations

Les primitives sont :

$RACINE : 1 , 1$
$FILS(C,K) : C \uparrow C_k+1, K+1$
$FRERE(C,K) : C \oplus C_k+1, K$
$FEUILLE(C,K) : C_k = N$
$E\_F(C,K) : C_k \neq N$
$PERE(C,K) : \downarrow C, K-1$
$INFO(C,K) : C$

5.1.5. - Structuration d'un arbre non binaire en un arbre binaire

Etant donné un arbre non binaire A, on peut toujours le structurer en arbre binaire de la manière suivante :

1. Description de l'information

à un noeud de l'arbre donné A est associé un noeud de l'arbre que l'on décrit (et réciproquement)

2. Description des relations

les primitives qui définissent l'arbre binaire sont les suivantes (nous indexons par la lettre A les noms des primitives correspondant à l'arbre donné pour éviter toute ambiguité) :

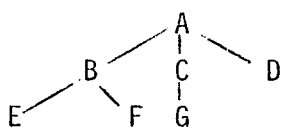
- $RAC : RAC_A$
- $G(N) : FILS_A(N_A)$ {N_A est le noeud auquel correspond N}
- $D(N) : FRERE_A(N_A)$
- $E\_G(N) : \neg FEUILLE_A(N_A)$
- $E\_D(N) : E\_F_A(N_A)$
- $PERE(N)$: choix
  - $ESTDROIT(N) : FRERE\_PRED_A(N_A)$
  - $\neg ESTDROIT(N) : PERE_A(N_A)$

  fchoix
- $ESTDROIT(N) : N_A \neq FILS_A(PERE_A(N))$
- $INFO(N) : INFO_A(N_A)$

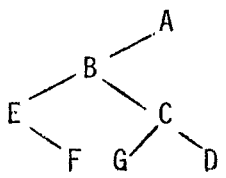
Exemple :

Dans le dessin ci-dessous, l'arbre binaire de droite est défini à partir de l'arbre non binaire de gauche, en appliquant le traitement arborescent selon ce qui précède.

Arbre non binaire



arbre binaire associé



Nous verrons plus loin (§ 5.2.1.) comment interpréter un traitement de l'arbre non binaire de départ, en termes d'un traitement de l'arbre binaire issu de cette application du traitement arborescent.

Remarque :

Réciproquement, on peut appliquer le traitement arborescent à tout arbre binaire, de manière à définir un arbre non binaire (nous en verrons une application au chapitre 7).

L'application particulière du traitement arborescent que nous venons de décrire est importante car elle nous permet de ramener le traitement des arbres non binaires à un traitement d'arbre binaire (et réciproquement). Ainsi nous pouvons limiter la présentation du traitement arborescent au cas d'une structuration en arbre binaire (nous choisissons les arbres binaires car leur structure est plus simple : on connaît pour chaque noeud le nombre maximum de fils).

Remarque :

Signalons toutefois que l'on peut toujours effectuer l'opération inverse, c'est-à-dire : ou bien structurer directement l'information en arbre non binaire, ou bien se ramener à ce cas en structurant un arbre binaire donné, en arbre non binaire (cf. par exemple [Vei 74]).

5.1.6. - Analyse récurrente et traitement d'arbres

De par leur définition récurrente, les arbres sont particulièrement adaptés à l'analyse récurrente : étant donné un problème portant sur un arbre, on cherche à le définir en termes du même problème portant sur les sous-arbres.

a) Première forme d'analyse récurrente

a.1. Dans le cas d'un arbre binaire :

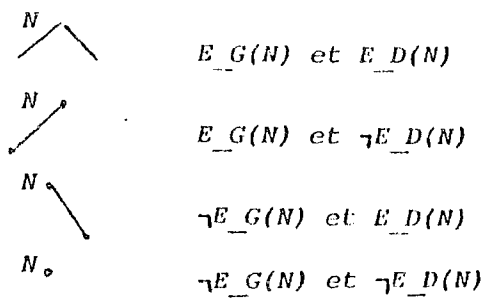
On examine les cas suivants qui s'excluent mutuellement et recouvrent l'ensemble des cas possibles :

cas n°1. noeuds ayant deux fils

cas n°2. noeud ayant un fils, le gauche

cas n°3. noeud ayant un fils, le droit

cas n°4. noeud terminal (feuille)



a.2 Dans les cas des arbres non-binaires

On examine les deux cas suivants :

Cas n°1 : le noeud est une feuille

Cas n°2 : le noeud n'est pas une feuille : on traite la forêt des fils

Remarque :

On peut se ramener à la technique précédente, en appliquant la structuration en arbre bianire décrite au § 5.1.5., ce qui revient à envisager pour un noeud onné l'existence ou la non existence d'une forêt de fils et d'une forêt de frères (*E_G* correspond à *FEUILLE* et *E_D* correspond à *E_F*).

b) Exemples

1. Exemple 1 : soit à calculer la profondeur d'un arbre binaire

La profondeur d'un arbre binaire est définie comme le nombre de noeuds du plus long chemin menant de la racine à une feuille. La définition récurrente de la profondeur est la suivante :

profondeur (X) :

: *1+max(profondeur(G(X),profondeur(D(X))*

: *1+profondeur(G(X))*

: *1+profondeur(D(X))*

: *1*

la profondeur de l'arbre donné est alors défini comme étant *profondeur(RAC)*.

b. 2. Exemple 2 : soit à calculer le degré maximum d'un arbre (non binaire).

Le degré maximum d'un arbre est défini comme le maximum des degrés (nombre de fils) des noeuds de l'arbre.

Nous proposons l'analyse suivante qui illustre le traitement arborescent. On considère l'arbre défini à partir de l'arbre donné en associant à chaque noeud son rang dans la liste de fils à laquelle il appartient. Cet arbre est simplement défini comme suit (les noms des primitives du nouvel arbre sont suivis du caractère ').

. Description de l'information

Un noeud est constitué d'un noeud de l'arbre de départ et d'un entier, qui correspond à son rang dans la liste de fils à laquelle il appartient.

. Description des relations

. *RAC'* : *RAC,1*
. *FILS'(N,F)* : *FILS(N),1*
. *FRERE'(N,F)* : *FRERE(N),F+1*
. *FEUILLE'(N,F)* : *FEUILLE(N)*
. *E_F'(N,F)* : *E_F(N)*

. Analyse récurrente

Le degré maximum de l'arbre de départ est alors le même que celui de ce nouvel arbre, que l'on peut simplement définir de manière récurrente :

degmax(N,F) :

: *max(degmax(FILS'(N,F)),degmax(FRERE'(N,F))*
soit :
max(degmax(FILS(N),1),degmax(FRERE(N),F+1)

: *max(F,degmax(FILS'(N,F))* soit
max(F,degmax(FILS(N),1))

: *degmax(FRERE'(N,F))* soit *degmax(FRERE(N),F+1)*

: *0*

Le résultat cherché est défini comme étant :

degmax(RACINE') soit *degmax(RACINE,1)*

c) Deuxième forme d'analyse récurrente (arbres binaires)

Un autre type d'analyse par cas consiste à considérer l'alternative suivante :

. l'arbre considéré est vide
. l'arbre considéré n'est pas vide.

Une telle analyse, bien que plus concise peut être plus délicate que la précédente, car elle nécessite de choisir une convention pour le traitement de l'arbre vide, qui soit compatible avec le cas général. Dans le cas général, on considère toujours que le noeud a deux fils, ces fils pouvant

être vides. La technique consiste alors à définir le résultat pour le cas général, puis à définir le cas de l'arbre vide en résolvant un système d'équations définies à partir du cas général.

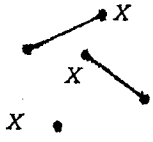
Exemple 1 : Reprenons le calcul de la profondeur d'un arbre binaire

profondeur (X)

X=NIL : Z {Z sera calculé ultérieurement}

X≠NIL : *1+max(profondeur(G(X),profondeur(D(X))*

et l'analyse précédente nous indique que :



: *1+max(profondeur(G(X)),Z)=1+profondeur(G(X))* (1)

1+max(Z,profondeur(D(X))=1+profondeur(D(X)) (2)

1+max(Z,Z) = 1 (3)

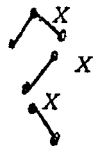
de (3) on tire $Z=0$, ce qui est compatible avec (1) et (2).

Ce type d'analyse peut ne pas aboutir dans la mesure où l'on est conduit à un système d'équations incohérentes. Nous donnons un exemple de ce phénomène.

Exemple 2 : plus petit chemin de la racine à une feuille

Etant donné un arbre binaire, on désire calculer la longueur du plus petit chemin menant de la racine à une feuille.
Appliquant la première forme d'analyse récurrente, on obtient :

long-min(X)



: *1+min(long_min(G(X)),long_min(D(X)))*

: *1+long_min(G(X))*

: *1+long_min(D(X))*

: *0*

et le résultat cherché est défini comme étant :

long_min(RAC)

Si nous appliquons la deuxième forme d'analyse récurrente, on écrit :

long-min(X)

arbre vide : Z

arbre_non_vide : *1+min(long_min(G(X)),long_min(D(X)))*

et pour calculer Z , nous cherchons à appliquer les résultats de la première analyse (considérés comme une définition du problème). On obtient alors :

(1)  $1+min(long\_min(G(X)), Z) = 1+long\_min(G(X))$

(2)  $1+min(Z, long\_min(D(X)))=1+long\_min(D(X))$

(3)  $1+min(Z,Z)=0$

de (3) on tire : $Z=-1$

ce qui est incompatible avec (1) et (2).

Ainsi on a bien un exemple pour lequel seul la première forme d'analyse est applicable.

Remarque :

Dans ce qui suit, nous utiliserons systématiquement la première forme d'analyse récurrente (quitte à éventuellement raffiner les algorithmes obtenus dans une phase de simplification).

5.1.7. - Conclusion

Pour présenter les principes du traitement arborescent, nous avons suivi une démarche analogue à celle qui nous a guidés lors de la présentation du traitement séquentiel (§ 1.1.). Le processus de résolution d'un problème par le traitement arborescent comporte ainsi :

. le raffinement de l'information traitée en un ensemble, sa structuration en arbre et la description de cet arbre en termes de primitives qui constituent autant d'objectifs vers lesquels tend le processus de structuration.

. la construction d'un algorithme à partir d'une analyse récurrente guidée par la structure d'arbre donnée à l'information.

Nous avons de plus montré comment interpréter tout traitement d'arbre non binaire en termes d'arbre binaire. Ceci nous permet de simplifier la présentation du traitement arborescent en envisageant essentiellement le cas des arbres binaires dans ce qui suit.

5.2 - ALGORITHMES DE TRAITEMENT D'ARBRES

Nous appliquons maintenant les résultats précédents à la construction d'algorithmes de traitement d'arbres. Nous voulons ainsi :

. illustrer la méthode du traitement arborescent
. élaborer un ensemble de schémas que l'on puisse utiliser directement, dès que les conditions de leur application sont reconnues.

Comme pour le traitement des files, nous considérons ici deux groupes d'algorithmes sur les arbres.

Les algorithmes de parcours qui ramènent le traitement d'un arbre à un traitement élémentaire de chacun de ses noeuds (y compris l'énumération partielle d'un arbre).

Les algorithmes de recherche qui ramènent le traitement de l'arbre au traitement élémentaire d'un seul de ses noeuds.

5.2.1. - Parcours d'arbres

Le parcours d'un arbre est l'application d'une même action élémentaire à chacun des noeuds de l'arbre.

Nous utilisons les mêmes notations que pour le parcours d'une file (§ 1.2.1) : chaque noeud *x* de l'arbre est visité une fois et une seule par l'action *VISITER(X)* ; le parcours est initialisé par l'action *INIT_P* ; il est suivi de l'action de terminaison *TERM_P*.

Remarque :
Nous étudions ici les parcours d'arbres que l'on peut construire en appliquant les principes présentés au § 5.1.6., c'est-à-dire les parcours qui découlent de la structure récurrente des arbres. Il existe évidemment d'autres parcours que nous ne traitons pas ici (cf. [Knu 69]) et notamment le parcours par "niveau" (qui correspond à la numérotation par niveau évoquée au § 5.1.4.). et les parcours "dynamiques" (technique de "branch and bound" [SaK 79]

a) Parcours d'arbres binaires

a.1 - Analyse

(nous utilisons le premier type d'analyse par cas).

Le parcours d'un arbre de racine *x* est défini par :

parcours (X) :

{cas n°1}  "visiter X et parcourir son sous-arbre gauche et son sous-arbre droit"

{cas n°2}  "visiter X et parcourir son sous-arbre gauche"

{cas n°3}  "visiter X et parcourir son sous-arbre droit"

{cas n°4}  "visiter X "

Le parcours d'un arbre donné s'exprime alors par :

INIT_P ; parcours (RAC) ; TERM_P

Remarque :
On prouve simplement par récurrence sur la taille des arbres, que la définition ci-dessus est correcte, c'est-à-dire que chaque noeud est visité une fois et une seule.

a. 2 - Différents ordres de parcours :

Pour poursuivre la construction d'un algorithme, on peut chercher à préciser l'ordre dans lequel sont effectuées les actions de visite et de parcours dans chacun des cas ci-dessus.
Il existe évidemment de nombreuses solutions.
Nous retenons ici les 6 parcours qui correspondent aux diverses manières d'ordonner les trois actions :

VISITER(X)
parcours(G(X))
parcours(D(X))

Nous obtenons ainsi les parcours d'arbres binaires classiques :

* *"préordre"* : tout noeud est visitë avant le parcours de ses sous-arbres.

VISITER(X) ; parcours(G(X)) ; parcours(D(X))

ou bien

VISITER(X) ; parcours(D(X)) ; parcours(G(X))

* *"ordre symétrique"*: un noeud est visité entre le parcours du sous-arbre gauche et celui du sous-arbre droit.

parcours(G(X)) ; VISITER(X) ; parcours(D(X))

ou bien

parcours(D(X)) ; VISITER(X) ; parcours(G(X))

*"*ordre terminal*" : un noeud est visité après le parcours de ses sous-arbres.

parcours(G(X)) ; parcours(D(X)) ; VISITER(X)

ou bien

parcours(D(X)) ; parcours(G(X)) ; VISITER(X)

Remarques :

- la terminologie est choisie en fonction du moment de la visite de la racine par rapport au parcours des sous-arbres.
- nous avons souligné par la présentation même, le rôle symétrique joué par les sous-arbres gauches et les sous-arbres droits. Pour alléger la présentation, nous réduisons dans ce qui suit, ces 6 cas à 3 cas, en privilégiant toujours le sous-arbre gauche : dès qu'il y a deux sous arbres, quel que soit le moment de la visite de la racine, on commence toujours par le sous-arbre gauche. Il est facile de restituer les trois autres cas en inversan les rôles de "gauche" et de "droit" dans tous les algorithmes.

a3. algorithmes

Nous déduisons de ce qui précède les trois algorithmes de base suivants :

algorithmes de parcours d'arbres binaires

PREORDRE

```
action préordre (arbre A) :
  VISITER(A)
  si E_G(A) alors préordre(G(A)) fsi
  si E_D(A) alors préordre(D(A)) fsi
faction préordre

{parcours en préordre}
INIT_P ; préordre(RAC) ; TERM_P
```

ORDRE TERMINAL

```
action terminal (arbre A) :
  si E_G(A) alors terminal(G(A)) fsi
  si E_D(A) alors terminal(D(A)) fsi
  VISITER(A)
faction terminal

{parcours en ordre terminal}
INIT_P ; terminal(RAC) ; TERM_P
```

ORDRE SYMETRIQUE

```
action symétrique (arbre A)
  si E_G(A) alors symétrique(G(A)) fsi
  VISITER(A)
  si E_D(A) alors symétrique(D(A)) fsi
faction symétrique

{parcours en ordre symétrique}
INIT_P ; symétrique(RAC) ; TERM_P
```

b) Parcours d'arbre, cas général

b.1. Analyse

Le parcours d'un arbre de racine *X* est défini par :

parcours(X) :

{cas n°1} *X* est une feuille
"visiter *X* "

{cas n°2} *X* n'est pas une feuille
"visiter *X* et parcourir les sous-arbres de *X* "

le parcours d'un arbre s'exprime alors par :

INIT_P ; parcours(RAC) ; TERM_P

Remarque :
On prouve la correction de cette analyse par récurrence sur la taille des arbres.

b.2. Différents ordres de parcours

Comme précédemment, on peut poursuivre la construction d'un algorithme de parcours, en précisant l'ordre dans lequel sont effectuées les actions de visite et de parcours des sous-arbres d'un noeud donné. Il existe de nombreuses solutions, mais nous ne présentons ici que celles que l'on peut déduire d'une application du traitement séquentiel à l'ensemble des fils d'un noeud.

Nous considérons les deux parcours remarquables qui découlent de la décomposition du parcours d'un arbre en deux actions :

. visite de la racine
. parcours de tous les sous-arbres

On a alors :

* le *"préfixé"* : la visite d'un noeud précède le parcours de ses sous-arbres

* le *"postfixé"* :la visite d'un noeud suit le parcours de ses sous-arbres.

Remarque :
Nous utilisons une terminologie différente de celle proposée pour les arbres binaires afin de bien distinguer les deux types d'arbres considérés.

b. 3. Parcours des sous-arbres d'un noeud

Nous construisons maintenant deux algorithmes, en supposant que les fils d'un noeud sont ordonnés (de manière arbitraire ou non).

Le parcours des sous-arbres d'un noeud est réalisé en appliquant le traitement séquentiel :
On considère la file des sous-arbres (décrite comme indiquée au § 5.1.3.), et on lui applique un schéma de parcours de file. On peut exprimer ce parcours de file itérativement (§ 2.2.2.) ou récursivement (§ 1.2.1).
On obtient ainsi deux versions pour chaque ordre de parcours. Nous détaillons ici la construction pour l'ordre préfixé (la démarche est la même pour l'ordre postfixé).

. version 1

Le parcours de la forêt des fils est exprimé itérativement : on applique directement le schéma de parcours de file P2 (§ 2.2.2.) et on obtient les versions 1 pour les deux ordres de parcours.

. version 2

Un premier algorithme est le suivant (cas du préfixé) :
Algorithme :

```
action préfixé (arbre A) :
       VISITER(A)
       si ¬FEUILLE(A) alors
          préfixé-forêt (FILS(A))
       fsi
faction
```

{où *préfixé_forêt(X)* désigne le parcours des sous-arbres de la forêt dont le premier arbre est *X*}

```
action préfixé-forêt(arbre X)
       {cf. § 1.2.1.}
       préfixé(X)
       si E_F(X) alors
          préfixé_forêt(FRERE(X))
       fsi
faction-préfixé_forêt
```

{Le parcours de l'arbre donné est alors exprimé par : }

INIT_P ; préfixé(RAC) ; TERM_P

A partir de cette première solution, nous obtenons un algorithme plus concis, par un mécanisme de substitution :

On remplace tout appel de *préfixé* par sa définition. On obtient ainsi :

Algorithme :

```
action préfixé_forêt (arbre X) :
        {préfixé(X)}
        VISITER(X)
        si ¬FEUILLE(X) alors préfixé_forêt(FILS(X)) fsi
        si E_F(X) alors préfixé_forêt(FRERE(X)) fsi
faction préfixé_forêt
```

{le parcours de l'arbre donné est alors exprimé par :}

```
INIT_P
{préfixé(RAC)}
VISITER(RAC)
si ¬FEUILLE(RAC) alors
   préfixé_forêt (FILS(RAC))
fsi
TERM_P
```

Finalement, la version définitive est obtenue en écrivant l'appel initial sous une forme plus concise.

b.4. Algorithmes

parcours d'un arbre - cas général

PREFIXE version 1

```
action préfixé (arbre A) :
 VISITER(A)
 si ¬FEUILLE(A) alors
    x:=FILS(A)
    itérer préfixé(x)
    arrêt : ¬E_F(x)
          x:=FRERE(x)
    fitérer
 fsi
faction préfixé

{parcours en préfixé}
INIT_P; préfixé(RAC); TERM_P
```

POSTFIXE version 1

```
action postfixé (arbre A) :
 si FEUILLE(A) alors
  x:=FILS(A)
  itérer postfixé(x)
  arrêt : ¬E_F(x)
        x:=FRERE(x)
  fitérer
 fsi
 VISITER(A)
faction postfixé

{parcours en postfixé}
INIT_P; postfixé(RAC); TERM_P
```

PREFIXE version 2

```
action préfixé(arbre A) :
 VISITER(A)
 si ¬FEUILLE(A) alors
    préfixé(FILS(A))
 fsi
 si E_F(A) alors
    préfixé(FRERE(A))
 fsi
faction préfixé

{parcours en préfixé :}
INIT_P; préfixé(RAC); TERM_P
```

POSTFIXE version 2

```
action postfixé(arbre A) :
 si ¬FEUILLE(A) alors
    postfixé(FILS(A))
 fsi
 VISITER(A)
 si E_F(A) alors
    postfixé(FRERE(A))
 fsi
faction postfixé

{parcours en postfixé :}
INIT_P; postfixé(RAC) ; TERM_P
```

Remarque :
Si on considère les versions 2, on peut identifier le schéma de préordre (préfixé version 2) et le schéma d'ordre symétrique (postfixé version 2) des arbres binaires. On découvre ainsi le résultat suivant : si l'on structure un arbre non binaire en arbre binaire comme indiqué au § 5.1.5., le parcours préfixé de l'arbre de départ s'exprime comme un parcours en préordre de l'arbre binaire et le parcours postfixé de l'arbre de départ s'exprime comme un parcours en ordre symétrique de l'arbre binaire.
Cette remarque complète le § 5.1.5. et nous permet de ne considérer que le cas des arbres binaires dans ce qui suit.

c) Deux exemples particuliers de parcours d'arbres

c.1. Enumération des feuilles d'un arbre

Nous considérons le parcours d'arbre particulier où seules les feuilles sont effectivement visitées. Pour construire l'algorithme, nous appliquons ce qui précède en substituant à l'action *VISITER* la séquence

si FEUILLE(X) alors VISITER(X) fsi

Quel que soit l'ordre de parcours choisi, on obtient alors :

Algorithme : énumération des feuilles d'un arbre binaire

```
action en_feuilles (arbre A) :
  choix
     FEUILLE(A) : VISITER(A)
    ¬FEUILLE(A) : si E_G(A) alors en_feuilles(G(A)) fsi
                  si E_D(A) alors en_feuilles(D(A)) fsi
  fchoix
faction en_feuilles
```

et l'énumération des feuilles de l'arbre donné est alors réalisée par :

INIT_P ; en_feuilles(RAC) ; TERM_P

c.2. "Evaluation postfixée" d'un arbre

Nous examinons maintenant un type particulier de traitement arborescent dont la solution s'exprime sous forme d'un parcours postfixé (parcours terminal si l'arbre est binaire).

. énoncé du problème :

Il s'agit d'un problème général fréquemment rencontré (cf. notamment l'évaluation des expressions arithmétiques).

On définit la "valeur d'un arbre" de manière récurrente :

La valeur d'un arbre A , soit $V(A)$ est une fonction de l'information que porte sa racine et de la valeur de ses sous-arbres, soit :

$$V(A) = f_A(INFO(A), V(X_1), V(X_2), \ldots, V(X_k))$$

où K est le degré de la racine de l'arbre A et les X_i sont les fils du noeud A racine de l'arbre A .

Dans cette définition, on associe à chaque sous-arbre A un mode de calcul f_A de sa valeur. Très souvent, on se trouve dans le cas particulier où le mode de calcul de la valeur d'un arbre ne dépend que du nombre de ses sous-arbres.

$$V(A) = f_k(info(A), V(X_1), \ldots, V(X_k))$$

. Mise sous forme binaire :

A partir de cette définition qui concerne un arbre non binaire, on peut se ramener facilement au cas d'un arbre binaire, s'il est possible de définir la valeur d'une forêt comme une fonction des valeurs de ses arbres :

Soit $U(A)$ la valeur de la forêt dont le premier arbre est A

alors on a : $V(A) = g(INFO(A),\ V(FILS(A)),\ U(FRERE(A))$

et $U(A) = h_i(V(A),\ V(Y_1), V(Y_2), \ldots, V(Y_i))$

où les Y_j sont les frères cadets de A .

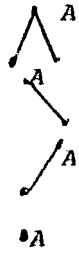
. Algorithme

On construit facilement un algorithme à partir des définitions récurrentes précédentes. Dans le cas d'un arbre binaire on obtient :

Algorithme : évaluation postfixée d'un arbre binaire

fonction ev_post (arbre A) → result :

choix



$: \rightarrow f_2\ (A,\ ev\_post(G(A)),\ ev\_post(D(A)))$

$: \rightarrow f_1\ (A,\ ev\_post(D(A)))$

$: \rightarrow f_1\ (A,\ ev\_post(G(A))$

$: \rightarrow f_0\ (A)$

fchoix

ffonction ev-post

Le calcul de la valeur de l'arbre donné est alors effectué par l'appel :

R := ev_post (RAC)

Cet algorithme engendre l'évaluation de tous les sous-arbres de l'arbre donné dans un ordre précis : pour évaluer un sous-arbre, on doit connaître les valeurs de ses sous-arbres. L'évaluation des sous-arbres se fait donc dans l'ordre terminal (postfixé) dans le cas d'un arbre non binaire).

Remarque :
Le calcul de la profondeur de l'arbre que nous avons développé au § 5.1.6. est un exemple d'évaluation postfixée d'un arbre binaire.

5.2.2. - *"Enumération partielle"* d'un arbre

L' *"énumération partielle"* d'un arbre est un cas particulier de parcours d'arbre, dans lequel le parcours d'un sous-arbre n'est effectué que si sa racine vérifie une propriété donnée, caractérisée par un prédicat P .

Remarque :
On peut rapprocher ce problème, du problème de parcours d'une sous-file (§ 1.2.3.) et de l'énumération partielle d'une file (§ 3.1.).

Nous étudions la construction d'un algorithme répondant à ce problème. L'une des solutions est particulièrement retenue car elle fournit un schéma général pour les problèmes de "retour-arrière" ("backtracking"). Nous illustrons ceci sur un exemple classique.

a) Construction d'un algorithme

En appliquant le modèle d'analyse proposé au § 5.1.6., nous obtenons l'algorithme général suivant :

Algorithme :

```
action enum-p (arbre A) :
   si P(A) alors
        "visiter la racine et appliquer l'énumération partielle aux
         sous-arbres gauche et droit"
   fsi
faction enum-p
```

L'énumération partielle de l'arbre donné est réalisée par :

INIT_P ; enum_p (RAC) ; TERM_P

Cet algorithme peut être raffiné comme nous l'avons fait pour les parcours d'arbres en fixant un ordre dans l'application des trois actions

VISITER(A)
enum_p(G(A))
enum_p(D(A))

et nous trouvons alors trois formes de base pour l'énumération partielle, analogues aux trois parcours d'arbres binaires.

Remarque :

Afin d'illustrer le traitement arborescent, nous proposons ici une autre manière de construire cet algorithme.
Nous considérons l'ensemble des noeuds de l'arbre donné qui sont effectivement visités. Ces noeuds sont caractérisés de la manière suivante : - ils vérifient *P*
- tous leurs ascendants vérifient *P*
Nous structurons cet ensemble en arbre binaire :

. description de l'information
un noeud du nouvel arbre est un noeud de l'arbre donné vérifiant *P* et dont les ascendants vérifient *P*

. description des relations

- *RACINE'* : *RACINE*
- *G'(N')* : *G(N)*
- *E_G'(N')* : *E_G(N) et-c P(G(N))*
- *D'(N')* : *D(N)*
- *E_D'(N')* : *E_D(N) et-c P(D(N))*
- *ESTDROIT'(N')* : *ESTDROIT(N)*
- *PERE'(N')* : *PERE(N)*

{*N'* désigne le noeud correspondant au noeud *N* de l'arbre donné}

Remarque : le nouvel arbre est vide si la racine de l'arbre donné ne vérifie pas *P* .

. Algorithme
. L'algorithme d'énumération partielle est alors obtenu en appliquant un parcours d'arbre adéquat.

c) Une application de l'algorithme d'énumération partielle

Une technique générale d'énumération d'un ensemble A consiste à procéder de la manière suivante (cf. notamment [Dij 71]) :
On cherche à définir un sur-ensemble B de A qui possède les caractéristiques suivantes :

1. B est "plus facile à énumérer" que A
2. il est "facile" de reconnaître les éléments de A parmi ceux de B.

Remarque :

Eventuellement, on applique la même technique pour énumérer B.

Le traitement arborescent peut alors être appliqué pour construire un algorithme :

+ l'ensemble B est structuré en arbre
+ on applique un algorithme d'énumération partielle pour retrouver les éléments de l'ensemble A.

Pour pouvoir appliquer cette méthode, il faut imposer une contrainte à la structuration en arbre de l'ensemble B : on doit pouvoir définir une propriété *P* qui assure que si un noeud vérifie *P* il existe au moins un élément de A dans le sous-arbre associé.

Remarque :
Une manière d'assurer ceci est de faire en sorte que tout élément de A se trouve, dans l'arbre B, relié à la racine par un chemin dont tous les noeuds appartiennent aussi à A.

Nous illustrons cette méthode en construisant une solution du problème classique des "8 reines".

d) Le problème des "8 reines" (cf. notamment [Flo 67], [Dij 72] ,[Ars 77], [Baa 78]).

d.1. Enoncé

Le problème consiste à placer 8 reines sur un échiquier sans qu'aucune d'entre elles ne soit en prise (une reine "prend" toute pièce se trouvant sur la même ligne, la même colonne, la même diagonale ascendante ou la même diagonale descendante qu'elle). La solution classique à ce problème consiste à caractériser une solution par une permutation P des 8 entiers 1,...,8 : cette permutation est interprétée comme suit : placer une reine à l'intersection de la colonne P(i) et de la ligne i, ceci pour toute valeur de i entre 1 et 8.

d.2. Définition d'un sur-ensemble

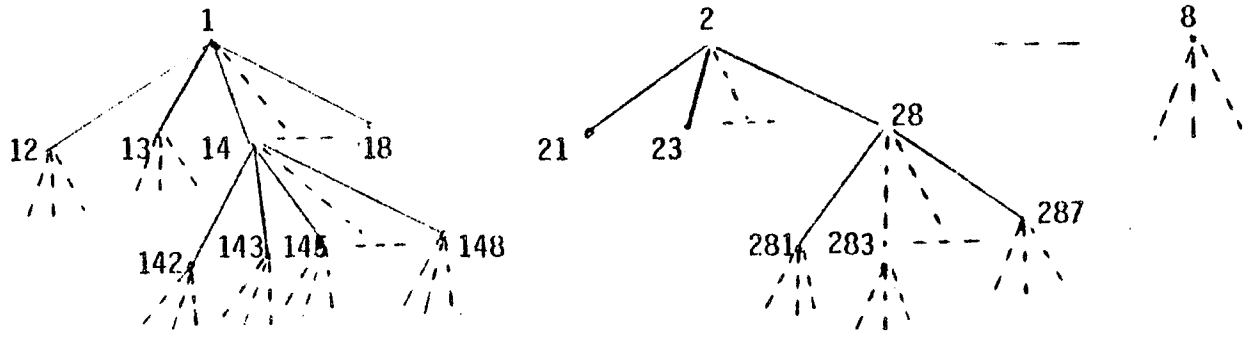
L'ensemble des solutions est un sous-ensemble de l'ensemble des permutations des 8 premiers entiers. Une permutation est une solution dans la mesure où aucune reine n'est en prise, ce qui est simplement vérifié en s'assurant que toutes les sommes i+P(i), qui caractérisent les diagonales d'équation x+y = constante, sont distinctes et que toutes les différences i-P(i), qui caractérisent les autres diagonales, le sont aussi.

d.3. Enumération du sur-ensemble

Pour énumérer l'ensemble des permutations, on peut choisir l'un des nombreux algorithmes connus (cf. [Sed 77]) et assurer à chaque permutation la vérification précédente. Par souci d'"optimisation", on peut essayer d'ordonner les permutations de telle sorte que l'on minimise les vérifications. Ayant placé K reines sur l'échiquier et ayant vérifié qu'aucune d'elles n'était en prise, on cherche à construire une ou plusieurs solutions ayant en commun la position de ces k reines (i.e. on cherche toutes les permutations solutions qui placent ces K reines dans les mêmes positions) : les vérifications concernant ces K reines ne seront ainsi faites qu'une fois.
On a ainsi une contrainte supplémentaire pour le choix de l'algorithme d'énumération des permutations : regrouper toutes les permutations ayant en commun les K premières valeurs.

d.4. Un nouveau sur-ensemble

Une solution consiste alors à énumérer l'ensemble des k-uplets d'entiers compris entre 1 et 8 et distincts deux à deux. Les permutations sont les k-uplets de taille 8. Cette énumération peut être faite en appliquant un traitement arborescent : on définit l'arbre suivant : à chaque noeud correspond un k-uplet d'entiers distincts deux à deux. Le sous-arbre de racine X comporte tous les K-uplets commençant par X. Le schéma suivant donne le début de l'arbre des permutations (k-uplets de taille 1, 2 et 3) :



d.5. Enumération partielle

La recherche des solutions est alors une énumération partielle de cet arbre dans la mesure où l'on fait la remarque suivante :
Etant donné un k-uplet de taille k, il correspond au placement de k reines sur les 8 à placer. Si deux de ces reines au moins sont en prise, toute permutation d'ordre 8 issue de ce k-uplet de taille k (i.e. appartenant au sous-arbre dont le k-uplet de taille k est racine) ne pourra être une solution. Il est donc inutile de parcourir ce sous-arbre lors de la recherche d'une solution.
Dans ces conditions, à chaque fois que l'on accède à un noeud, on sait que le k-uplet père correspond à des reines qui ne sont pas en prises : par conséquent pour vérifier que le noeud a lui aussi cette propriété, il suffit de vérifier que la dernière reine placée ne se trouve sur aucune des diagonales contrôlées par les autres reines.

Remarque :
Une solution légèrement différente consiste à énumérer partiellement l'arbre des k-uplets formés sur les 8 premiers entiers (on ne spécifie plus que les éléments sont distincts deux à deux). Comme précédemment, les solutions sont des feuilles appartenant à des sous-arbres dont les racines correspondent au placement de K reines qui ne sont pas en prises. Le parcours d'un sous-arbre n'a lieu que si cette condition est vérifiée. Pour la vérifier, on s'assure que la dernière reine placée, n'est en prise avec aucune des reines déjà placée, i.e. on considère la colonne et les diagonales auxquelles cette reine appartient. (La solution présentée en général correspond à cette énumération d'arbre. Voir les références données plus haut).

e) Conclusion

L'exemple des 8 reines est souvent utilisé pour illustrer une technique générale dite de "retour arrière" ("backtracking" - cf par exemple [Flo 67] [Vei 74] [BiR 75]) La forme générale des programmes obtenus en appliquant cette technique correspond au schéma d'énumération partielle (cf. par exemple [Wir 76], [MaB 78]). Nous venons de montrer comment utiliser la méthode du traitement arborescent pour construire de tels programmes.

5.2.3. - Recherche d'un élément dans un arbre

Nous considérons maintenant le problème général suivant : Etant donné un arbre binaire A , rechercher s'il existe un noeud dans cet arbre vérifiant une propriété caractérisée à l'aide d'un prédicat P . On veut localiser un noeud qui vérifie la propriété, s'il en existe.

a) Cas général

Pour construire l'algorithme, nous appliquons les deux formes d'analyse proposées au § 5.1.6. On obtient alors deux algorithmes :

Algorithme : Recherche d'un élément dans un arbre binaire (forme 1)

```
fonction rech (arbre A) → logique, neu :
 {recherche un élément vérifiant la propriété  P
  si on en trouve un, renvoie le couple : vrai , noeud où on a trouvé
  sinon,              renvoie le couple : faux , non défini}
 choix
   P(A)   : → vrai,A
   ¬P(A)  : choix
              E_G(A) et E_D(A) : (t,ou):=rech(G(A))
                                 choix
                                   t  : →vrai,ou
                                   ¬t : →rech(D(A))
                                 fchoix
              E_G(A) et ¬E_D(A)  : →rech(G(A))
             ¬E_G(A) et E_D(A)   : →rech(D(A))
             ¬E_G(A) et ¬E_D(A)  : →faux,A
            fchoix
 fchoix
ffonction rech
```

et la recherche dans l'arbre donné est réalisé par la séquence :

```
(t,ou) := rech(RAC)
choix
    t  : "trouvé en ou "
   ¬t  : "non trouvé"
fchoix
```

La deuxième forme d'algorithme est la suivante :

Algorithme : recherche d'un élément dans un arbre binaire (forme 2)

```
fonction rech (arbre A) → logique, neu :
  {même spécifications que ci-dessus}
  choix
      A=NIL : → faux,NIL                          {arbre vide}
      A≠NIL : choix
                  P(A)    : → vrai,A
                  ¬P(A)   : (t,ou):=rech(G(A))
                            choix
                               t : → vrai,ou
                              ¬t : → rech(D(A))
                            fchoix
              fchoix
  fchoix
ffonction rech
```

et la recherche dans l'arbre donné est effectuée par la même séquence que précédemment.

Remarque :
Il est clair que l'on peut établir le même type d'algorithme pour un arbre non binaire.

Nous étudions maintenant quelques cas particuliers importants.

b) Enumération partielle de l'arbre :

La recherche est simplifiée par le fait que l'on dispose d'un prédicat Q qui permet d'affirmer que l'élément cherché ne se trouve pas dans un arbre. On retrouve les conditions de l'énumération partielle étudiée en 5.2.2. Un tel algorithme est notamment utilisé dans tout problème de type "retour arrière" où on ne s'intéresse qu'à une seule solution.

Algorithme : Recherche d'un élément avec énumération partielle

(sur la base de la forme 2 de *rech*).

```
fonction rech_p (arbre A) → logique, neu :
     {recherche un élément caractérisé par une propriété P
     Le prédicat Q permet de tester s'il est possible qu'un tel élément
     existe dans un sous-arbre
     si l'élément est trouvé, renvoie le couple : vrai, noeud où on a trou
     sinon,                   renvoie le couple : faux , non défini}
     choix
         A= NIL  : → faux, NIL
         A≠ NIL  : choix
                     Q(A)   : → faux, A
                     ¬Q(A)  : choix
                                P(A)   : → vrai, A
                                ¬P(A)  : (t,ou):=rech_p(G(A))
                                         choix
                                           t  : → vrai, ou
                                           ¬ t: → rech_p(D(A))
                                         fchoix
                              fchoix
                   fchoix
     fchoix
ffonction rech_p
```

La recherche dans l'arbre donné est alors effectuée par la même séquence que précédemment.

Remarque :

On peut illustrer ceci par l'exemple suivant :
on définit un "*tas*" ("*heap*", cf. [Knu 73]) de la manière suivante :
un "tas" est ensemble structuré en arbre binaire de telle sorte que pour tout noeud N , on ait les relations :

$N \leq G(N)$ {où $\leq$ est une relation d'ordre total défini
$N \leq D(N)$ sur l'ensemble}

L'algorithme ci-dessus peut alors etre appliqué pour effectuer la recherche dans un ensemble structuré en tas. Ceci est à la bas d'une méthode de tri performante ("heapsort" cf. [Wil 64].

c) Recherche dichotomique

c.1 . "arbre binaire de recherche"

Ce deuxième cas particulier est caractérisé par l'hypothèse suivante Il existe un prédicat R défini sur l'ensemble des noeuds de l'arbre binair considéré, et permettant de tester si un sous-arbre peut contenir l'élémen cherché. On peut spécifier R de la manière générale suivante :

X étant un noeud quelconque de l'arbre binaire :

. $R(X) = vrai \Rightarrow \forall E \in D(X),\quad P(E) = faux$
. $R(X) = faux \Rightarrow \forall E \in G(X),\quad P(E) = faux$

Nous appelons un tel arbre : "arbre binaire de recherche, ordonné par le prédicat R".

c.2. Construction de l'algorithme

Nous construisons un algorithme en appliquant la première forme d'analyse récurrente et nous obtenons :

Algorithme : recherche dichotomique d'un élément dans un arbre binaire

```
fonction rech_d (arbre A) → logique, neu :
  {recherche d'un élément vérifiant une propriété P dans un "arbre binaire
   de recherche" ordonné par un prédicat R.
   Si l'élément est trouvé, renvoie le couple : vrai, noeud où on a trouvé
   sinon                  , renvoie le couple : faux, indéfini}
   choix
       P(A)   : → vrai, A
       ¬P(A)  : choix
                  ¬R(A) : choix
                             E_D(A)  : → rech_d(D(A))
                             ¬E_D(A) : → faux, A
                          fchoix
                  R(A)  : choix
                             E_G(A) : → rech_d(G(A))
                             ¬E_G(A):→faux, A
                          fchoix
                fchoix
   fchoix
ffonction rech_d
```

La recherche est alors effectuée par la même séquence que précédemment.

d) Exemple d'application de la recherche dichotomique

Nous considérons le problème suivant :
Etant donné un vecteur trié selon une relation d'ordre noté ≤, rechercher le premier élément ayant une valeur donnée.

Remarque :
Nous avons déjà construit une solution à ce problème, par application du traitement séquentiel (cf. § 1.2.4.).

Nous appliquons le traitement arborescent de la manière suivante :
(VT désigne le vecteur, N le nombre d'éléments du vecteur et on suppose que la borne inférieure de VT est 1).

1. Raffinement de l'information

Nous considérons un ensemble de sous-vecteurs du vecteur donné obtenus les uns à partir des autres en les coupant en deux. Chaque sous-vecteur est désigné par un couple de positions dans VT correspondant à ses bornes.
(on pourrait aussi caractériser chaque sous-vecteur par sa borne inférieure et sa longueur).

2. Structuration en arbre binaire d'un vecteur

- la racine est le vecteur
- le sous-arbre gauche est obtenu en structurant en arbre binaire le premier demi-vecteur
- le sous-arbre droit est obtenu en structurant en arbre binaire la deuxième demi-vecteur.

3. Description de l'arbre binaire

- description de l'information

  Un noeud est un sous-vecteur de *VT*, désigné par un couple de positions (i,s) dans *VT* correspondant aux bornes inférieure et supérieure du sous-vecteur.

- description des relations

  - *RACINE* : 1, *N*
  - *G(i,s)* : *i*, ⌊(*i+s*) *div* 2⌋*-1*
  - *D(i,s)* : ⌊(*i+s*) *div*2⌋*+1*, *s*
  - *E_G(i,s)*: $i+1 \leq (i+s)$ *div* $2-1$ {soit $s \geq i+2$}
  - *E_D(i,s)*: $s-1 \geq (i+s)$ *div* $2+1$ {soit $s > i$}
  - la fonction *PERE* n'est pas définie
  - *INFO(i,s)*: *VT*⌊(*i+s*) *div* 2⌋ {seul l'élément "milieu" nous intéresse}

- Algorithme

  On applique alors l'algorithme précédent, sachant que si X est la valeur cherchée, les prédicats P et R ont pour sens :

  P(i,s) : *X=VT*⌊(*i+s*) *div* 2⌋ {i.e.*X=info(i,s)*}
  R(i,s) : *X≤VT*⌊(*i+s*) *div* 2⌋ {i.e.*X* ≤ *INFO(i,s)*}

Remarque :

Le nombre maximal de comparaisons est égal à la profondeur de l'arbre, soit $\lfloor \log_2 N + 1 \rfloor$. On trouvera une étude détaillée du coût de la recherche dichotomique dans [MeB 78].

5.2.4. - Conclusion

Nous avons poursuivi la présentation du traitement arborescent selon les mêmes lignes méthodologiques que pour le traitement séquentiel : en conservant la même classification des algorithmes, nous avons construit les schémas du traitement arborescent. Nous avons ainsi illustré les principes donnés au paragraphe précédent (§ 5.1.) et nous avons montré le réalisme de notre propos par l'importance pratique des problèmes qui ont été étudiés ("parcours d'arbres", "retour arrière", "recherche dichotomique", "évaluation postfixée").

5.3 - UN EXEMPLE D'APPLICATION DU TRAITEMENT ARBORESCENT : ETUDE D'UN MODELE DE REDUCTION LOGARITHMIQUE

Nous développons ici un modèle de réduction logarithmique du coût d'un algorithme de recherche dans une file. Le traitement arborescent nous permet de faire l'analyse de ce modèle et de formuler les algorithmes optimisés.

5.3.1. - Le problème considéré

Le problème que nous étudions est le suivant :
On considère une suite infinie de valeurs croissantes définie sous forme récurrente :

$$\cdot\ u_o = A$$
$$\cdot\ u_{n+1} = f(u_n) \text{ et } u_{n+1} \geq u_n \text{ pour tout } n$$

On cherche à résoudre les deux problèmes suivants :

Problème 1 : calculer u_n pour $n > o$ donné

Problème 2 : étant donné une valeur Y , $(Y \geq A)$ trouver n tel que $u_n \leq Y < u_{n+1}$ on suppose que ce n existe.

5.3.2. - Application du traitement séquentiel

On structure en file l'ensemble des u_i , en utilisant directement la définition récurrente donnée. (un élément u_i est caractérisé par l'entier i et la valeur u_i). Le problème 1 est une "recherche par comptage" (cf. 1.2.4.b dans cette file. Le problème 2 est une recherche associative (§ 1.2.3.)

dans la file des couples $\{u_i, u_{i+1}\}$. Les algorithmes que nous obtenons sont simplifiés par rapport à ce qui est donné au chapitre 1, dans la mesure où l'on sait a priori que les recherches vont aboutir.

Problème 1		Problème 2	
{calcul de u_n pour $n \geq 0$}		{calcul de n tel que $u_n \leq Y < u_{n+1}$ pour $Y \geq A$}	
$r:=A$; $i:=0$	1	$r:=A$; $i:=0$	1
tantque $i \neq n$ *faire*	n+1	*répéter*	
$\{r=u_i\}$		$\{r=u_i$ *et* $Y \geq r\}$	
$i:=i+1$; $r:=F(r)$	n	$i:=i+1$; $r:=F(r)$	n+1
ftantque		*jusqua* $Y<r$	n+1
$\{r=u_n\}$		$\{r=u_i$ *et* $u_{i-1} \leq Y < u_i\}$	
{le résultat est r}		{le résultat est $i-1$}	
évaluations :		Evaluations :	
accés séquentiel : F :n		accés séquentiel : F:n+1	
contrôle : n+1		contrôle : n+1	
affectations : 2n+2		affectations : 2n+4	
comparaisons : n+1		comparaisons : n+1	
opérations "+1" : n		opérations "+1" : n+1	

Clairement le coût est d'ordre n dans les deux algorithmes.
C'est ce coût que nous cherchons à réduire. L'application du traitement arborescent va nous permettre de trouver dans quelles conditions on peut obtenir une réduction logarithmique de ce coût.

5.3.3. - Application du traitement arborescent

Nous structurons l'ensemble des éléments u_i sous forme d'un arbre binaire et appliquons les algorithmes de recherche. La réduction logarithmique sera obtenue si l'on peut appliquer l'algorithme de recherche dichotomique à un arbre de profondeur minimale.

Remarque :

Dans le problème 1 la profondeur minimale est atteinte lorsque les éléments $u_1, \ldots, u_n$ sont organisés sous forme d'un arbre binaire complet. (nous supposons $n>0$). La profondeur est alors $\lfloor \log_2 n + 1 \rfloor$.
Dans le problème 2, on a un élément de plus (u_{n+1}) et la profondeur minimale est $\lfloor \log_2 (n+1) + 1 \rfloor$.
On peut aussi utiliser un arbre binaire équilibré.

1. Structuration en arbre binaire

Le principe de cette structuration est le suivant :
a la file des n éléments $u_1, u_2, \ldots, u_n$ on associe l'arbre binaire suivant :

. la racine est l'élément milieu u_m $(m = (n+1) \text{ div } 2)$
. le sous-arbre gauche est obtenu en structurant en arbre la sous-file $u_1, u_2, \ldots, u_{m-1}$
. le sous-arbre droit est obtenu en structurant en arbre la sous-file $u_{m+1}, \ldots, u_n$.

Pour pouvoir réaliser cette structuration il faut connaître la valeur de n a priori. Ce n'est pas le cas dans le problème 2. On cherche alors à connaître a priori une borne supérieure *nsup* de n . La méthode couramment proposée est de choisir la plus petite puissance de 2 supérieure à n :

$$nsup = 2^{k+1} \text{ avec } 2^k \leq n < 2^{k+1}, \text{ soit } k = \lfloor \log_2 n \rfloor$$

Remarques :

. . Ce choix se justifie par trois raisons :
- le principe de structuration précédent est plus facile à appliquer si le nombre d'éléments est une puissance de 2.
- l'arbre binaire structurant les *nsup-1* premiers éléments de la file a pour profondeur k+1, c'est-à-dire la même profondeur que l'arbre binaire complet structurant les n premiers éléments.
- le calcul de *nsup* peut se faire (lorsque notre méthode est applicable, nous le verrons ci-dessous) en k étapes.

. On peut, pour les mêmes raisons, utiliser le même arbre pour résoudre le problème 1 (ce que nous faisons).

L'arbre binaire que l'on obtient par ce procédé de structuration est un "arbre binaire de recherche ordonné par la relation $\leq$ (§ 5.2.3.,c) : en effet la suite u_i étant croissante, la racine de l'arbre a une valeur - inférieure ou égale aux valeurs de tous les noeuds du sous-arbre droit
- supérieure ou égale aux valeurs de tous les noeuds du sous-arbre gauche.

On pourra donc lui appliquer la recherche dichotomique.

Pour étudier dans quelles conditions cette structuration est possible, nous devons préciser la description de l'arbre binaire.

4. Algorithmes

Nous pouvons maintenant utiliser l'algorithme de recherche dichotomique pour résoudre nos deux problèmes.

a) Problème 1

algorithme : *calcul de* u_n $(n > 0)$

```
{On cherche le noeud de numéro n dans l'arbre précédent, par recherche dichotomique}
  fonction r1 (entier x,t ; valeur vf) → valeur :
  {adaptation de l'algorithme de recherche dichotomique : on sait
   que la valeur cherchée est dans l'arbre}
   choix
       x = n  :→ vf
       x ≠ n : choix
                    x < n : r1(x+t,t/2,P(vf,t))
                    x > n : r1(x-t,t/2,M(vf,t))
               fchoix
   fchoix
 ffonction R1

 {pour calculer u_n il faut déterminer la racine de l'arbre}
 vf:=F(A) ; x:=1
 itérer x:=2*x                          {x=2^(i+1), vf=u_(x/2), i > 0}
 arrêt : x > n
         vf:=FF(vf)
 fitérer
 {vf=u_(x/2)}
 {x/2≤n<x  et x  est une puissance de 2}
 résultat:=r1(x/2,x div 4 vf)
```

Evaluation : soit $k = \lfloor \log_2 n \rfloor$

L'itération de départ (calcul de la racine) est exécutée k+1 fois. Ceci entraîne k appels de la fonction *FF* et k+1multiplications de x par 2 ainsi que k+1 tests.

La valeur n que l'on cherche dans l'arbre est atteinte en parcourant un chemin de longueur k au maximum. Dans le pire des cas, on appelle k fois l'une ou l'autre des fonctions *M* et *P* . On effectue k divisons par 2 et k additions ou soustractions. Enfin, on effectue 2k+1 tests.

La réduction logarithmique n'est effective que dans la mesure où le coût des calculs *FF* , *M* et *P* est de même ordre que celui de *F* .

2. Description de l'arbre binaire

. Un noeud est caractérisé par trois valeurs x, t et v

x est un entier (rang de l'élément u_x associé au noeud)

t est une puissance de 2 (taille des sous-arbres du noeud)

$v = u_x$ (valeur de l'élément u_x associé au noeud).

. Les relations sont :

$RACINE$: $nsup\ 2$ $nsup2$, u_{nsup2} {où $k = log_2 n$, $nsup2 = 2^K$}

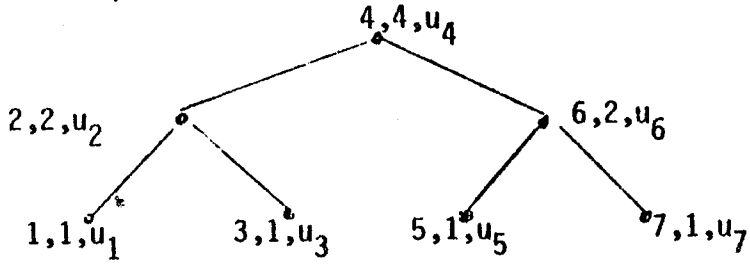
$G(x,t,u_x)$: $x-t/2$, $t/2$, $u_{x-t/2}$

$D(x,t),u_x)$: $x+t/2$, $t/2$, $u_{x+t/2}$

$E\_G(x,t,u_x)$: $t \neq 1$

$E\_D(x,t,u_x)$: $t \neq 1$

Cet arbre est un arbre binaire complet de $2^{k+1}-1$ noeuds et de profondeur k. Par exemple pour k = 2 (valeurs de n comprises entre 4 et 7), on a :



3. Conditions d'application de la structuration

Pour pouvoir définir cet arbre, il faut pouvoir définir les primitives ci-dessus. Il faut donc pouvoir calculer

. u_{nsup2}

. $u_{x+t/2}$ en fonction de x,t et u_x

. $u_{x-t/2}$ en fonction de x,t et u_x

Nous formalisons ceci à l'aide de trois fonctions FF, M et P définies comme suit :

. $FF(u_x) = u_{2x}$

. $M(u_x,h) = u_{x-h}$

. $P(u_x,h) = u_{x+h}$

b) Problème 2

Algorithme : calcul de n tel que $u_n \leq Y < u_{n+1}$, pour Y donné

$(Y \geq u_o)$

{On cherche Y dans l'arbre précédent par recherche dichotomique. Cet arbre n'est défini (n'est pas vide) que si $Y \geq u_1$}

```
fonction r2 (entier x,t ; valeur vf) → entier
  choix
      Y = vf  : → x
      Y > vf  : choix
                  t≠1 : → r2(x+t,t/2,P(vf,t))
                  t=1 : → x
                fchoix
      Y < vf  : choix
                  t≠1 :→r2(x-t,t/2,M(vf,t))
                  t=1 :→x-1
                fchoix
  fchoix
ffonction R2
{pour calculer n il faut déterminer la racine de l'arbre}
{si Y < u1 , l'arbre est vide et le résultat est trivial}
choix
    Y < F(A) : résultat :=0
    Y ≥ F(A) : vf:=F(A) ; x:=1
               tantque vf ≤ y faire vf:=FF(vf) ; x:=2*x ftantque
               résultat := r2(x/2,x/2,M(vf,X/2)
fchoix
```

Evaluation : soit $k = \lfloor \log_2 n \rfloor$

L'itération de départ est exécutée k+1 fois, ce qui entraîne : k+1 appels de FF, k+1 multiplications par 2 et k+2 tests.

Lorsque la valeur Y n'est pas dans l'arbre (cas pire), ceci est détecté en traitant une feuille. On a alors parcouru un chemin de longueur k, ce qui a entraîné : k appels de l'une ou l'autre des fonctions P ou M, k additions ou soustractions, k divisions par 2 et 3(k+1) tests.

Ici encore, la réduction logarithmique n'est effective que dans la mesure où le coût des fonctions FF, M et P est de même ordre que celui de F.

5.3.4. - Exemples :

a) Multiplication, division entière :

Nous considérons la suite des multiples d'un entier b donné :

$$\begin{cases} u_o = 0 \\ u_n = u_{n-1} + b \end{cases} \qquad \text{soit } u_n = n * b$$

on a par ailleurs :

$$u_{2x} = 2 * u_x$$
$$u_{x+h} = u_x + u_h$$
$$u_{x-h} = u_x - u_h$$

Le problème 1 correspond au calcul $n * b$, le problème 2 correspond au calcul du quotient et du reste de la division entière de Y donné par b. En effet, encadrer Y donné par deux valeurs successives de la suite u s'écrit :

calculer n tel que : $n * b \leq Y < (n+1)*b$
soit en posant : $r = Y - n * b$
calculer n tel que : $Y = n * b + r$ et $0 \leq r < b$

Nous adaptons les résultats précédents, en associant à chaque noeud x,t,u_x la valeur supplémentaire u_t (parce que u_{x+t} et u_{x-t} dépendent de u_t et non de t) :

```
RACINE          :2^k, 2^k, u_2k, u_2k
G(x,t,vx,vt)    : x-t/2, t/2, vx-vt/2, vt/2
D(x,t,vx,vt)    : x+t/2, t/2, vx+vt/2, vt/2
E_G(x,t,vx,vt):t≠1              {ou bien vt≠b}
E_D(X,t,vx,vt):t≠ 1             {ou bien vt≠b}
```

Les algorithmes se déduisent alors directement de ceux donnés ci-dessus. Par exemple le calcul du reste et du quotient sera basé sur la fonction suivante :

```
fonction r2_div(entier,x,t,vx,vt) → entier, entier  {quotient, reste }
  choix
      Y=vx : → x,0
      Y>vx :  choix
                  t=1 : → x,Y-vx
                  t≠1 : → r2_div(x+t/2,t/2,vx+vt/2, vt/2)
              fchoix
      Y<vx :  choix
                  t=1 : → x-1, Y-vx+b
                  t≠1 : →  r2_div(x-t/2,t/2,vx-vt/2,vt/2)
              fchoix
  fchoix
ffonction r2-div
```

et la séquence d'appel est la suivante :

```
choix
    Y < b : reste:=y ; quotient:=0
    Y > b : vx:=b ; x:=1
            tantque vx ≤ Y faire vx:=2*vx ; x:=2*x ftantque
            quotient, reste :=r2_div(x/2,vx/2,vx/2,vx/2)
fchoix
```

Remarques :

. Il est clair qu'à partir de cet algorithme on peut aboutir à une forme itérative d'algorithme (en appliquant par exemple les résultats du chapitre 4) dans laquelle les calculs sont organisés de manière à ne calculer la même valeur qu'une fois.

. On peut rapprocher cette solution du problème à celle que nous avons donnée au chapitre 4 : les ordres de grandeurs des coûts sont les mêmes.

b) D'autres exemples peuvent être développés de la même manière en considérant :

- la suite des carrés des entiers (calcul de x^2 et $\sqrt{Y}$)
- la suite des puissances d'un entier b donné (calcul de b^n et de $\log_b Y$).

5.3.5. - Une autre forme d'accélération logarithmique

Nous illustrons la même démarche en proposant une autre manière de résoudre le problème 1 : on se base sur la structuration en arbre binaire suivante :

Description de l'arbre :

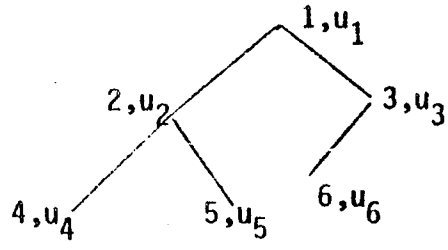
. chaque noeud est un couple (x, u_x)

. les primitives de l'arbre sont :

$RACINE : 1, u_1$
$G(x,v) : 2*x, u_{2*x}$
$D(x,v) : 1+2*x, u_{1+2*x}$
$E\_G(x,v) : 2*x \leq n$
$E\_D(x,v) : 2*x \leq n-1$

{ceci est un arbre binaire complet de profondeur $\log_2 n$}

Pour pouvoir définir cet arbre, il faut pouvoir calculer les quantités u_{2x} et u_{2x+1} à partir de u_x
Le calcul de u_b pour n donné correspond à rechercher n dans cet arbre.
Pour n=6 on a l'arbre suivant :



Comme précédemment, nous supposons qu'il existe un moyen de calculer les quantités nécessaires à la définition de l'arbre soit l'existence de deux fonctions FF0 et FF1 définies par :

$$FF0(u_x) = u_{2x}$$
$$FF1(u_x) = u_{2x+1}$$

Remarque : FF1 est la composition de FF et de F.

Dans ces conditions l'accélération logarithmique du calcul de u_n pour donné est obtenue dès lors que l'on connait la représentation de n en base 2. En effet l'arbre précédent répartit les valeurs de n suivant cette représentation binaire.
Si b est un tableau de bits correspondant à la représentation de n en base 2 (i.e. $n=b_0+2b_1+2^2b_2+\ldots+2^kb_k$ où $k = \log_2 n$),on peut écrire l'algorithme suivant :

Algorithme : calcul de u_n connaissant n et b.

```
{On applique l'algorithme de recherche dichotomique :
 x,u_x étant un noeud de niveau i, la valeur de b_{k-i} indique où se trouve
 le noeud n,u_n : dans le sous-arbre gauche si b_{k-i}=0 , dans le sous-arbre
 droit si b_{k-i}=1}.
     fonction r3 (entier x,i ; valeur v) → valeur
     {étant donné le noeud x,v de niveau i cherche le noeud n, u_n}
     {b est la représentation binaire de n}
     choix
         x=n  : → v
         x≠n  : choix
                b_{k-i} → b_i=0 : → r3(2*x,i+1,FFo(v))
                        → b_i=1 : → r3(1+2*x,i+1,FF1(v))
                fchoix
     fchoix
    ffonction r3
```

La séquence d'appel est alors :

```
calcul de b
résultat :=r3(1,1,u_1)
```

Evaluation : Il est clair que l'exécution de cet algorithme correspond à cheminer dans l'arbre de la racine au noeud n, u_n : ce chemin est de longueur k : on appelle donc k fois l'une ou l'autre des fonctions *FFO* et *FF1* ;on effectue k multiplications par 2 et au plus k incrémentations ; le nombre de tests se monte à 2k+1. Enfin le coût du calcul de b est d'ordre k.

Exemples : On peut reprendre les exemples précédents et appliquer cet algorithme pour résoudre le problème 1.

Remarque :
Tout ce que nous venons dire s'applique à des suites décroissante Ceci n'influe pas sur les solutions proposées pour le problème 1 (algorithmes mettant en jeu les fonctions r1 et r3). En ce qui concerne le problème 2, il suffit de changer les tests de manière adéquate : la suite étant décroissante, on trouvera en sous-arbre gauche d'un noeud les valeurs supérieures à celle du noeud et en sous-arbre droit les valeurs inférieures à celle du noeud.

5.3.6. - Conclusion

Nous avons énoncé deux problèmes généraux de recherche dans une suite. Nous avons appliqué le traitement séquentiel et obtenu un premier algorithme pour chacun des problèmes. Pour améliorer ces algorithmes, nous avons cherché à appliquer le traitement arborescent. Nous avons ainsi pu mettre en évidence les hypothèses supplémentaires que la suite doit satisfaire, pour qu'une réduction logarithmique soit possible. Nous avons alors donné les algorithmes correspondants. Enfin nous avons illustré la même démarche en proposant une autre application du traitement arborescent permettant aussi d'obtenir une réduction logarithmique de l'un des problèmes.

Ces résultats illustrent l'intérêt des méthodes que nous pratiquons. Nous avons pu, par l'intermédiaire du traitement séquentiel et du traitement arborescent, unifier les solutions apportées à des problèmes divers (cf. l'étude séparée de divers exemples dans [Dij 76], [Ars 77], [Cha 77], [Dij 78]). et présenter un modèle d'analyse systématique conduisant le cas échéant, à une amélioration sensible des algorithmes.

5.4 - CONCLUSION (sur le chapitre 5)

Résoudre un problème en appliquant le traitement arborescent consiste à :

1 - structurer l'information en arbre, après l'avoir raffinée en un ensemble d'informations élémentaires.

2 - guider la construction d'un algorithme par cette interprétation de l'information :
   - soit en appliquant un modèle général d'analyse de traitements d'arbres,
   - soit en identifiant un problème général de parcours d'arbre ou de recherche dans un arbre, et en appliquant le schéma correspondant.

Comme pour le traitement séquentiel, nous avons proposé des outils permettant d'appliquer ce principe de construction d'algorithmes de manière systématique :

. un modèle du processus de structuration d'un arbre
. des notations pour la description d'un arbre
. un modèle d'analyse récurrente des traitements d'arbres
. des algorithmes généraux de parcours d'arbres et de recherche dans un arbre.

Nous avons fait apparaître l'intérêt pratique de notre méthode en décrivant plusieurs techniques importantes (schémas de "retour arrière", recherche dichotomique, réduction logarithmique) à l'aide du traitement arborescent.

Soulignons enfin qu'une certaine unité dans la démarche méthodologique est apparue dans ce chapitre, par la mise en évidence constante des analogies entre le traitement séquentiel et le traitement arborescent.

CHAPITRE 6

APPLICATION DU TRAITEMENT SEQUENTIEL AU TRAITEMENT D'ARBRES

Nous poursuivons la présentation du traitement arborescent en nous plaçant dans le contexte d'un langage de programmation conventionnel, comme nous l'avons fait pour le traitement séquentiel (chapitre 2). Nos objectifs sont les mêmes :

. décrire les étapes supplémentaires du processus de construction de programmes, liées au niveau de programmation choisi.

. élaborer un ensemble de schémas dans un environnement de programmation standard (illustrant ainsi le caractère pratique et général de la méthode).

Comme pour le traitement séquentiel, le processus de construction de programmes de traitement arborescent comporte trois phases supplémentaires

1. élaboration d'un répertoire d'actions élémentaires permettant de décrire le processus dynamique de "déplacement" dans un arbre. Cette phase est rendue systématique par le recours à un modèle de "machine arbre". Les notations proposées nous permettent de décrire les algorithmes en faisant abstraction de la représentation effective de l'arbre considéré.

2. construction d'un algorithme itératif à la suite d'une analyse du problème conduite selon les principes présentés au chapitre 5.
   Pour réaliser cette construction, nous proposons 2 méthodes :
   a) L'application du traitement séquentiel : la forme récursive permet de définir la file caractéristique du traitement séquentiel, et d'interpréter le problème en termes de cette file. Un algorithme itératif est alors obtenu en appliquant l'ensemble des résultats de la première partie.
   Nous utilisons cette méthode pour construire les algorithmes itératifs de parcours d'arbres.

b) L'interprétation du problème posé en termes de parcours d'arbres et l'utilisation directe des schémas de parcours.
Nous illustrons cette méthode en étudiant les algorithmes de recherche dans un arbre.

3. Simplification du programme obtenu par des transformations simples, en tenant compte des particularités du problème étudié par rapport au schéma général employé.

L'étude systématique des parcours d'arbres nous permet d'illustrer ces trois phases et de montrer qu'au niveau d'abstraction où nous nous plaçons on peut concevoir de nombreuses formes d'algorithmes en variant l'approche initiale du problème, les raffinements successivement appliqués au cours de la construction et les hypothèses de départ sur les propriétés de l'arbre considéré.

Comme dans la première partie, nous évaluons systématiquement les algorithmes produits : ces évaluations ont un triple intérêt (discuté au § 2.2.1.) : vérification de la cohérence des programmes, étude du comportement des diverses parties des programmes, analyse des coûts dans un environnement fixé.

6.1. - *"ACCES ARBORESCENT"* : UN MODELE DE *"MACHINE ARBRE"*

6.1.1. - *"Machine arbre"*

Une *"machine arbre"* est un répertoire d'actions élémentaires permettant de décrire le processus dynamique de "déplacement" dans un arbre. (nous disons *"l'accés arborescent"* à l'arbre donné).

La description d'une machine arbre est analogue à la description d'une machine séquentielle.
Elle comporte :

1. La description des informations caractérisant l'état de la machine.
   A tout moment, on doit pouvoir connaître le noeud de l'arbre sur lequel on est placé (auquel les déplacements antérieurs nous ont conduits). Nous appelons ce noeud, le *"noeud courant"* de l'accès arborescent.

2. La description du répertoire de primitives d'accès arborescent qui sont réparties en deux groupes selon qu'elles permettent de consulter l'état de la machine ou de le modifier.

6.1.2. - Un modèle de machine arbre (notations) :

Nous proposons deux ensembles de notations selon que l'on considère un arbre binaire ou non. L'effet des primitives est décrit en termes des notations du § 5.1.3.

a) Machine arbre binaire :

1. Description de l'état

On décrit la représentation de l'arbre et le mécanisme définissant le noeud courant.

2. Description du répertoire

. Consultation

- *NOEUD_COURANT* en abrégé *NC*
  désigne le noeud courant.
- *INFO_COURANT* en abrégé *INF_C*
  désigne l'information propre au noeud courant
- *EST_RACINE* en abrégé *EST_RAC*
  est un prédicat indiquant si le noeud courant est la racine *(NC=RACINE)*
- *FINAL_GAUCHE* en abrégé *FIN_G*
  est un prédicat indiquant si le noeud courant n'a pas de fils gauche *($\neg$E_G(NC))*.
- *FINAL_DROIT* en abrégé *FIN_D*
  est un prédicat indiquant si le noeud courant n'a pas de fils droit *($\neg$E_D(NC)*

- *A_DROITE*

  est un prédicat indiquant si le noeud courant est un fils droit *(ESTDROIT(NC))*.

Remarque :

Lorsque l'on s'intéresse à la notion d'arbre vide, on pourra vérifier que le noeud courant est la racine du sous-arbre vide à l'aide du prédicat :

EST_NIL (NC=NIL)

. Modifications

- *AVANCER_GAUCHE* en abrégé *AV_G*

  remplace le noeud courant par son fils gauche : *NC:=G(NC)*

- *AVANCER_DROIT* en abrégé *AV_D*

  remplace le noeud courant par son fils droit : *NC:=D(NC)*

- *AVANCER_PERE* en abrégé *AV_P*

  remplace le noeud courant par son père : *NC:=PERE(NC)*

Remarque :

Ces modifications ne peuvent être faites que si le noeud courant a un fils gauche dans le cas de *AV_G* , un fils droit dans le cas *AV_D* n'est pas la racine dans le cas de *AV_P*

Tout accés à l'arbre commence par l'initialisation du noeud courant au noeud racine. Ceci est noté à l'aide de la primitive :

- *DEMARRER_RACINE* en abrégé *D_RAC*

  initialise le noeud courant par la racine : *NC:=RACINE*

b) machine arbre (non binaire)

1. Etat : comme pour la machine arbre binaire
2. Répertoire :

. consultation

-*FINAL_FILS* en abrégé *FIN_FILS* est un prédicat indiquant si le noeud courant n'a pas de fils

-*FINAL_FRERE* en abrégé *FIN_FRERE* est un prédicat indiquant si le noeud courant n'a pas de frère.

. Modifications

- *AVANCER_FILS* en abrégé *AV_FILS*

  remplace le noeud courant par son fils s'il existe : *NC:=FILS(NC)*

- *AVANCER_FRERE* en abrégé *AV_FRERE*

  remplace le noeud courant par son frère s'il existe : *NC:=FRERE(NC)*

Remarque :
Les primitives *EST_RAC* , *EST_NIL* , *INF_C* sont utilisées avec le même sens que dans le cas de la machine arbre binaire.

c) Un exemple de description d'un accès arborescent

Nous reprenons l'exemple des combinaisons de k objets pris parmi n, que nous avons développé au § 5.1.4.-b . L'arbre est un ensemble de k-uplets chacun d'eux correspondant à une combinaison de k objets pris parmi les n objets donnés. Nous choisissons de représenter un k-uplet à l'aide d'un tableau de n entiers dont seuls les k premiers sont significatifs. On définit alors la machine arbre suivante à partir de l'étude que nous avions faite au § 5.1.4.-b.

Machine arbre des combinaisons :

Etat : un tableau C de n entiers de bornes 1 et N
Le noeud courant est caractérisé par un état du tableau C et un entier K correspondant à la taille du k-uplet courant.
C[1..K] est la combinaison associée au noeud courant.

Modifications :

D_RAC : $C(1):=1$; $k:=1$
AV_FILS : $k:=k+1$; $C(k):= C(k-1)+1$
AV_FRERE : $C(k):=C(k)+1$
AV_P : $k:=k-1$

Consultations :

EST_RAC : $k=1$ et $c(1)=1$
FIN_FILS : $C(k)=n$
FIN_FRERE: $C(k)=n$

Ainsi à l'aide de ces primitives, on pourra énumérer l'ensemble des combinaisons de k objets pris parmi n tout'en ne représentant en mémoire qu'une seule de ces combinaisons. (cf.[LuS 79]).

6.1.3. - Réalisation de certaines primitives de la machine arbre

Nous proposons ici des techniques systématiques permettant de réaliser les primitives *PERE* et *ESTDROIT* quels que soient les choix faits pour la représentation de l'arbre.

a) Réalisation de la primitive *PERE*

Lorsque la fonction *PERE* n'est pas directement définie, on peut toujours appliquer au problème le traitement arborescent suivant :
On définit un nouvel arbre à partir de l'arbre donné :

. Description de l'information

A chaque noeud de l'arbre est associé le chemin qui conduit de la racine à ce noeud (i.e. la liste des noeuds qui constituent ce chemin). C étant un tel chemin, nous utilisons les notations suivantes :

- $C \uparrow n$ est un chemin obtenu à partir de C en rajoutant le noeud n à la fin.
- $\downarrow C$ est un chemin obtenu à partir de C en enlevant le dernier élément
- C_{ext} désigne le noeud qui se trouve à l'extrémité du chemin C

. Description des relations :

RACINE	: *RACINE, chemin vide*	
G'(X,C)	: *G(X), C↑X*	{*X* désigne un noeud,
D'(X,C)	: *D(X), C↑X*	*C* est le chemin conduisant de
E_G'(X,C)	: *E_G(X)*	la racine à *X*}
E_D'(X,C)	: *E_D(X)*	
ESTDROIT'(X,C)	: *ESTDROIT(X)*	
PERE'(X,C)	: $C_{ext/\downarrow C}$	

Il est clair que la machine arbre se déduit directement de cette définition une fois que l'on a fait le choix de représentation d'un chemin : il s'agit d'une liste d'éléments sur laquelle on ne fait des modifications et/ou consultations qu'à une extrémité. Un chemin sera donc représenté à l'aide d'une pile.

Remarque :
La même technique peut être appliquée dans le cas d'un arbre non binaire.

b) Réalisation de la fonction *ESTDROIT*

Nous proposons deux manières de réaliser la fonction *ESTDROIT*

. à chaque noeud de l'arbre, on associe une valeur logique indiquant si le noeud est fils gauche ou fils droit. De plus on associe au noeud la liste des directions (à gauche ou à droite) suivies pour aller de la racine jusqu'à lui. Les primitives de la machine arbre sont alors réalisées de manière analogue à ce que nous venons de faire pour la fonction *PERE*

. une autre solution est basée sur la numérotation suivante des noeuds (cf. § 5.1.4.-a) :

```
NUM(RACINE) = 1
NUM(G(X))   = 2*NUM(X)        {si X a un fils gauche}
NUM(D(X))   = 1+2*NUM(X)      {si X a un fils droit}
```

Tout noeud qui est fils droit a un numéro impair. Cette propriété permet de réaliser simplement la fonction *ESTDROIT*. On définit un nouvel arbre à partir de l'arbre donné dans lequel on associe à chaque noeud son numéro dans la numérotation précédente.

. Description de l'information :

Un noeud X de l'arbre donné et son numéro N

. Description des relations

```
RACINE'          :  RACINE,1
G'(X,N)          : G(X), 2*N
D'(X,N)          : D(X), 1+2*N
PERE'(X,N)       : PERE(X), N div 2          {division entière}
E_G'(X,N)        : E_G(X)
E_D'(X,N)        : E_D(X)
ESTDROIT'(X,N)   : impair N
```

Remarque :

Si l'on veut comparer les deux solutions, on voit que la première consiste à gérer une pile de valeurs logiques et la seconde à gérer un entier sur lequel on ne fait que les opérations *2, div2, +1. Il est clair qu'à un niveau machine ces deux solutions sont équivalentes. On choisira l'une ou l'autre selon le niveau de programmation auquel on se place et le répertoire d'instructions disponibles.

6.1.4. - Quantités intervenant dans l'évaluation des algorithmes (notations)

Nous évaluons les algorithmes itératifs portant sur les arbres, en supposant connues les quantités suivantes :
- le nombre de noeuds de l'arbre, noté N
- le nombre de sous-arbres gauches non vides, noté FG
- le nombre de sous-arbres droits non vides, noté FD
- le nombre de feuilles, noté NF

Il est clair d'après ces définitions, que le nombre de noeuds qui sont fils gauche est FG, et que le nombre de noeuds qui sont fils droit est FD. Par ailleurs on a clairement N = FG + FD + 1. Cette relation exprime une partition sur l'ensemble des noeuds de l'arbre : la racine, les fils gauches et les fils droits.

Nous faisons maintenant apparaître d'autres quantités qui permettent de dénombrer les cas obtenus dans l'analyse par cas proposée en 5.1.6. :
- le nombre de noeuds ayant deux fils, noté NT2
- le nombre de noeuds ayant un seul fils, le gauche, noté NT1G
- le nombre de noeuds ayant un seul fils, le droit, noté NT1D
  (le quatrième cas correspond à la quantité NF)
- de plus nous considérons le nombre de noeuds qui n'ont qu'un fils, soit NT1.

Nous avons là encore défini une partition sur l'ensemble des noeuds : les noeuds qui ont deux fils, les noeuds qui ont un fils, les feuilles. On a donc : N = NT2 + NT1 + NF.
On peut dénombrer les fils de deux manières différentes. Ceci conduit à : nombre de fils + 1 = 2*NT2 + NT1 + 1 = N.

On aboutit ainsi aux relations :

$$NT2 = NF-1$$

$$NT1 = N-2*NF+1$$

D'autre part, on peut dénombrer les fils droits et gauches de deux manières :

$$FD = NT2+NT1D \quad \text{soit} \quad NT1D = FD-NF+1$$

$$FG = NT2+NT1G \quad \text{soit} \quad NT1G = FG-NF+1$$

Cas des arbres binaires "homogènes"

Un tel arbre est caractérisé par NT1 = 0

Par conséquent :

$N = 2*NF-1$ (N est forcément impair)

$FG = FD = NT2 = (N-1)/2$

$NF = (N+1)/2$

Cas des arbres binaires complets :

On considère ici les arbres binaires complets dont le dernier niveau est entièrement rempli : N est de la forme 2^p-1.

p est la profondeur de l'arbre, et on a :

$FG = FD = NT2 = 2^{p-1}-1$

$NF = 2^{p-1}$

6.2. - ETUDE DES FORMES ITERATIVES DES PARCOURS D'ARBRES

Nous construisons les algorithmes itératifs de parcours d'arbres de manière systématique en appliquant le traitement séquentiel et l'analyse récurrente de la manière suivante :

à chacun des ordres de parcours correspond la définition des trois primitives caractérisant un traitement séquentiel *PREMIER, SUCCESSEUR, DERNIER.*

Pour étudier ces primitives, nous appliquons systématiquement l'analyse récurrente en nous basant sur la définition récursive du parcours que nous avons donnée au § 5.2.1.

6.2.1. - Construction d'un algorithme itératif de parcours en préordre

Nous rappelons la forme récursive de ce parcours d'arbre :

```
action PRE (arbre A)
  VISITER(A)
  si E_G(A) alors PRE(G(A)) fsi
  si E_D(A) alors PRE(D(A)) fsi
faction PRE
```

L'arbre donné est parcouru par la séquence :

INIT_P ; PRE(RAC) ; TERM_P

a) Application du traitement séquentiel

Appliquant le traitement séquentiel, nous considérons la file des noeuds dans le préordre que nous caractérisons par les trois primitives suivantes :

- *PREMIER_PREORDRE* en abrégé *PRE_P*
- *SUCCESSEUR_PREORDRE* en abrégé *SUC_P*
- *DERNIER_PREORDRE* en abrégé *DER_P*

PRE_P et DER_P sont en fait étendues de la manière suivante : elles désignent respectivement le premier et le dernier noeud visité lors du parcours en préordre d'un sous-arbre quelconque de l'arbre considéré. On écrira donc PRE_P(A) et DER_P(A) si A est ce sous-arbre.
SUC_P désigne le successeur d'un noeud donné X lors du parcours en préordre de l'arbre donné. On écrira donc : SUC_P(X). Il est très important de souligner que SUC_P(X) n'est défini que si le successeur cherché existe effectivement, c'est-à-dire si X n'est pas le dernier noeud en préordre.
Dans ces conditions, un premier algorithme peut être écrit en appliquant un schéma de traitement séquentiel :

Algorithme : parcours en préordre d'un arbre donné non vide
{*NC* désigne le noeud courant dans ce parcours}

```
NC:=PRE_P(D)
itérer VISITER(NC)
arrêt : NC=DER_P(D)
        NC:=SUC_P(NC)
fitérer
```

La suite de la construction consiste à étudier chacune des primitives *PRE_P*, *SUC_P* et *DER_P*.

b) Description de la file des noeuds en préordre

Cette étude est conduite de la manière suivante : à chaque cas de l'analyse récurrente, on associe l'algorithme récursif du parcours en préordre simplifié en tenant compte de ce cas. On en déduit la définition des primitives dans chacun des cas :

Cas n° 1 :

exécuter *PRE(X)* revient à exécuter la séquence :
VISITER(X) ; PRE(G(X)) ; PRE(D(X))

On en déduit que dans ce cas :

PRE_P(X) = X
DER_P(X) = DER_P(D(X))
SUC_P(X) = PRE_P(G(X))

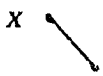
Cas n° 2 :

exécuter *PRE(X)* revient à exécuter la séquence :
VISITER(X) ; PRE(G(X))

On en déduit :

PRE_P(X) = X
DER_P(X) = DER_P(G(X))
SUC_P(X) = PRE_P(G(X))

Cas n° 3 :



exécuter *PRE(X)* revient à exécuter la séquence :
VISITER(X) ; PRE(D(X))

On en déduit :

PRE_P(X) = X
DER_P(X) = DER_P(D(X))
SUC_P(X) = PRE_P(D)(X))

Cas n° 4 :

exécuter *PRE(X)* revient à exécuter la séquence :
VISITER(X)

On en déduit

PRE_P(X) = X
DER_P(X) = X

mais on ne peut rien dire de *SUC_P(X)*.
Pour analyser *SUC_P(X)* dans ce cas, nous examinons dans quelles conditions *PRE(X)* a été appelé. La séquence d'appel du parcours en préordre de l'arbre donné et la forme récursive de l'action PRE, montrent que cet appel doit être l'un des trois cas suivants :

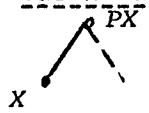
Cas n° 0 : appel initial : *PRE(RAC)*
Dans ce cas on a forcément $X = RAC$
Ce cas doit être exclu puisqu'il correspond à un arbre n'ayant qu'un seul noeud. Ce noeud est forcément le dernier dans le parcour en préordre et nous avons précisé que nous ne cherchons pas à définir *SUC_P* pour ce noeud.

Cas n° 1 : dans l'exécution d'un appel *PRE(PX)* , l'appel considéré *PRE(X)* correspond au parcours en préordre du sous-arbre gauche de *PX* on a $X = G(PX)$

Cas n° 2 : dans l'exécution d'un appel *PRE(PX)* , l'appel considéré *PRE(X)* correspond au parcours en préordre du sous-arbre droit de *PX*.
On a donc : $X = D(PX)$

Dans ces conditions peut-on répondre à la question : quel est le successeur en préordre de X dans le parcours de l'arbre donné ? Nous étudions alors les deux cas que nous venons de retenir :

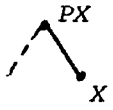
Cas n° 1 :



l'exécution de *PRE(PX)* revient à exécuter la séquence
VISITER(PX) ; VISITER(X) ; si E_D(PX) alors PRE(D(PX)) fsi

Clairement on ne peut répondre que si *PX* a un fils droit. Dans le cas contraire, on doit à nouveau considérer le contexte de l'appel *PRE(PX)* c'est-à-dire, considérer le père de *PX*

Cas n° 2 :



l'appel *PRE(PX)* revient à exécuter la séquence :
VISITER(PX) ; si E_G(PX) alors PRE(G(PX)) fsi ; VISITER(X)

Pour déterminer le successeur en préordre de X , on doit examiner le contexte de l'appel *PRE(PX)* , c'est-à-dire considérer le père de *PX*.

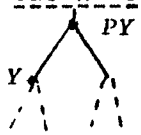
Pour aboutir dans cette analyse, nous cherchons à généraliser la question initiale, de manière à pouvoir ramener la question du successeur en préordre de *X* à cette généralisation appliquée au noeud PX. Nous cherchons donc a trouver une propriété qui caractérise à la fois X et PX à cet instant du parcours en préordre de l'arbre donné . La réponse que nous donnons est la suivante :

Dans les cas où notre recherche est sans réponse, nous ne pouvons conclure par l'examen du contexte d'appel considéré parce que, la dernière visite effectuée termine le parcours en préordre du sous-arbre. Nous cherchons donc à répondre à la question suivante :

Etant donné un sous-arbre *Y* dont on vient de terminer le parcours, lors du parcours en préordre de l'arbre donné, quel est le prochain noeud que l'on doit visiter. Nous désignons ce noeud par *REMONTER_PREORDRE(Y)* en abrégé *REM_P(Y)*.

Etude de *REM-P(Y)* : nous reprenons le principe d'analyse qui nous a guidé jusqu'à présent : nous examinons le contexte de l'appel *PRE(Y)* . Nous savons que le noeud cherché existe. Nous avons ainsi les 4 cas suivants issus de l'examen du père *PY* de *Y* :

Cas n° 1 :

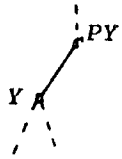


exécuter *PRE(PY)* revient à exécuter la séquence :
VISITER(PY) ; PRE(G(PY)) ; PRE(D(PY))

Nous savons que le dernier noeud visité est le dernier noeud en préordre de l'arbre *Y* . Par conséquent :

$$REM\_P(Y) = PRE\_P(D(PERE(Y))$$

Cas n° 2 :

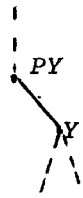


exécuter *PRE(PY)* revient, à exécuter la séquence :
VISITER(PY) ; PRE(G(PY))

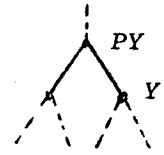
La fin du parcours de l'arbre *Y* entraîne la fin du parcours de l'arbre *PY*
Par conséquent :

$$REM\_P(Y) = REM\_P(PERE(Y))$$

Cas n° 3 :



et Cas n° 4 :



L'exécution de l'appel *PRE(PY)* revient à exécuter la séquence :
VISITER(PY) ; si E_G(PY) alors PRE(G(PY) fsi ; PRE(D(PY))

La fin du parcours de l'arbre *Y* entraîne la fin du parcours de l'arbre *PY*
Par conséquent :

$$REM\_P(Y) = REM\_P(PERE(Y))$$

L'analyse des trois fonctions qui caractérisent le parcours en préordre d'un arbre binaire est ainsi terminée. Nous résumons les résultats que nous avons obtenus dans le tableau suivant :

cas	*PRE_P(X)*	*DER_P(X)*	*SUC_P(X)*
E_G(X) et E_D(X)	*X*	*DER_P(D(X))*	*G(X)*
E_G(X) et ¬E_D(X)	*X*	*DER_P(G(X))*	*G(X)*
¬E_G(X) et E_D(X)	*X*	*DER_P(D(X))*	*D(X)*
¬E_G(X) et ¬E_D(X)	*X*	*X*	*REM_P(X)*

et *REM_P(Y)* est définie comme suit :

cas	*REM_P(Y)*
¬ESTDROIT(Y) et E_D(PERE(Y)	*D(PERE(Y))*
ESTDROIT(Y) ou ¬E_D(PERE(Y))	*REM_P(PERE(Y))*

A partir de cette analyse, nous pouvons facilement écrire d'une part les primitives de la machine séquentielle définissant le préordre sur les noeuds d'un arbre, d'autre part l'algorithme général de parcours en préordre.

Remarques :

- Le calcul du dernier noeud dans le préordre "coûte" une descente dans l'arbre qui est compensée par le fait que l'on n'a pas besoin de remonter ce chemin lorsque le dernier noeud a été visité.
- La version 1 est applicable directement dès que le nombre de noeud est connu.

Machine séquentielle de parcours en préordre d'un arbre binaire

le dernier élément de la file est reconnu lorsqu'on l'atteint
on utilise les primitives de la machine arbre - § 6.1.1. - qui consultent et/ou modifient le noeud courant NC .

état : - le noeud courant *NC* de la machine arbre
- le dernier noeud visité dans le parcours, soit *DP*

Primitives :

. initialisation du parcours :

```
action DEMARRER_PREORDRE
   {calcule DP et initialise NC au premier en préordre}
   DEMARRER_RACINE
   itérer tantque ¬FINAL_DROIT faire AVANCER_DROIT ftantque
   arrêt: FINAL_GAUCHE
          AVANCER_GAUCHE
   fitérer
   DP:=NC
   {on se place sur le premier en préordre}
   DEMARRER_RACINE
faction DEMARRER_PREORDRE
```

. progression dans le parcours :

```
action AVANCER_PREORDRE
   {calcule le successeur en préordre de NC. On sait qu'il existe}
   choix
        ¬FINAL_GAUCHE : AVANCER_GAUCHE
        FINAL_GAUCHE :  choix
                          ¬FINAL_DROIT : AVANCER_DROIT
                          FINAL_DROIT : répéter dedroite:=A_DROITE
                                        AVANCER_PERE
                                        jusqua ¬dedroite et ¬FINAL_DROIT
                                        AVANCER_DROIT
                        fchoix
   fchoix
faction AVANCER_PREORDRE
```

. détection du dernier élément

```
fonction FINAL_PREORDRE → logique :
   → NC = DP
ffonction FINAL_PREORDRE
```

Algorithme : parcours en préordre d'un arbre binaire - version 1

```
{on utilise les primitives de la machine arbre définies au § 6.1.1.}
{calcul du dernier en préordre }
DEMARRER_RACINE                                                    1
itérer tantque ¬FINAL_DROIT faire                                  lg+1+ld=L+1
         AVANCER_DROIT                                             ld
       ftantque
arrêt : FINAL_GAUCHE                                               lg+1
        AVANCER_GAUCHE                                             lg
fitérer
DP:=NC                                                             1
{parcours en préordre proprement dit}
DEMARRER_RACINE  {la racine est le premier en préordre}            1
itérer VISITER(NC)                                                 N
arrêt : NC=DP                                                      N
        si ¬FINAL_GAUCHE                                           N-1
            alors AVANCER_GAUCHE                                   FG
            sinon si ¬FINAL_DROIT                                  FD
                     alors AVANCER_DROIT                           NT1D=FD-NF+1
                     sinon {remontée}                              NF-1
                           répéter
                              dedroite:=A_DROITE                   N-L-1
                              AVANCER_PERE                         N-L-1
                           jusqua ¬dedroite et ¬FINAL_DROIT        N-L-1
                           AVANCER_DROIT                           NF-1
                  fsi
        fsi
fitérer
```

Evaluation : L est la longueur du chemin allant de la racine à DP (ld fils droits, lg fils gauches)

accès arborescent : *DEMARRER_RACINE*:2, *AVANCER_GAUCHE*:lg+FG,
AVANCER_DROIT:ld+FD
AVANCER_PERE:N-L-1, *FINAL_GAUCHE*:N+lg *FINAL_DROIT*:FD+N
A_DROITE:N-L-1 , *NC*:2N+1

opérations logiques : N-L-1
comparaisons : N
affectations : N-L
contrôle : 3N+FD+lg

6.2.2. - Sept variantes pour l'algorithme en préordre

Nous proposons dans les pages qui suivent plusieurs autres versions pour le parcours en préordre d'un arbre binaire. Ces versions diffèrent en fonction de l'approche qui guide l'analyse, des conditions particulières de réalisation, où d'hypothèses précises sur l'arbre considéré.

Nous commentons ici chacune de ces versions, les algorithmes correspondants étant regroupés à la fin du paragraphe.

a) Analyse rapide des différentes versions

Version 2 : On considère que l'arbre est une file de "branches gauches", i.e. de chemins dont le premier élément est soit la racine soit un fils droit et dont tous les autres éléments se déduisent par une descente à gauche. Tous les noeuds d'un tel chemin sont visités l'un à la suite de l'autre dans un parcours en préordre.
Ceci peut être exprimé récursivement de la manière suivante :

```
action PRE1 (arbre A)
{A est soit la racine de l'arbre parcouru, soit la racine d'un sous-arbre
                                                              droit}
  X:=A
  itérer VISITER(X)
  arrêt : ¬E_G(X)
          X:=G(X)
  fitérer
  itérer si E_D(X) alors PRE1(D(X)) fsi
  arrêt : X=A
          X:=PERE(X)
  fitérer
faction PRE1
```

Remarque :
Cet algorithme peut être obtenu directement à partir de la première forme récursive du préordre PRE, en appliquant une technique de suppression partielle de la récursivité (cf. notamment [Cha 77]).

La version 2 est déduite de cette action récursive de manière analogue à ce que nous avons fait dans l'analyse de la version 1 : on étudie les primitives suivantes : *PREMIERE_BRANCHE_GAUCHE* , *BRANCHE_GAUCHE_SUIVANTE*.
La fin de l'algorithme est contrôlée par la visite du dernier noeud en préordre calculé comme précédemment.

Remarque :

On peut aussi construire la version 2 de la manière suivante : on considère la file des noeuds de la " branche droite" issue de la racine : chacun de ces noeuds est racine d'un sous-arbre. On appliqu alors un schéma de parcours de file :

```
action PRE2 (arbre A)
 {A est soit la racine, soit un fils droit}
  X:=A
  itérer
       si E_G(X) alors PRE2(G(X)) fsi
  arrêt : ¬E_G(X)
          X:=G(X)
  fitérer
faction PRE2
```

En fait on vient de réaliser une structuration d'un arbre binaire, en arbre non binaire (opération inverse de ce qui a été présenté au § 5.1.5.) et on a appliqué à cet arbre un parcours préfixé. A partir de cette forme d'algorithme on obtient la version 2 en appliquant des techniques de transformation "récursif-itératif" (comme par exemple dans [Vei 74]).

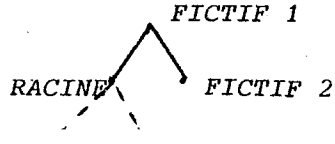
Version 3 : L'itération de "remontée" dans les versions 1 et 2 est issue de la même analyse : lorsque l'on vient de visiter une feuille, trouver le prochain noeud en préordre dans la version 1, la prochaine "branche gauche" dans la version 2 (ce qui est la même chose dans la mesure où une "branche gauche" est désignée par son premier noeud).

Dans la version 3, on modifie légèrement les conditions de cette analyse, en essayant de définir la "remontée" lorsque le dernier noeud en préordre a été atteint. En d'autres termes, cette version ne diffère de ce qui précède que par la manière de contrôler l'algorithme. On ne calcule plus au départ le dernier noeud en préordre. Par contre on introduit des cas supplémentaires dans l'analyse de la remontée (il ne faut pas dépasser la racine). La version 3 que nous proposons applique cette idée sur la base de la version 2.

Remarque :

En termes de "machines séquentielles", les versions 1 et 2 correspondent au modèle 2, la version 3 au modèle 1. cf. 2.1.2.

Version 3-bis : La version 3-bis supprime la nécessité du calcul du dernier noeud en préordre grâce à deux noeuds "fictifs" auxquels l'arbre traité est raccroché :



Dans le parcours en préordre de ce nouvel arbre, le dernier en préordre est FICTIF2. Cette méthode est applicable, si l'on peut définir ces deux noeuds fictifs :

on doit avoir : *PERE(RACINE) = FICTIF1*
GAUCHE(FICTIF1) = RACINE
DROIT(FICTIF1) = FICTIF2
ESTDROIT(RACINE)= faux

FICTIF1 est la racine du nouvel arbre.
On applique alors la version 1 ou la version 2 à cet arbre. Toutefois on ne veut pas visiter les noeuds fictifs : le schéma général est modifié en conséquence.
(nous suffixons les noms des primitives par une apostrophes pour indiquer qu'elles concernent le nouvel arbre).

DEMARRER_RACINE' ; AVANCER_GAUCHE'
répéter "traiter branche gauche"
"calculer prochaine branche gauche"
jusqua noeud courant=fictif2

Nous ne donnons pas l'algorithme correspondant. L'évaluation est pratiquement la même que celle obtenue pour la version 2 avec L=1.

Version 4 : Dans cette version, nous abordons les conditions de représentation de l'arbre donné. Nous supposons que les primitives *PERE* et *ESTDROIT* ne sont pas définies. Une première solution consiste à employer la technique présentée au § 6.1.4. et à appliquer l'une des versions précédentes sur le nouvel arbre ainsi défini. Ceci conduit à un premier type de solution faisant intervenir une pile.

La version 4 raffine l'analyse en tenant compte des particularités du parcours en préordre. Les primitives *PERE* et *ESTDROIT* n'interviennent que pour réaliser la primitive *REMONTER_PREORDRE*.

Nous essayons de définir un nouvel arbre à partir de l'arbre donné, tel que l'on puisse répondre à la question : "quel est le prochain noeud en préordre", lorsque le noeud est une feuille ?" (seul cas où l'analyse de *SUCCESSEUR_PREORDRE* a motivé la définition de *REMONTER_PREORDRE*.
Pour cela on associe à chaque noeud la liste des sous-arbres droits qui ont été laissés de côté pour arriver à ce noeud. Cette liste est gérée en pile et l'on aboutit finalement à définir le nouvel arbre par la machine suivante :

D_RAC' : vider_pile ; D_RAC
AV_G' : si ¬FIN_D alors empiler (NC) fsi ; AV_G

Les autres primitives sont inchangées.
Dans ces conditions, la séquence de remontée s'écrit :

{on vient de visiter une feuille de l'arbre binaire}
NC:=dépiler

Dans la version 4, on considère que la pile et les primitives qui la manipulent sont attachées à la machine arbre.

Remarque :
On peut appliquer simplement le principe expliqué pour la version 3-bis : il suffit d'empiler au départ l'élément FICTIF.

Versions 5, 6 et 7 : Nous étudions le cas des arbres binaires homogènes : le fait que tout noeud non terminal ait deux fils simplifie les algorithmes.

b) Algorithmes

Algorithme : parcours en préordre d'un arbre binaire - version 2

{Sur la base de la suite des "branches gauches". *DP* est le dernier noeud en préordre : L est la longueur du chemin conduisant de la racine à *DP* Ce chemin comporte lg branches gauches et ld branches droites : L=lg+ld. On utilise les primitives de la machine arbre § 6.1.1.}

```
{calcul du dernier en préordre}                                          1
DEMARRER_RACINE                                                          1
itérer tantque ¬FINAL_DROIT faire                                        lg+1+ld=L+1
        AVANCER_DROIT                                                    ld
       ftantque
arrêt : FINAL_GAUCHE                                                     lg+1
        AVANCER_GAUCHE                                                   lg
fitérer
DP := NC                                                                 1
{parcours en préordre proprement dit}
DEMARRER_RACINE  {la racine est la tête de la première                   1
                  branche gauche}
itérer
        itérer VISITER(NC)     {visites des noeuds de la                 N
        arrêt : FINAL_GAUCHE    branche gauche}                          N
                AVANCER_GAUCHE                                           GF=N-(FD+1)
        fitérer
arrêt : NC = DP                                                          FD+1
        si -FINAL_DROIT                                                  FD
           alors AVANCER_DROIT                                           NT1D=FD-NF+1
           sinon répéter
                   dedroite:=A_DROITE ; AVANCER_PERE                     N-L-1
                 jusqua ¬dedroite et ¬FINAL_DROIT                        N-L-1
                 AVANCER_DROIT                                           NF-1
        fsi
fitérer
```

Evaluation :

accès arborescent : *FINAL_GAUCHE*:N+lg+1, *NC*:N+FD+2
pour les autres primitives comme version 1

opérations logiques : N-L-1
comparaisons : FD+1
affectations : N-L
contrôle : 2N+2FD+lg+2

Algorithme : parcours en préordre d'un arbre binaire - version 3

```
{la fin du parcours est détectée lorsque la remontée conduit à la racine}
{sur la base de la version 2}
fini:=faux                                                          1
DEMARRER_RACINE                                                     1
itérer
        itérer VISITER(NC)                                          N
        arrêt : FINAL_GAUCHE                                        N
                AVANCER_GAUCHE                                      FG=N-(FD+1)
        fitérer
        {recherche de la prochaine branche gauche}
        si FINAL_DROIT alors                                        FD+1
           {on est sur une feuille : remontée}
           répéter si EST_RACINE                                    N
                     alors fini := vrai                             1
                     sinon dedroite:=A_DROITE ; AVANCER_PERE        N-1
           jusqua fini ou (¬dedroite et ¬FINAL_DROIT)               N
        fsi
arrêt : fini                                                        FD+1
        AVANCER_DROIT                                               FD
fitérer
```

Evaluation :
accès arborescent : *DEMARRER_RACINE*:1, *AVANCER_GAUCHE*: FG, *AVANCER_DROIT*:FD
AVANCER_PERE:N-1, *FINAL_GAUCHE*:N, *FINAL_DROIT*:N+FD+1
A_DROITE:N-1, *NC*:N, *EST_RACINE*:N
opérations logiques : 2N
affectations:N+1
contrôle:3N+2FD+2

Algorithme : parcours en préordre d'un arbre binaire : version 4

{Les primitives *PERE* et *ESTDROIT* ne sont pas définies. La machine arbre comporte des primitives de manipulation de pile : *VIDER_PILE* initialise la pile par un *FICTIF*, *EMPILER* empile le noeud courant, *DEPILER* met à jour le noeud courant avec le sommet de la pile et la décapite. L'initialisation de la pile par un *FICTIF* permet d'appliquer le principe de la version 3_bis, i.e. d'appliquer la version 2 sans avoir à calculer le dernier en préordre}.

DEMARRER_RACINE ; VIDER_PILE	1
répéter	
itérer VISITER(NC)	N
arrêt : FINAL_GAUCHE	N
si ¬FINAL_DROIT alors	FG
EMPILER	NT2=NF-1
fsi	
AVANCER_GAUCHE	FG
fitérer	
si ¬FINAL_DROIT	FD+1
alors AVANCER_DROIT	NT1D=FD-NF+1
sinon DEPILER	NF
fsi	
jusqua NC=fictif	FD+1

Evaluation :

accès arborescent : *DEMARRER_RACINE*:1, *AVANCER_GAUCHE*:FG, *AVANCER_DROIT*:FD
FINAL_GAUCHE:N, *FINAL_DROIT*:N, *NC*:N+FD+1, *FICTIF*:FD+1
EMPILER:NF-1, *DEPILER*:NF, *VIDER_PILE*:1
{la taille qu'il faut prévoir pour la pile est bornée par NF≤(N+1)/2}.

comparaisons : FD+1
contrôle : 2N+FD+2

Algorithme : parcours en préordre d'un arbre binaire - version 5

```
{l'arbre est homogène - on applique la version 2}
DEMARRER_RACINE                                          1
tantque ¬FINAL_DROIT faire                               L+1
  AVANCER_DROIT                                          L
ftantque
DP:=NC                                                   1
DEMARRER_RACINE                                          1
itérer itérer VISITER(NC)                                N
       arrêt : FINAL_GAUCHE                              N
               AVANCER_GAUCHE                            FG=(N-1)/2
       fitérer
arrêt : NC=DP                                            FD+1=(N+1)/2
       tantque A_DROITE faire                            2FD-L
         AVANCER_PERE                                    FD-L
       ftantque
       AVANCER_PERE ; AVANCER_DROIT                      FD
fitérer
```

Evaluation :

accès arborescent : *FINAL_GAUCHE*:N, *FINAL_DROIT*:L+1, pour les autres mêmes chiffres que pour la version 2, sachant que lg=0

comparaisons : (N+1)/2

contrôle:(5N+1)/2

algorithme : parcours en préordre d'un arbre binaire - version 6

{on applique la version 3 au cas des arbres binaires homogènes}

```
DEMARRER_RACINE                                              1
itérer
        itérer VISITER(NC)                                   N
        arrêt : FINAL_GAUCHE                                 N
                AVANCER_GAUCHE                               FG=(N-1)/2
        fitérer
{(1)}   tantque ¬EST_RACINE et_c A_DROITE faire              2FD+1=N
                AVANCER_PERE                                 FD=(N-1)/2
        ftantque
arrêt : EST_RACINE.                                          FD+1=(N+1)/2
        AVANCER_PERE ; AVANCER_DROIT                         FD=(N-1)/2
fitérer
```

Evaluation :
accès arborescent : *FINAL_DROIT*:0, *A_DROITE*:N-1, *EST_RACINE*:(3N+1)/2
pour les autres primitives voir version 3.
opérations logiques : N
contrôle : (5N+1)/2

remarque : (version 6-vis)
Si l'on peut définir la fonction *ESTDROIT* de telle sorte que *ESTDROIT(RACINE)=faux* (i.e. tout se passe comme si l'arbre est accorché en fils gauche d'un noeud fictif), le contrôle de la remontée dans l'arbre (marqué (1) dans l'algorithme ci-dessus) peut être écrit plus simplement (application d'une technique de sentinelle) : il s'écrit alors : *tantque A_DROITE faire...*
L'évaluation est alors changée de la manière suivante :
- primitives : *EST_RACINE*:(N+1)/2
- opérations logiques:0

algorithme : parcours en préordre d'un arbre binaire - version 7

{On applique la version 4 au cas d'un arbre binaire homogène.
EMPILER, DEPILER sont des primitives de gestion d'une pile qui respectivemen
empile le noeud courant et décapite la pile après avoir mis à jour le noeud
courant avec le sommet de pile. *VIDER_PILE* initialise la pile avec un *FICTIF*

```
VIDER_PILE ; DEMARRER_RACINE                                 1
itérer
        itérer VISITER(NC)                                   N
        arrêt : FINAL_GAUCHE                                 N
                EMPILER ; AVANCER_GAUCHE                     FG=(N-1)/2
        fitérer
        DEPILER                                              FD+1=(N+1)/2
arrêt : NC=FICTIF                                            FD+1=(N+1)/2
        AVANCER_DROIT                                        FD=(N-1)/2
fitérer
```

Evaluation :
accès arborescent : *FINAL_DROIT*:0, pour les autres primitives voir version 4
comparaisons : (N+1)/2
contrôle : (3N+1)/2

c) Comparaison des solutions obtenues

Les différentes versions que nous venons de présenter diffèrent d'une part du point de vue de leurs conditions d'applications, d'autre part du point de vue de leurs performances. Nous n'insistons pas davantage sur les conditions d'application, et nous cherchons à comparer les solutions d'un point de vue quantitatif. Pour cela, nous nous plaçons dans le contexte particulier suivant :

- l'arbre considéré est représenté en mémoire sour forme d'un ensemble de noeuds chaînés entre eux à l'aide de pointeurs. Sauf lorsque cela est précisé autrement, l'ensemble des informations contenues dans un tel noeud suffit à donner directement la valeur des primitives attachées à ce noeud. Ainsi par exemple, chaque noeud porte une information logique indiquant si le noeud est fils gauche ou fils droit.

- nous nous plaçons dans un contexte de programmation tel que l'on puisse mesurer les coûts des algorithmes en termes d'une unité de temps dont la valeur exacte n'est pas précisée (nous reprenons ainsi la technique donnée dans [Knu 69]).

Dans ces conditions, notre évaluation comporte deux étapes :

1. Le calcul du coût de l'implémentation particulière de l'arbre ("coût arbre") : nous cumulons les coûts correspondant à l'utilisation des primitives de la machine arbre, sur la base des hypothèses suivantes :

   . le noeud courant est maintenu dans un registre
   . *D_RAC, AV_G, AV_D, AV_P* : chacune de ces primitives provoque un chargement de *NC* soit un coût de 2 unités.
   . dans le cas où une pile est utilisée : le sommet est désigné à l'aide d'un registre.
   . *VIDER_PILE* : initialisation du registre, de la pile par *FICTIF* , soit un coût de 3 unités.
   . *EMPILER, DEPILER* : incrémentation ou décrémentation du registre pile, chargement ou déchargement de la pile, soit un coût de 3 unités.

opérations/version	coefficient	arbre brinaires quelconques				arbres binaires homogènes			
		1	2	3	4	5	6	6 bis	7
D_RAC+AV_G+AV_D+AV P	2	2N	2N	2N	N	2N	2N	2N	N
VIDER+EMPILER+DEPILER	3	0	0	0	2NF	0	0	0	N+1
FIN_G+FIN_D+EST_RAC	2	2N+FD+1g	2N+FD+1g+1	3N+FD+1	2N	N+L+1	(5N+1)/2	(3N+1)/2	N
comparaison	2	N	FD+1	0	FD+1	(N+1)/2	0	0	(N+1)/2
et/ou	1	N-L-1	N-L-1	2N	0	0	N	0	0
contrôle	1	3N+FD+1g	2N+2FD+1g+2	3N+2FD+2	2N+FD+2	(5N+1)/2	(5N+1)/2	(5N+1)/2	(3N+1)/2
affectations	2	N-L	N-L	N+1	0	0	0	0	0
coût arbre		8N+2FD+21g	8N+2FD+21g+2	10N+2FD+2	6N+6NF	6N+2L+2	9N+1	7N+1	7N+3
coût algorithme		8N+FD+2L-1d-1	5N+4FD-3L+1g+3	7N+2FD+4	2N+3FD+4	(7N+3)/2	(7N+1)/2	(5N+1)/2	(5N+3)/2
coût total		16N+3FD-21d-1	13N+6FD-31d+5	17N+4FD+6	8N+3FD+6NF+4	(19N+3)/2+2L+2	(25N+3)/2	(19N+3)/2	(19N+9)/2
coût supplémentaire lorsque ESTDROIT n'est pas définie		6N	6N	0	0	6N	6N	6N	0

Evaluation des 8 versions de parcours en préordre d'un arbre binaire dans le cas d'une représentation "chaînée" de l'arbre.

. *FIN_G, FIN_D, EST_RACINE* : mise à jour d'un indicateur de comparaison, soit 2 unités.

. tout accés à *NC*, *FICTIF*, et utilisation de *A_DROITE* est comptabilisée dans l'instruction les utilisant.

. lorsque la fonction *ESTDROIT* n'est pas définie, nous utilisons la techniqu proposée au § 6.1.2. : ceci revient à dire que le coût des primitives *D_RAC, AV_G, AV_D* et *AV_P* est augmenté de 3 unités.

2. Le calcul du coût lié à la forme de l'algorithme choisi ("coût algorithme") Nous supposons que chaque fois qu'il y a une variable logique elle est implantée dans un registre. Dans ces conditions, nous nous basons sur les chiffres suivants : comparaison : 2 unités, contrôle : 1 unité, opération logique et/ou : 1 unité, affectation : 2 unités.
L'ensemble de ces évaluations est résumé dans le tableau de la page suivante

Pour comparer les coûts obtenus dans ce tableau, nous nous plaçons dans le cas moyen où FG = FD = NF = N/2 et où N est suffisamment grand pour que l'on puisse ignorer les constantes. On suppose de plus que L est négligeable par rapport à N. Dans ces conditions, on obtient les chiffres approximatifs suivants (coût 1 : fonction *ESTDROIT* définie, coût 2 : fonction *ESTDROIT* non définie).

version	1	2	3	4	5	6	6 bis	7
coût 1	35N/2	32N/2	38N/2	25N/2	19N/2	25N/2	19N/2	19N/2
coût 2	47N/2	44N/2	50N/2	25N/2	31N/2	37N/1	31N/2	19N/2

On peut faire les remarques suivantes :

. l'importance relative du coût supplémentaire (6N) du à la réalisation de *ESTDROIT*

. les performances des solutions ayant recours à une pile (versions 4 et 7) qui notamment ne dépendent pas de *ESTDROIT*.

Remarque :

Ceci se comprend intuitivement de la manière suivante : le recours à la pile permet d'éviter un certain nombre de remontées élémentaires (retour au père), dans la mesure où on remonte directement au prochain noeud à visiter.

. en ce qui concerne le coût 1 :

- les différences entre les versions qui traitent les arbres binaires homogènes (5 à 7) sont moins accentuées que les différences entre les versions qui traitent le cas général (1 à 4).

. à titre indicatif, on a (C_i désigne le coût de la version i)
C_6-C_{6bis} / C_6 = 24 % , C_3-C_4/C_3 = 34 %, C_3-C_2/C_3 = 16 %, C_1-C_2/C_1 = 8 %.

Mais il faut surtout souligner que nous n'avons pas tenu compte pour l'instant du coût du traitement de l'arbre proprement dit : chacun des algorithmes applique N fois l'action VISITER. Les différences que nous venons d'établir ne restent significatives, que dans la mesure où le coût de VISITER n'est pas trop important. A titre indicatif, on peut voir l'évolution du gain entre les versions 3 et 6 bis dans le cas du coût 1 et les versions 3 et 7 dans le cas du coût 2.

	coût de VISITER	2	4	8	12	20	30	50	75	100	1
(coût1)	gain C_3-C_{6bis}/C_3	45%	41%	35%	30,6%	24,35%	19,5%	13,8%	10,1%	8%	5
(coût2)	gain C_3-C_7/C_7	57,4%	53,4%	47%	42%	34,4%	28,2%	13,6%	15,5%	12,4%	8

d) Conclusion

Cette étude a montré l'intérêt d'une démarche systématique. D'une part nous maîtrisons mieux le problème général traité en proposant diverses solutions aux conditions d'application variées, d'autre part nous pouvons, une fois fixées les conditions particulières d'implantation des algorithmes, comparer les solutions de manière quantitative et notamment décider si les différences sont significatives. Soulignons par ailleurs l'importance du rôle que joue le choix d'un formalisme : la machine arbre a permis de séparer les aspects relatifs à l'implantation des données de ceux relatifs à la formulation d'un algorithme.

6.2.3. - Etude du parcours en ordre symétrique

Nous partons de la forme récursive du parcours en ordre symétrique :

```
action SYM (arbre A)
  si E_G(A) alors SYM(G(A)) fsi
  VISITER(A)
  si E_D(A) alors SYM(D(A)) fsi
faction SYM
```

L'arbre donné est parcouru en ordre symétrique par l'appel :

INIT_P ; SYM(RAC) ; TERM_P

a) Construction de l'algorithme

Nous suivons la même démarche que pour le préordre. Appliquant le traitement séquentiel, nous considérons la file des noeuds en ordre symétrique et étudions les trois primitives *PREMIER_SYMETRIQUE* (en abrégé *PRE_S*), *SUCCESSEUR_SYMETRIQUE* (en abrégé *SUC_S*) et *DERNIER_SYMETRIQUE* (en abrégé *DER_S*). Comme précédemment, *PRE_S* et *DER_S* sont étendues de la manière suivante : elles désignent respectivement le premier et le dernier noeud visité lors du parcours en ordre symétrique d'un sous-arbre quelconque dans l'arbre considéré. A étant ce sous-arbre, on écrira *PRE_S(A)* et *DER_S(A)*. *SUC_S(X)* désigne le successeur d'un noeud donné *X* lors du parcours en ordre symétrique de l'arbre donné. Nous étudions *SUC_P(X)* dans l'hypothèse où ce successeur existe (i.e. *X≠DER_S(D)*).

L'analyse de ces trois primitives est faite selon les mêmes principes que pour le préordre (§ 6.2.1.) : comme précédemment, l'étude de *SUC_S(X)*, conduit à définir une primitive *REMONTER_SYMETRIQUE* (en abrégé *REM_S*) définie comme suit :

REM_S(Y) désigne le prochain noeud qui doit être visité lors du parcours en ordre symétrique de l'arbre donné, sachant que le dernier noeud qui a été visité a terminé le parcours en ordre symétrique du sous-arbre de racine *Y*.

Nous nous bornons ici à donner le tableau récapitulatif de cette analyse :

cas	*PRE_S(X)*	*DER_S(X)*	*SUC_S(X)*
E_G(X) et *E_D(X)*	*PRE_S(G(X))*	*DER_S(D(X))*	*PRE_S(D(X))*
E_G(X) et *¬E_D(X)*	*PRE_S(G(X))*	*X*	*REM_S(X)*
¬E_G(X) et *E_D(X)*	*X*	*DER_S(D(X))*	*PRE_S(D(X))*
¬E_G(X) et *¬E_D(X)*	*X*	*X*	*REM_S(X)*

et la fonction *REM_S* est définie comme suit :

cas	*REM_S(Y)*
ESTDROIT(Y)	*REM_S(PERE(Y))*
¬*ESTDROIT(Y)*	*PERE(Y)*

A partir de cette analyse on aboutit à la version 1 du parcours en ordre symétrique d'un arbre binaire. Comme précédemment, nous étudions d'autres versions de cet algorithme.

b) Autres versions

. Pour éviter le calcul du dernier noeud visité lors du parcours en ordre symétrique, on peut définir une version 1 bis suivant le principe suivant : à partir de l'arbre donné, on considère un nouvel arbre dont la racine est un noeud *FICTIF* et qui a pour sous-arbre l'arbre donné :

Dans ces conditions, l'ordre symétrique du nouvel arbre se termine par la visite de *FICTIF* . Comme on ne veut pas "visiter" ce noeud, la version 1 est adaptée en conséquence : le calcul du dernier en ordre symétrique est évité (c'est *FICTIF*), et le parcours proprement dit est décrit à l'aide d'un schéma *tantque*(ce qui permet de ne pas visiter le dernier élément).

Nous ne donnons pas cet algorithme qui se déduit directement de la version 1.

. La version 2 est basée sur la même analyse que la version 1, mais on évite de calculer le dernieur noeud en ordre symétrique : l'analyse de *REMONTER_SYMETRIQUE* fait intervenir un nouveau cas qui contrôle la fin de l'algorithme.

. Une version 3 que nous ne donnons pas, consiste à regrouper les descentes à gauche dans l'algorithme 2 (selon un principe analogue à celui de la version 2 du.préordre).

.La version 4 correspond au cas où les fonctions *PERE* et *ESTDROIT* ne sont pas définies. On peut appliquer la technique proposée au § 6.1.3. La version 4 que nous proposons est basée sur le principe suivant : on associe à chaque noeud de l'arbre l'ensemble des noeuds dont il est descendant par un arc gauche. Cet ensemble permet de réaliser la fonction *REMONTER_SYMETRIQUE*. Comme pour la version 4 du préordre, on utilise des primitives de gestion de pile attachées à la machine arbre. D'autre part le contrôle de l'algorithme est réalisé en suivant le principe de la version 1-bis : on suppose que l'arbre est fils gauche d'un noeud fictif qui est mis dans la pile lors de l'initialisation. Enfin, cette version 4 est basée sur le principe de la version 3 (traitement d'une suite de "branchements gauches").

. Les versions 5 et 6 concernent le cas des arbres binaires homogènes.

c) Algorithmes

algorithme : parcours en ordre symétrique d'un arbre binaire - version 1

```
{calcul du dernier en ordre symétrique}
DEMARRER_RACINE                                          1
tantque ¬FINAL_DROIT faire                               ld+1
  AVANCER_DROIT                                          ld
ftantque
DS:=NC ; DEMARRER_RACINE                                 1
{parcours proprement dit}
{calcul du premier en ordre symétrique}
tantque ¬FINAL_GAUCHE faire                              lg+1
  AVANCER_GAUCHE                                         lg
ftantque
itérer VISITER(NC)                                       N
arrêt : NC=DS                                            N
        {passage au successeur symétrique}
        si ¬FINAL_DROIT                                  N-1
           alors AVANCER_DROIT                           FD
                 tantque ¬FINAL_GAUCHE faire             N-lg+1
                   AVANCER_GAUCHE                        FG-lg
                 ftantque
           sinon  {remontée}
                 tantque A_DROITE faire                  N-ld-1
                   AVANCER_PERE                          FD-ld
                 ftantque
                 AVANCER_PERE                            FG
        fsi
fitérer
```

Evaluation : (lg nombre d'arcs gauche entre la racine et le premier visité
ld nombre d'arcs droit entre la racine et le dernier visité)

accès arborescent : *DEMARRER_RACINE*:2, *AVANCER_GAUCHE*:FG, *AVANCER_DROIT*:FD+ld
AVANCER_PERE:N-ld-1, *FINAL_GAUCHE*:N, *FINAL_DROIT*:N+ld
A_DROITE:N-ld-1, *NC*:2N+1

comparaisons : N
contrôle : 4N-1

Algorithme : parcours en ordre symétrique d'un arbre binaire - version2

```
{le dernier en ordre symétrique n'est pas calculé. La fin de l'algorithme
 est déterminée par une remontée sur la racine}
fini:=faux                                                    1
{premier élément dans l'ordre symétrique}
DEMARRER_RACINE                                               1
tantque ¬FINAL_GAUCHE faire                                   lg+1
  AVANCER_GAUCHE                                              lg
ftantque
répéter
  VISITER(NC)                                                 N
  si ¬FINAL_DROIT                                             N
     alors AVANCER_DROIT'                                     FD
           tantque ¬FINAL_GAUCHE faire                        N-lg-1
             AVANCER_GAUCHE                                   FG-lg
           ftantque
     sinon {remontée}
{(1)}      tantque ¬EST_RACINE et_c A_DROITE faire            N
             AVANCER_PERE                                     FD
           ftantque
           si EST_RACINE                                      FG+1
              alors fini:=vrai                                1
              sinon AVANCER_PERE                              FG
           fsi
  fsi
jusqua fini                                                   N
```

Evaluation :

accès arborescent :*DEMARRER_RACINE*:1, *AVANCER_GAUCHE*:FG, *AVANCER_DROIT*:FD
AVANCER_PERE:N-1, *FINAL_GAUCHE*:N, *FINAL_DROIT*:N
EST_RACINE:N+FG+1 , *A_DROITE*:N, *NC*:N

opérations logiques : N

contrôle : 4N+FG+1

Remarque : version 2bis

Si l'on peut définir la fonction *ESTDROIT*pour la racine de telle sorte que *ESTDROIT(RACINE)=faux* l'itération de remontée marquée (1) dans la version 2 ci-dessus peut s'écrire *tantque A_DROITE faire...*
L'évaluation est alors changée en conséquence :
primitives : *EST_RACINE*:FG+1
opérations logiques : 0

algorithme : parcours en ordre symétrique d'un arbre binaire - version 4

{les fonctions *PERE* et *ESTDROIT* ne sont pas définies. A chaque noeud est associé l'ensemble des noeuds dont il descend par un arc gauche. Cet ensemble est géré en pile : *EMPILER* empile le noeud courant, *DEPILER* met à jour le noeud courant avec le sommet de pile et décapite la pile. La pile est initialisée par un *FICTIF* à l'aide de *VIDER_PILE* . On a : *E_D(FICTIF)=faux*. On regroupe ces "descentes à gauche"}.

DEMARRER_RACINE ; VIDER_PILE	1
itérer tantque ¬FINAL_GAUCHE faire	N
EMPILER_ ; AVANCER_GAUCHE	FG
ftantque	
{visite de tous les noeuds sans fils droit}	FD+1
tantque FINAL_DROIT faire	N+1
VISITER(NC) ; DEPILER	NT1G+NF=FG+1
ftantque	
arrêt : NC=FICTIF	FD+1
VISITER(NC) ; AVANCER_DROIT	FD
fitérer	

Evaluation :

accès arborescent: *DEMARRER_RACINE*:1, *AVANCER_GAUCHE*:FG, *AVANCER_DROIT*:FD
VIDER_PILE:1, *EMPILER*:FG, *DEPILER*:FG+1
FINAL_GAUCHE:N, *FINAL_DROIT*:N+1, *NC*:N, *FICTIF*:FD+1

comparaison : FD+1

contrôle : 2N+FD+2

Algorithme : parcours en ordre symétrique d'un arbre binaire. Version 5

{on considère le cas des arbres binaires homogènes. On applique la version 3 : regroupement des descentes à gauche". On suppose que la fonction *ESTDROIT* peut être définie pour la racine : *ESTDROIT(RACINE)=faux*}

```
DEMARRER_RACINE                                          1
itérer tantque ¬FINAL_GAUCHE faire                       N
         AVANCER_GAUCHE                                  FG=(N-1)/2
       ftantque
       VISITER(NC)                                       FD+1=NF=(N+1)/2
       tantque A_DROITE faire                            N
         AVANCER_PERE                                    FD=(N-1)/2
       ftantque
arrêt : EST_RACINE                                       FD+1=(N+1)/2
       AVANCER_PERE ; VISITER(NC) ; AVANCER_DROIT        FD=(N-1)/2
fitérer
```

Evaluation :
accès arborescent: *DEMARRER_RACINE*:1, *AVANCER_GAUCHE*:(N-1)/2 ,
AVANCER_DROIT:(N-1)/2
AVANCER_PERE:N-1, *FINAL_GAUCHE*:N, *FINAL_DROIT*:0
EST_RACINE:(N+1)/2
A_DROITE:N, *NC*:N
contrôle : (5N+1)/2

Remarque : version 5-bis
Une forme analogue d'algorithme est obtenue si l'on peut définir un fictif comme père de la racine. Ce fictif n'est pas visité, mais sert uniquement à contrôler la fin de l'algorithme.

Algorithme : parcours en ordre symétrique d'un arbre binaire - version 6

{cas où la fonction *PERE* et la fonction *ESTDROIT* ne sont pas définies. on applique la version 4 au cas des arbres binaires homogènes}

```
DEMARRER_RACINE ; VIDER_PILE                             1
itérer : tantque ¬FINAL_GAUCHE faire                     N
          EMPILER ; AVANCER_GAUCHE                       (N-1)/2
         ftantque
         VISITER(NC) ; DEPILER                           (N+1)/2
arrêt : NC=FICTIF                                        (N+1)/2
        VISITER(NC) ; AVANCER_DROIT                      (N-1)/2
fitérer
```

Evaluation :
accès arborescent : *DEMARRER_RACINE*:1, *AVANCER_GAUCHE*:(N-1)/2,,
AVANCER_DROIT:(N-1)/2
VIDER_PILE:1, *EMPILER*:(N-1)/2, *DEPILER*:(N+1)/2
FINAL_GAUCHE:N, *FINAL_DROIT*:0
comparaison : (N+1)/2
contrôle : (3N+1)/2

d) Comparaison des solutions proposées

Nous faisons cette comparaison dans les mêmes hypothèses que pour le préordre. On obtient les chiffres suivants (on suppose que *ESTDROIT* est définie. Lorsqu'elle ne l'est pas, il faut ajouter 6N aux versions 1, 2, 2 bis et 5).

version	1	2	2 bis	4	5	6
coût	14N+21d-1	15N+3FG+3	12N+3FG+3	11N+3FG+9	6N	9N+4

soit en supposant FD = FG = N/2, les coûts approximatifs suivants :

28N/2	33N/2	27N/2	25N/2	12N/2	18N

On peut donc constater :
. les faibles variations entre les solutions dans le cas général (versions 1 à 4) (ceci est du à la "symétrie" du parcours)
. la bonne performance de la version 5 par rapport à ce qu'on avait pour le préordre.

6.2.4. - Etude du parcours en ordre terminal

Nous partons de la forme récursive du parcours en ordre terminal :

```
action TER(arbre A)
  si E_G(A) alors TER(G(A)) fsi
  si E_D(A) alors TER(D(A)) fsi
  VISITER(A)
faction TER
```

L'arbre donné est parcouru en ordre terminal par la séquence :

INIT_P ; TER(RAC) ; TERM_P

a) Analyse

Nous construisons un algorithme itératif en suivant la même méthode que pour les autres ordres de parcours. Appliquant le traitement séquentiel, nous considérons la file des noeuds en ordre terminal et étudions les trois primitives *PREMIER_TERMINAL*, (en abrégé *PRE_T*), *SUCCESSEUR_TERMINAL* (en abrégé *SUC_T*) et *DERNIER_TERMINAL* (en abrégé *DER_T*). *PRE_T* et *DER_T* sont définies pour tout sous-arbre de l'arbre donné. *SUC_T(X)* désigne le successeur en ordre terminal du noeud *X* lors du parcours de l'arbre donné. On suppose que *X* n'est pas le dernier en ordre terminal. Comme précédemment, l'étude

de *SUC_T* fait apparaître une primitive *REMONTER_TERMINAL* (en abrégé *REM_T*) définie comme suit :
REM_T Y désigne le prochain noeud qui doit être visité lors du parcours en ordre terminal de l'arbre donné, lorsque le dernier noeud qui a été visité a terminé le parcours en ordre terminal du sous-arbre de racine Y.
Nous nous bornons à donner ici le tableau récapitulatif de cette analyse :

cas	*PRE_T(X)*	*DER_T(X)*	*SUC_T(X)*
E_G(X) et E_D(X)	*PRE_T(G(X))*	*X*	*REM_T(X)*
E_G(X) et ¬E_D(X)	*PRE_T(G(X))*	*X*	*REM_T(X)*
¬E_G(X) et E_D(X)	*PRE_T(D(X))*	*X*	*REM_T(X)*
¬E_G(X) et ¬E_D(X)	*X*	*X*	*REM_T(X)*

et *REM_T* est définie comme suit :

cas	*REM_T(Y)*
¬ESTDROIT(Y) et E_D(PERE(Y))	*PRE_T(D(PERE(Y)))*
¬ESTDROIT(Y) et ¬E_D(PERE(Y))	*PERE(Y)*
ESTDROIT(Y) et E_G(PERE(Y))	*PERE(Y)*
ESTDROIT(Y) et ¬E_G(PERE(Y))	*PERE(Y)*

On déduit directement un algorithme de cette analyse (version 1).

b) Variantes

Comme précédemment, on peut construire plusieurs autres solutions :
. la version 1 bis (qui n'est pas donnée ici) s'attache à regrouper en un seul endroit dans l'algorithme les instructions correspondants à la recherche du premier noeud visité en ordre terminal. L'algorithme se déduit directement de la version 1 de la manière suivante. On considère un nouvel arbre constitué d'une racine FP ayant pour sous-arbre gauche un noeud *FF*, et pour sous-arbre droit, l'arbre donné.
La version 1 est alors appliquée à cet arbre, en tenant compte que *FF* en est le premier en ordre terminal et qu'il n'est pas visité. De plus on ne visite pas le dernier noeud *FP*.

Pour que cette réalisation soit possible, il faut pouvoir étendre les primitives définissant l'arbre donné de la manière suivante :

PERE(RACINE)=FP, ESTDROIT(RACINE)=vrai, GAUCHE(FP)=FF,
DROIT(FP)=RACINE , E_G(FF)=E_D(FF)=faux, PERE(FF)=FP, ESTDROIT(FF)=faux

. La version 2 (que nous ne donnons pas) traite le cas où la fonction *PERE* n'est pas définie. Contrairement à ce qui se passait pour le préordre et l'ordre symétrique, on doit appliquer ici la technique proposée au § 6.1.3., c'est-à-dire, appliquer la version 1 à un arbre défini à partir de l'arbre donné en associant à chaque noeud le chemin qui y conduit en partant de la racine. On peut remarquer que l'on a toujours besoin de la fonction ESTDROIT (contrairement aux versions analogues pour le préordre et l'ordre symétrique).

. Les versions 3 et 4 concernent le cas des arbres binaires homogènes. Nous ne donnons que la version 3 dérivée de la version 1, la version 4 en étant une application lorsque l'on recourt à la technique du § 6.1.3. pour réaliser les fonctions *PERE* et *ESTDROIT*.

c) Algorithmes

algorithme : parcours en ordre terminal d'un arbre binaire - version 1

```
DEMARRER_RACINE                                          1
{calcul du premier en ordre terminal}
itérer tantque ¬FINAL_GAUCHE faire                       L+1
        AVANCER_GAUCHE                                   lg
      ftantque
arrêt : FINAL_DROIT                                      ld+1
       AVANCER_DROIT                                     ld
fitérer
itérer VISITER(NC)                                       N
arrêt : EST_RACINE                                       N
        {remontée}
      si A_DROITE                                        N-1
         alors AVANCER_PERE                              FD
         sinon AVANCER_PERE                              FG
               si FINAL_DROIT alors                      FG
                  AVANCER_DROIT                          NT2=NF-1
                  itérer tantque FINAL_GAUCHE faire      N-L-1
                          AVANCER_GAUCHE                 FG-lg
                        ftantque
                  arrêt : FINAL_DROIT                    FD-ld
                          AVANCER_DROIT                  FD-NF-ld+1
                  fitérer
               fsi
      fsi
fitérer
```

Evaluation :

accès arborescent: *DEMARRER_RACINE*:1, *AVANCER_GAUCHE*:FG, *AVANCER_DROIT*:FD
AVANCER_PERE:N-1, *FINAL_GAUCHE*:FG, *FINAL_DROIT*:N+FD
EST_RACINE:N, *A_DROITE*:N-1, *NC*:1

contrôle : 4N-1

Algorithme : parcours en ordre terminal d'un arbre binaire - version 3

{on applique la version 1 au cas d'un arbre binaire homogène}

```
DEMARRER_RACINE                                          1
tantque ¬FINAL_GAUCHE faire                              lg+1
  AVANCER_GAUCHE                                         lg
ftantque
itérer VISITER(NC)                                       N
arrêt : EST_RACINE                                       N
        si A_DROITE                                      N-1
           alors AVANCER_PERE                            FD=(N-1)/2
           sinon AVANCER_PERE ; AVANCER_DROIT            FG=(N-1)/2
                 tantque ¬FINAL_GAUCHE faire             2FG-lg=N-lg-1
                   AVANCER_GAUCHE                        FG-lg
                 ftantque
        fsi
fitérer
```

Evaluation :

accès arborescent: *DEMARRER_RACINE*:1, *AVANCER_GAUCHE*:(N-1)/2,
AVANCER_DROIT:(N-1)/2
AVANCER_PERE:N-1, *FINAL_GAUCHE*:N-1, *FINAL_DROIT*:0
EST_RACINE:N
A_DROITE:N-1, *NC*:N

contrôle : 3N-1

d) Evaluation :

L'évaluation de ces 4 versions est faite dans les conditions décrites au § 6.2.2. lors de l'étude du préordre. Les "coûts 1" correspondent au cas où chaque noeud porte l'information *ESTDROIT*, les "coûts 2" correspondent au cas où l'information *ESTDROIT* est calculée au cours du parcours à l'aide d'une pile.

version	1	2	3	4
coût 1	14N-3	15N-3	11N-5	12N-3
coût 2	20N-3	21N-3	17N-8	18N-6

On peut constater, que les versions utilisant une pile (2 et 4) sont plus chères que les autres (contrairement à ce qui se passait pour les deux autres ordres de parcours). Par ailleurs les différences entre les versions sont moins importantes que précédemment.

Remarque :

Ceci peut être expliqué intuitivement : comme le parcours en ordre terminal impose que tout noeud soit visité après ses descendants, il y a une remontée systématique au noeud après le parcours du sous-arbre droit. (ce qui n'était pas le cas pour les autres parcours)

e) Comparaison des trois parcours étudiés

Nous concluons cette étude des parcours d'arbre en proposant un tableau récapitulatif qui permet de comparer les parcours.

		préordre	symétrique	terminal
général	version sans pile	35N/2 (v.1)	28N/2 (v.1)	28N/2 (v.1)
	version avec pile	25N/2 (v.4)	25N/2 (v.4)	30N/2 (v.2)
homogène	version sans pile	19N/2 (v.5)	12N/2 (v.5)	22N/2 (v.3)
	version avec pile	19N/2 (v.7)	19N/2 (v.6)	24N/2 (v.4)

Nous avons de plus élaboré un ensemble de schémas et fourni des évaluations permettant d'en vérifier la cohérence, d'en juger le comportement et d'en estimer le coût (dans un contexte particulier à titre d'exemple).

Nous montrons dans le prochain paragraphe comment utiliser directement ces schémas au cours de la résolution d'un problème.

6.2.5. - Conclusion

L'étude des parcours d'arbres a un rôle exemplaire dans la mesure où nous avons utilisé l'ensemble des résultats présentés dans les chapitres précédents, au cours d'un processus de construction décomposé en plusieurs étapes :

1. l'application du traitement arborescent comportant :
   . la définition et la description de l'arbre (donné dans notre énoncé généra
   . une analyse récurrente (selon le § 5.1.6.) conduisant à un algorithme récursif ((5.2.1.).

2. l'application du traitement séquentiel comportant
   . la définition d'une file selon les principes du chapitre 1 (§ 1.1.)
   . la reconnaissance d'un problème de parcours de file.

3. la construction d'algorithmes itératifs comportant
   . la définition d'une machine séquentielle
   . l'étude du répertoire de cette machine, selon le modèle d'analyse du § 5.1.6.

4. la mise en évidence de divers choix possibles et de cas particuliers conduisant à de nombreuses solutions (qui recouvrent les solutions couramment proposées).

Cette étude montre la puissance de nos méthodes face à un problème algorithmique difficile (quoique résolu depuis longtemps): l'application successive du traitement arborescent et du traitement séquentiel a permis la construction systématique d'un ensemble de solutions à ce problème (ce l'on peut opposer à une présentation de ces solutions qui est en général très intuitive).

6.3. - UTILISATION DES PARCOURS D'ARBRES DANS UN TRAITEMENT ARBORESCENT

Jusqu'à présent, nous avons basé la résolution d'un traitement arborescent sur les deux étapes suivantes :

1. structuration en arbre, en appliquant le modèle donné au § 5.1.2.
2. résolution du problème par analyse récurrente en appliquant le modèle donné au § 5.1.6.

Nous complétons la méthode en proposant une alternative à la deuxi étape :

2'. résolution du problème par application du traitement séquentiel à l'ar qui a été défini.

En d'autres termes, nous suggérons de ramener le traitement de l'arbre à un traitement élémentaire de chacun de ses noeuds : chaque noeud est traité une fois et une seule, et le cumul de ces traitements résoud le problème posé. Les parcours d'arbres ont été présentés (§ 5.2.1.) comme des modèles de cette démarche, et les algorithmes fournis aux paragraphes 5.2.1. et 6.2. sont autant de schémas généraux directement utilisables dès que l'on peut ramener un problème à un tel parcours.

Remarque :
Nous n'avons considéré que les trois parcours les plus couramment utilisés ; tout autre parcours d'arbre peut intervenir dans notre démarche de manière analogue ; c'est notamment le cas du parcours par "niveau" (nous donnons un exemple plus loin, § 7.3.).

Pour expliciter cette manière de faire, nous donnons tout d'abord un exemple, puis nous étudions les algorithmes fondamentaux du § 5.2. Enfin la démarche sera constamment utilisée dans le chapitre 7.

6.3.1. - Exemple : profondeur d'un arbre

Nous reprenons cet exemple pour lequel nous avons donné une analyse récurrente au § 5.1.6. Nous proposons maintenant l'analyse suivante :

La profondeur de l'arbre est la valeur maximum que peut prendre le niveau d'un noeud quelconque de l'arbre. Plus précisément, c'est la valeur maximum des niveaux des feuilles de l'arbre. Nous savons simplement calculer la valeur maximum d'une suite de nombres. Notre problème est résolu si l'on sait d'une part associer à chaque feuille son niveau, d'autre part énumérer l'ensemble des feuilles.

a) Traitement arborescent

Pour associer à chaque feuille de l'arbre son niveau, nous appliquons le traitement arborescent en définissant un nouvel arbre à partir de l'arbre donné :

1. Description de l'information :
   Un noeud du nouvel arbre est constitué d'un noeud de l'arbre donné et d'un entier correspondant au niveau du noeud.

2. Description des relations :

(*x* désigne un noeud , *niv* son niveau).

```
RACINE'           : RACINE,1
G'(x,niv)         : G(x), niv+1
D'(x,niv)         : D(x), niv+1
E_G'(x,niv)       : E_G(x)
E_D'(x,niv)       : E_D(x)
PERE'(x,niv)      : PERE(x), niv-1
ESTDROIT'(x,niv)  : ESTDROIT(x)
```

3. Traitement

Comme nous l'avons vu au § 5.2.1., l'énumération des feuilles de cet arbre peut être réalisée en appliquant l'un quelconque des parcou d'arbre, dans lequel seules les feuilles sont visitées. Le choix du parcours ne dépend pas ici du traitement puisque la recherche d'un maximum ne dépend pas de l'ordre dans lequel les éléments sont examinés. Nous prenons par exemple le préordre. Nous obtenons ainsi une solution, soit sous forme récursive, soit sous forme itérative.

b) Algorithmes.

b.1) Forme récursive :

Algorithme : profondeur d'un arbre binaire (forme 1)

```
entier p     {pour cumuler la valeur maximum}
action prof(arbre X, entier NIV)
  si ¬E_G(X) et ¬E_D(X) alors     {on applique un parcours en préordre
  si NIV>p alors p:=NIV fsi        seules les feuilles sont visitées}
  fsi
  si E_G(X) alors prof(G(X),NIV+1) fsi
  si E_D(X) alors prof(D(X),NIV+1) fsi
faction
{le calcul de la profondeur de l'arbre donné est fait par la séquence
 suivante:}
p:=0 ; prof(RACINE,1) ;
{le résultat est p}
```

b.2) Forme itérative :

L'algorithme est obtenu directement à partir de ceux que nous présentons au paragraphe suivant (§ 6.3.2.) en réalisant la machine arbre à partir de la définition du nouvel arbre que nous avons donnée ci-dessus. Par exemple si on utilise la version 1 donnée au § 6.3.2. on obtient :

+ machine arbre (on utilise la machine arbre de l'arbre donné)

. état :

Le noeud courant est constitué du noeud courant *NC* de l'accès arborescent à l' arbre donné et d'un entier *niv* donnant le niveau de ce noeud.

. répertoire :

(on définit les primitives requises par la version 1)

```
DEMARRER_RACINE' : DEMARRER_RACINE ; niv:=1
AVANCER_GAUCHE'  : AVANCER_GAUCHE ; niv := niv+1
AVANCER_DROIT'   : AVANCER_DROIT  ; niv:= niv+1
FINAL_GAUCHE'    : FINAL_GAUCHE
FINAL_DROIT'     : FINAL_DROIT
A_DROITE'        : A_DROITE
AVANCER_PERE'    : AVANCER_PERE ; niv:=niv-1
```

+ traitement

. accumulateur : un entier *p*

. répertoire :

```
INIT_P : p:=0
VISITER(NC,niv) : si niv>p alors p:=niv fsi
TERM_P : "le résultat est p"
```

+ algorithme

il est obtenu en substituant dans la version 1 de l'algorithme d'énumération d'une feuille (voir § 6.3.2.) les primitives par les définitions données ci-dessus.

Remarque :

La solution issue d'une analyse récurrente (§ 5.1.6.) implique un parcours en ordre terminal de l'arbre binaire. (c'est une "évaluation postfixée" de l'arbre binaire). Il est intéressant de noter que, l'on peut passer d'une solution à l'autre de manière quasi automatique par des techniques de transformations comme celles définies dans [BuD 77].

6.3.2. - Enumération des feuilles d'un arbre binaire

Nous considérons le cas particulier où le parcours d'arbre n'est fait que pour visiter les feuilles. Une solution consiste à appliquer l'un quelconque des algorithmes précédents, l'action VISITER n'étant appliquée que si le noeud est une feuille. Du point de vue des coûts, ceci peut ne pas être négligeable (dans les hypothèses que nous avons prises pour faire les évaluations précédentes, ceci augmente le coût total de 6N unités). C'est pourquoi, nous cherchons un algorithme qui mette à part la visite des feuilles : un exemple peut être obtenu à partir de la version 2 du préordre : l'itération principale est changée en :

Algorithme : {variante de la version 2 du préordre pour énumérer les feuilles}.

```
... {cf version 2 préordre}
...
itérer tantque ¬FINAL_GAUCHE faire AVANCER_GAUCHE ftantque
          {on ne visite plus les non-terminaux}
arrêt : NC=DP
        si ¬FINAL_DROIT
           alors AVANCER_DROIT
           sinon {le noeud est une feuille}
                 VISITER_FEUILLE(NC)
                 {remontée}
                 ...
        fsi
fitérer
VISITER_FEUILLE(NC)      {le dernier en préordre est une feuille}
```

On peut de même adapter toutes les versions du préordre qui "regroupent les descentes à gauche".

Mais on peut développer une analyse particulière pour résoudre l'énumération des feuilles : on considère la suite des feuilles, et on applique un traitement séquentiel. Nous considérons par exemple que l'ordre qui définit cette suite est l'ordre dans lequel les feuilles sont visitées dans les parcours précédents. Dans ces conditions, à la suite d'une analyse récurrente totalement analogue à ce que nous avons fait pour les parcours, on obtient par exemple :

Algorithme : Enumération des feuilles d'un arbre binaire - version 1

```
{calcul de la dernière feuille DF}
DEMARRER_RACINE
itérer tantque ¬FINAL_DROIT faire AVANCER_DROIT ftantque
arrêt : FINAL_GAUCHE
        AVANCER_GAUCHE
fitérer
DF:=NC
{parcours proprement dit}
itérer {NC est soit la racine, soit un fils droit
        on cherche la première feuille dans ce sous-arbre}
      itérer tantque ¬FINAL_GAUCHE faire AVANCER_GAUCHE ftantque
      arrêt : FINAL_DROIT
              AVANCER_DROIT
      fitérer
      VISITER_FEUILLE(NC)
arrêt : NC=DF
      {remontée à la recherche du prochain sous-arbre droit}
      répéter dedroite:=A_DROITE ; AVANCER_PERE
      jusqua -dedroite et -FINAL_DROIT
      AVANCER_DROIT
fitérer

{le coût de cet algorithme dans les hypothèses que nous avons prises
 pour l'évaluation des parcours d'arbre est de : 13N+3FG+3NF-3ld+3}
```

Algorithme : énumération des feuilles d'un arbre binaire - version 2

```
{la remontée dans l'arbre est réalisée à l'aide d'une pile gérée à
 l'aide de primitives telles que celles définies pour la version 4
 du préordre}
VIDER_PILE ; DEMARRER_RACINE
itérer itérer tantque ¬FINAL_GAUCHE faire
                 si ¬FINAL_DROIT alors EMPILER fsi ; AVANCER_GAUCHE
                 ftantque
       arrêt : FINAL_DROIT
               AVANCER_DROIT
       fitérer
       VISITER_FEUILLE(NC) ; DEPILER
arrêt : NC=FICTIF
        AVANCER_DROIT
fitérer

{le coût de l'algorithme dans les conditions habituelles est 4N+7NF+3}
```

Remarque :

Dans le cas des arbres binaires complets, on obtient les versions 5 et 7 du parcours en préordre.

6.3.3. -"Evaluation postfixée" d'un arbre binaire

a) Algorithme récursif

Nous reprenons l'algorithme donné au § 5.2.1. c :

```
fonction ev_post(arbre A) → result
   choix
   E_G(A) et E_D(A)    : → f2(A,ev_post(G(A)),ev_post(D(A))
   E_G(A) et ¬E_D(A)   : → f1(A,ev_post(G(A))
   ¬E_G(A) et E_D(A)   : → f1(A,ev_post(D(A))
   ¬E_G(A) et ¬E_D(A)  : → f0(A)
   fchoix
ffonction ev_post
```

{fo,f1 et f2 sont des fonctions qui permettent le calcul de la valeur d'un noeud en fonction de celles de ses fils. Ce sont ces fonctions qui expriment la réduction du problème lorsque l'on applique un schéma d'évaluation postfixée}

b) Construction d'un algorithme itératif

Nous cherchons à écrire un algorithme itératif qui calcule la valeur *ev_post(RACINE)* pour un arbre donné.

Une première étape consiste à mettre en évidence la gestion des valeurs de type *result* qui sont successivement attribuées à chaque noeud de l'arbre. De manière classique, nous avons recours à une pile de valeurs de type *result* que l'on peut manipuler avec les primitives *EMP(r)* (empiler

x en haut de la pile), $x:=DEP$ (mettre le sommet de pile dans x et décapiter la pile) et *VIDER* (initialisation de la pile).

Nous pouvons alors écrire l'action récursive suivante :

```
action ev_post1 (arbre A)
  {met la valeur de A en haut de la pile de valeurs, sans modifier
   le reste de la pile}
  choix
      E_G(A) et E_D(A)    :  ev_post1(G(A)) ; ev_post1(D(A))
                             xd:=DEP ; xg:=DEP ; EMP(f2(A,xg,xd))
      E_G(A) et ¬E_D(A)   :  ev_post1(G(A)) ; EMP(f1(A),DEP))
      ¬E_G(A) et E_D(A)   :  ev_post1(D(A)) ; EMP(f'1(A), DEP))
      ¬E_G(A) et -E_D(A)  :  EMP(f0(A))
  fchoix
ffaction ev_post1
  {xg et xd sont des variables de travail}
```

et la valeur de l'arbre donné est calculée par la séquence :

VIDER ; ev_post1(RACINE)

le résultat se trouvant en haut de la pile.

On identifie dans cette action un parcours en ordre terminal de l'arbre donné et la deuxième étape de l'écriture d'une forme itérative pour ce calcul consiste simplement à appliquer un algorithme de parcours d'arbre en ordre terminal, la visite d'un noeud étant définie comme suit :

cas	*VISITER(X)*
E_G(A) et E_D(A)	*xd:=DEP ; xg:=DEP ; EMP(f2(A,xg,xd))*
E_G(A) et ¬E_D(A)	*EMP(f1(A,DEP))*
¬E_G(A) et E_D(A)	*EMP(f1(A,DEP))*
¬E_G(A) et -E_D(A)	*EMP(f0(A))*

6.3.4. - Enumération partielle d'un arbre

Comme nous l'avons vu au paragraphe 5.2.2., l'énumération partielle d'un arbre est définie comme un cas particulier de parcours d'arbre dans lequel un sous-arbre n'est parcouru que si sa racine vérifie une propriété donnée. L'écriture d'un algorithme itératif correspondant à une telle énumération partielle est une simple application du trairement arborescent : on applique un algorithme de parcours d'arbre à l'arbre défini à partir de l'arbre donné, en supprimant tous les sous-arbres dont la racine ne vérifie pas le prédicat P (nous avons donné un exemple de définition de cet arbre au § 5.2.2.).

6.3.5. - Recherche d'un élément dans un arbre

Nous reprenons ce problème tel que nous l'avons défini au § 5.2.3. : on recherche dans un arbre binaire donné *A* s'il existe un noeud vérifiant une propriété donnée *P*. En 5.2.3. nous avions fait l'analyse récurrente de ce problème. Appliquant maintenant le traitement séquentiel, il suffit de choisir l'un des ordres de parcours d'arbre et la machine séquentielle associée. Par exemple, si nous choisissons le préordre et en utilisant la machine séquentielle de parcours en préordre décrite au § 6.2.1., nous pouvons écrire l'algorithme suivant :

Algorithme : recherche d'un élément dans un arbre binaire

```
DEMARRER_PREORDRE
tantque ¬P(NC) et ¬FINAL_PREORDRE faire
  AVANCER_PREORDRE
ftantque
si P(NC)
   alors ...{on a trouvé en NC}
   sinon ...{on n'a pas trouvé}
fsi
```

Remarques :

- cet algorithme est une application directe des schémas de recherch d'un élément dans une file proposée au § 2.2.4.
  On voit ici encore comment le recours au traitement arborescent et au traitement séquentiel peuvent intervenir simultanément dans une analyse.
- Une autre manière de faire consiste à intervenir directement sur les algorithmes de parcours proposés au § 6.2. et de les modifier de manière à arrêter le parcours dès qu'un élément vérifiant *P* a été rencontré : ceci ne pose que peu de problèmes dans les version directement fondées sur l'idée d'un traitement séquentiel des noeuds (les versions 1 pour les trois parcours). Les modifications sont plus délicates dans les autres cas.

6.4. - CONCLUSION - RESUME DU TRAITEMENT ARBORESCENT

Nous résumons ici la méthode du traitement arborescent en récapitulant d'une part l'ensemble des modèles et outils proposés, d'autre part les étapes du processus de construction de programmes qu'implique l'application systématique de cette méthode. Ces deux récapitulatifs sont organisés en fonction des deux principes de base : recherche d'une organisation logique de l'information et formulation du problème posé en termes de cette organisation.

6.4.1. - Récapitulatif des modèles et outils proposés

A. Recherche d'une organisation logique de l'information en une arborescence d'informations élémentaires.

1. modèle du processus de structuration en arbre (§ 5.1.2.)
2. modèle de description de l'arbre (§ 5.1.3.) comportant
   - la description de l'information élémentaire (résultat du raffinement)
   - la description des relations (résultat de la structuration)
3. modèle d'accès arborescent, appelé "machine arbre"
4. modèle de description d'une machine arbre comportant
   - la description de l'état de la machine
   - la description du répertoire d'actions élémentaires permettant le déplacement dans l'arbre

Remarque :
Nous distinguons deux types d'arbres (§ 5.1.1.) :
Les arbres binaires et les arbres non binaires (cas général). Toute la présentation du traitement arborescent est essentiellement basée sur le cas des arbres binaires parce que cette structure est plus simple (nombre maximum de fils d'un noeud connu a priori) , et qu'il existe une application du traitement arborescent qui permet de passer systématiquement du traitement d'un arbre non binaire au traitement d'un arbre binaire (§ 5.1.4.).

B. Recherche d'une formulation du problème posé en termes d'un traitement de cet arbre.

1. un modèle d'analyse récurrente (§ 5.1.6.)
2. un ensemble de schémas fondamentaux répartis en trois groupes :
   - parcours d'arbres (§ 5.2.1., § 6.2., § 6.3.2. et § 6.3.3.
   - énumération partielle d'un arbre (§ 5.2.2. et § 6.3.4.)
   - recherche dans un arbre (§ 5.2.3. et § 6.3.5.)

Remarque :
Ici encore les modèles proposés sont dédoublés en fonction des deux types d'arbres. Toutefois les schémas ne sont étudiés que pour les arbres binaires (sauf au § 5.2.1.).

6.4.2. - Etapes du processus de construction de programmes

Dans le tableau suivant, nous décrivons chaque étape de ce processus en le qualifiant d'un nom et en explicitant la nature de son résultat

A. Organisation logique de l'information		
	1. Raffinement	description de l'information élémentaire
	2. Structuration	description des relations
	3. accès arborescent	description de la machine arbre - état - répertoire
B. Formulation du problème en termes d'un traitement d'arbre		
	1. Analyse récurrente (choix entre deux types d'analyse par cas)	Algorithme récursif
	1' Identification d'un problème général	Choix d'un schéma
	2. Examen de cas particuliers	raffinement du schéma
	3. Evaluation	mesure de comportement de l'algorithme

CHAPITRE 7

APPLICATION DU TRAITEMENT ARBORESCENT A L'ETUDE D'ALGORITHMES RECURSIFS

La méthode du traitement arobrescent conduit en général à des algorithmes récursifs. Inversement, on peut étudier un tel algorithme à l'aide de cette méthode : le traitement arborescent aide alors à évaluer le coût de l'algorithme, à le coder dans un contexte de programmation particulier, éventuellement à l'optimiser et il contribue ainsi à une meilleure maîtrise de l'algorithme récursif étudié.

Nous illustrons ces divers points au travers d'exemples, après avoir présenté de manière générale le principe du traitement arborescent que l'on peut associer à tout algorithme récursif. De nombreux autres exemples sont donnés dans [Sch 76], où les idées du traitement arborescent ont été présentées pour la première fois.

7.1 - TRAITEMENT ARBORESCENT ASSOCIE A UN ALGORITHME RECURSIF

7.1.1. - Forme générale des algorithmes considérés

a) Schéma général

Nous considérons le schéma général suivant, qui recouvre un grand nombre d'algorithmes récursifs :

```
action AC (param P)
 ALPHA(P)
 si A1(P) alors AC(T1(P)) fsi
 BETA(P)
 si A2(P) alors AC(T2(P))
 GAMMA(P)
faction AC
```

où nous employons les conventions suivantes :

. *P* désigne l'ensemble des paramètres de l'action (appel par valeur). Le type *param* représente l'ensemble des types de ces paramètres.

. *A1* et *A2* sont des prédicats dont la valeur dépend uniquement de l'examen du paramètre *P*

. *T1* et *T2* sont des fonctions qui étant donné une valeur de type *param* , calculent une valeur de type *param*.

. *ALPHA*, *BETA* et *GAMMA* sont des actions dont l'effet dépend de *P* , mais qui ne modifient pas *P*. Leur effet est spécifié en fonction d'un ensemble de variables externes à l'action.

b) Cas particulier

Un cas particulier de ce schéma, qui survient assez fréquemment, est le suivant :

```
action AC1(param P)
  si A(P)
     alors DELTA(P)
     sinon ALPHA(P) ; AC1(T1(P)) ; BETA(P); AC2(T2(P));GAMMA(P)
  fsi
faction
```

7.1.2. - Arbre associé à un algorithme récursif

Nous considérons l'appel initial de l'action : *AC(X)*
Cet appel engendre un nombre fini d'appels de *AC* . C'est cet ensemble d'appels que nous organisons en arbre.

a) Structuration en arbre de l'ensemble des appels

. la racine de l'arbre est l'appel initial *AC(X)*
. le sous-arbre gauche est défini comme étant l'arbre associé à l'ensemble des appels engendrés par l'appel*AC(T1(X))* . Ce sous-arbre est éventuellement vide (dans le cas où *A1(X) = faux)*.
. le sous-arbre droit est défini comme étant l'arbre associé à l'ensemble des appels engendrés par l'appel *AC(T2(X))* . Ce sous-arbre droit est éventuellement vide (dans le cas où *A2(X) = faux)*.

b) Description de l'arbre

Cette définition récurrente de l'arbre montre qu'un noeud peut être caractérisé par la valeur du paramètre qui est passé lors de l'appel correspondant. Dans ces conditions, nous pouvons définir l'arbre comme suit :

. description de l'information
un noeud est une valeur de type *param*
. description des relations

RACINE : *X*
G(P) : *T1(P)*
D(P) : *T2(P)*
E_G(P) : *A1(P)*
E_D(P) : *A2(P)*

Au niveau d'abstraction où nous nous plaçons, nous ne pouvons rien dire des fonctions *PERE* et *ESTDROIT* . De manière générale, elles peuvent être définies en appliquant les techniques présentées au paragraphe 6.1.3. Nous illustrerons ceci dans les exemples qui suivent.

Remarque :
Dans le cas particulier de l'action *AC1* , on obtient un arbre binaire homogène.

7.1.3. - Traitement de l'arbre associé à un algorithme récursif

Ayant défini l'arbre associé à un appel *AC(X)* , on peut l'étudier et notamment essayer d'interpréter l'effet de *AC(X)* comme un traitement de cet arbre. La forme générale que nous avons choisie pour le schéma *AC* ne permet pas d'être plus précis sur cette étude. Remarquons toutefois qu'à ce niveau d'abstraction, on peut toujours interpréter l'effet de l'appel *AC(X)* comme

un parcours particulier de l'arbre des appels : il s'agit de combinaison des trois parcours d'arbres que nous avons étudiés précédemment. (Un algorithme itératif peut être écrit, sur la base des algorithmes itératifs de parcours du § 6.2.).

Mais nous sommes souvent en présence d'algorithmes récursifs qui sont des cas particuliers du schéma général que nous avons donné. On peut alors se ramener directement à l'un des algorithmes fondamentaux que nous avons présentés dans les chapitres précédents. Cette identification permet d'utiliser les résultats acquis sur le problème reconnu et notamment d'utiliser les schémas itératifs correspondant.

7.1.4. - Cas des algorithmes récursifs engendrant plus de deux appels récursifs.

a) Schéma général

Il s'agit d'algorithmes qui expriment la solution d'un problème par une réduction de ce problème en N problèmes de même nature. Un exemple de schéma général est le suivant :

```
action ACNB (param P)
  entier c ; param pp
  ALPHA(p)
  si A(p) alors
     c:=1 ; pp:=TO(p)
     itérer ACNB(pp)
     arrêt : C=NB(p)
             BETA(p) ; pp:=T1(p) ; c:=c+1
     fitérer
  fsi
  GAMMA(p)
faction ACNB
```

Ce schéma exprime la solution du problème caractérisé par *p* sous forme d'un traitement séquentiel de *NB(p)* sous-problèmes.

. *TO* et *T1* sont comme précédemment des fonctions qui engendrent une valeur de type *param* , à partir d'une valeur de type *param* . Elles permettent de définir la file des sous-problèmes à traiter pour résoudre le problème donné.

Remarque :
Nous avons pris un cas particulier dans la mesure où nous supposons pouvoir calculer le nombre de sous-problèmes *NB(P)* . Une autre forme de schéma peut être donnée, dans lequel le dernier élément de la file de sous-problèmes est défini par un prédicat.

. *A* est un prédicat qui permet de reconnaître la situation où le problème est résolu de manière simple (sans réduction supplémentaire).

. *ALPHA*, *BETA* et *GAMMA* sont des actions qui répondent aux conditions données pour le schéma précédent (§ 7.1.1.).

b) Principe de l'étude

Pour traiter ce type d'algorithme, on peut toujours utiliser la méthode suivante, qui est fondée sur une transformation générale d'algorithmes récursifs (cf. par exemple [BPP 76]) :

. on exprime l'itération que comporte l'action *ACNB* sous forme récursive. On obtient :

```
action ACB (param p, pp ; entier c)
  ACNB(pp)
  si c≠NB(p) alors BETA(p) ; ACB(p,T1(pp),c+1) fsi
faction ACB
```

. L'action *ACNB* s'écrit alors :

```
action ACNB (param p)
  ALPHA(p)
  si A(p) alors ACB(p,TO(p),1) fsi
  GAMMA(p)
faction ACNB
```

. On supprime maintenant toute occurence de *ACNB* par substitution :

```
action ACB (param p, pp ; entier c)
  ALPHA(pp)
  si A(pp) alors ACB(pp,TO(pp),1) fsi
  GAMMA(pp)
  si c≠NB(p) alors BETA(p) ; ACB(p,T1(pp),c+1) fsi
faction ACB
{l'appel initial ACNB    étant :}
ALPHA(x) ; si A(x) alors ACB(x,TO(x),1) fsi ; GAMMA(x)
```

Nous nous sommes ainsi ramené à une action récursive qui répond aux conditions du schéma donné au § 7.1.1.

c) Cas particulier

Dans certaines conditions, on peut appliquer le traitement arborescent directement sur l'action *ACNB* : il s'agit de cas où l'on peut reconnaître un schéma de parcours d'arbre non binaire. Plus précisément, on peut donner en exemple le schéma suivant où l'action *ACNB* est un cas particulier du schéma précédent dans lequel l'action *BETA* n'est appliquée que pour *TO(p)* :

```
action ACNB1 (param p) :
  entier c ; param pp
  ALPHA(p)
  si A(p) alors
     pp:=TO(p) ; c:=1
     ACNB(pp) ; BETA(p)
     tantque c≠NB(p) faire pp:=T1(pp);c:=c+1;ACNB(pp) ftantque
  fsi
  GAMMA(p)
faction ACNB1
```

dans ces conditions, on peut associer à l'appel *ACNB1(x)* l'arbre des appels définis comme suit :

. un noeud est caractérisé par une valeur de type *param* et deux entiers.

. les primitives de l'arbre sont les suivantes :

```
RACINE          : x, 1, 1
FILS(p,c,nf)    : TO(p), 1, NB(p)
FRERE(p,c,nf)   : T1(p), c+1, nf
E_FILS(p,c,nf)  : A(p)
E_FRERE(p,c,nf) : c≠nf
```

De cet arbre on déduit un arbre binaire (cf. § 5.1.5.) et l'action *ACNB1* peut être interprêtée comme une combinaison des trois parcours d'arbres binaire

7.1.5. - Exemples :

On trouvera de nombreux exemples de cette démarche dans [Sch 76] : nous y traitons notamment les exemples suivants : algorithme du "quicksort" de HOARE, algorithme résolvant le problème des tours de HANOI, algorithme de multiplication de matrices de STRASSEN . Dans ce qui suit, nous proposons deux exemples qui montrent comment le traitement arborescent permet de construire un algorithme de manière très systématique à partir d'une solution récursive du problème considéré.

7.2. - EXEMPLE 1 - PERMUTATIONS

Nous considérons le problème suivant : étant donné un entier positif X , engendrer les permutations d'ordre X, c'est-à-dire toutes les permutations des X premiers entiers.

Remarque :
Nous avons évoqué une application du traitement arborescent à ce problème au § 5.2.2. (un sur-ensemble de l'ensemble des permutations est structuré en arbre, puis énuméré partiellement).

Nous travaillons ici sur une solution générale du problème proposée dans [Sed 77] : étant donné une permutation d'ordre X , si l'on échange les positions respectives de deux entiers, on obtient une nouvelle permutation. Le principe de l'algorithme est d'engendrer un ensemble d'échanges qui permettent d'énumérer toutes les permutations d'ordre X .

7.2.1. - Algorithme de départ (cf. [Sed 77])

. On fait pour l'instant abstraction de la représentation d'une permutation. On suppose seulement que P est un objet de type *PERMUTATION* et que l'on peut échanger les positions de deux entiers dans P à l'aide de l'action :

```
action ECH(entier i,j)
{échange les positions des éléments P_i et P_j}
```

. On considère l'algorithme récursif suivant :

```
action PERMUTATIONS (entier K)
{P étant dans un état initial P^i, engendre les K!-1 échanges
 qui produisent toutes les permutations d'ordre X se terminant
 par la séquence P^i_{k+1}...P^i_X     1 ≤ K ≤ N}
  entier C
  C:=1
  itérer si K>2 alors PERMUTATIONS(K-1) fsi
  arrêt : C=K
          {choix de la prochaine valeur à mettre en position K}
          ECH(K,f(K,C)  {f détermine la position de la prochaine
          imprimer (P)   valeur à mettre en K}
          C:=C+1
  fitérer
faction PERMUTATIONS
{l'appel initial est :}
INITIALISER_P ; imprimer (P) ; PERMUTATIONS(X)
{INITIALISER_P initialise  P par une permutation quelconque des
X premiers entiers}.
```

Remarques :

. Il existe de nombreuses manières de définir la fonction de choix f. Nous retenons la définition suivante attribuée à HEAP [Hea 63] par SEDGEWICK

f(K,C) = 1 si K est impair
f(K,C) = C si K est pair

. Dans ce qui suit, afin d'alléger la présentation, nous désignerons par *NOUVP(K,C)* la séquence *ECH(F,f(K,C)) ; imprimer P*

7.2.2. - Une autre forme de l'action récursive *PERMUTATIONS*

Nous nous ramenons à une forme d'action dans laquelle un appel ne peut engendrer que deux appels récursifs au plus.

Nous détaillons l'application de la technique proposée au § 7.1.4. :

. l'itération qui constitue le corps de l'action *PERMUTATION* est mise sous forme récursive :

```
action PERMB(entier K,C)
  si K>2 alors PERMUTATIONS(K-1) fsi
  si C≠K alors NOUVP(K,C) ; PERMB(K,C+1) fsi
faction PERMB
```

L'effet de l'appel *PERMB(X,1)* est alors le même que celui de l'appel *PERMUTATION(X)*.

. La nouvelle forme de l'action *PERMUTATIONS* est alors obtenue en substituant dans *PERMB* l'appel *PERMUTATIONS(K-1)* par l'appel de *PERMB* ayant le même effet, c'est-à-dire *PERMB(K-1,1)*

```
action PERMB(entier K,C)
  si K>2 alors PERMB(K-1,1) fsi
  si C≠K alors NOUVP(K,C) ; PERMB(K,C+1) fsi
faction PERMB
```

l'ensemble des permutations d'ordre N est alors engendré par la séquence :

INITIALISER_P ; imprimer(P) ; PERMB(X,1)

7.2.3. - Application du traitement arborescent

a) Arbre des appels

On définit l'arbre des appels de *PERMB* comme nous l'avons dit au § 7.1.2.

. un noeud de l'arbre est constitué de deux entiers

. Les primitives sont :

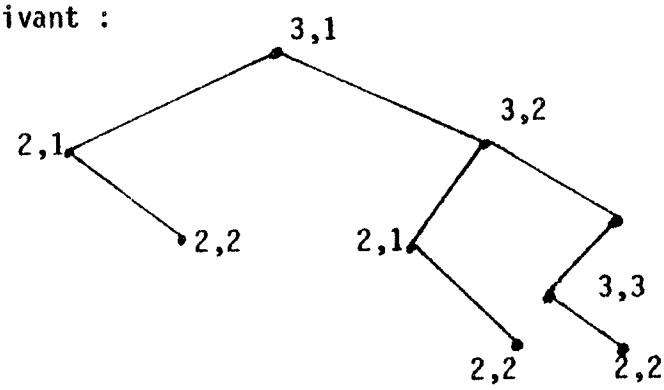
RACINE : *X,1*
G(K,C) : *K-1,1*
D(K,C) : *K,C+1*
E_G(K,C) : *K>2*
E_D(K,C) : *K≠C*
ESTDROIT(K,C) : *K≠1*

{la fonction *PERE* n'est pas entièrement définie}

PERE(K,C) : *K-1, ?* si *C=1*
K, C-1 si *C≠1*

b) Exemple

A l'appel *PERMB(3,1)* qui engendre les permutations d'ordre 3 correspond l'arbre suivant :



c) Evaluation :

On calcule les quantités qui nous servent à évaluer le coût des algorithmes (§ 6.1.4.).

1. nombre de noeuds : soit n_x le nombre de noeuds de l'arbre des appels issus de l'appel *PERMB(X,1)*

   On a : $n_1 = 1$
   $n_2 = 2$
   $n_x = x + x n_{x-1}$

   soit :

   $$n_x = x!\ (1/1! + 1/2! + \ldots + 1/i! + \ldots + 1/(x-1)!)$$

   dont une approximation pour x grand est : $n_x \tilde{=} \lfloor x!(e-1) \rfloor$

2. Le nombre de fils gauches fg_x et le nombre de fils droit fd_x de l'arbre des appels issus de *PERMB(X,1)* est calculé de la même manière :

   On a : $fg_1 = fg_2 = 0$
   $fg_x = x + x fg_{x-1}\ (x > 2)$

   soit :

   $$fg_x = x!\ (1/2! + \ldots + 1/i! + \ldots + 1/(x-1)!)$$

   dont une valeur approchée pour x grand est :

   $fg_x \tilde{=} \lfloor x!(e-2) \rfloor$

De même on a : $fd_1 = 0$
$fd_2^1 = 1$
$fd_x^2 = x-1 + xfd_{x-1}$ $(x > 2)$

soit : $1+fd_x = x(1+fd_{x-1})$
$1+fd_2^x = 2$
$1+fd_1^2 = 1$

soit :
$fd_x = x!-1$

. Enfin on peut calculer le nombre de feuilles de l'arbre des appels *PERMB(x,1)* :

On a : $nf_1 = 1$
$nf_2^1 = 1$
$nf_x^2 = xnf_{x-1}$ $(x > 2)$

soit :
$nf_x = x!/2$

d) Interprétation de *PERMB*

PERMB engendre une succession d'appels de *NOUVP* : pour l'appel *PERMB(K,C)* on engendre d'abord les appels de *NOUVP* correspondant à *PERMB(K-1,1)* (le cas échéant) puis on exécute *NOUVP(K,C)* et finalement on engendre les appels de *NOUVP* correspondant à *PERMB(K,C+1)*.

En termes de l'arbre que nous venons de définir, on reconnaît la définition du parcours en ordre symétrique (cf. § 5.2.1.) ou *NOUVP* joue le rôle de l'action élémentaire *VISITER* . (en fait on n'applique *NOUVP* que si $C \neq K$)

e) Ecriture d'un algorithme itératif

A partir de l'interprétation précédente, nous écrivons un algorithme itératif qui engendre les permutations d'ordre X , en appliquant l'une des versions développées au § 2.2.3.

Nous appliquons la version 4 de l'algorithme de parcours en ordre symétrique : en effet la fonction *PERE* n'est que partiellement définie. On obtient alors, en réalisant la machine arbre d'après la définition précédente de l'arbre :

a) Algorithme : permutations d'ordre X - forme 1

```
{Les commentaires montrent les relations avec la version 4 (§ 5.2.3.)}
{On fait pour l'instant abstraction de la représentation de la pile}
{le noeud courant est implémenté sous forme des 2 entiers K et C}
VIDER_PILE                                                        1
k:=x ; c:=1                          {démarrer_racine}            1
itérer tantque k>2 faire             {¬final_gauche}              n_x
         EMPILER(k,c)                                             fg_x
         k:=k-1 ; c:=1               {avancer_gauche}             fg_x
       ftantque                                                   x
       tantque k=c faire             {final_droit}                nx_{+1}
         (k,c):=DEPILER              {on ne visite pas}           fg_{x+1}
       ftantque                                                   x
arrêt : (k,c) = FICTIF                                            fd_{x+1}
        NOUVP                        {visiter}                    fd_x
        c:=c+1                       {avancer_droit}              fd_x
fitérer                                                           x
```

Pour que cet algorithme soit correct, il faut pouvoir choisir une valeur de *FICTIF* telle que *E_D(FICTIF)=faux* : il suffit de choisir k et c de sorte que $k \neq c$ et le couple (k,c) ne soit pas l'un des noeuds de l'arbre;
Ce choix sera fait lorsque l'on décidera de l'implantation de la pile.

7.2.4. - Réalisation de la pile

a) Principe

On doit empiler des couples d'entiers. On peut toutefois constater que chaque fois que l'on effectue l'opération *EMPILER* , on décrémente la valeur de k. Par ailleurs, c'est le seul endroit où k est modifié. Il existe donc une relation entre k et le nombre h d'éléments dans la pile :
$k+h = constante = x+1$ (en tenant compte du fictif empilé au départ). On choisit la réalisation de la pile en fonction de ces remarques :
On dispose d'un tableau d'entiers pc dans lequel sera implantée la pile.
La taille de ce tableau est choisie en fonction du nombre maximum de valeurs que l'on peut empiler. Tenant compte de l'élément fictif, cette taille est donc $(x-2)+1 = x-1$
Un élément de la pile soit $pc(i)$ a pour sens : i correspond à la valeur de k qui a été empilée, $pc(i)$ correspond à la valeur de c.
Dans ces conditions, on peut formuler la réalisation de la pile de la manière suivante :

```
{réalisation de la pile}
tableau (3:x+1) entier pc
VIDER_PILE : k:=x ; pc(x+1)=1
{fictif est le couple (x+1,1). Pour tester (k,c)≠fictif,
 il suffira de tester k≠x+1}
{k° représente la prochaine position libre pour empiler}
EMPILER : pc(k) := c ; k :=k-1
DEPILER : k:=k+1 ; → pc(k)
```

b) Algorithme

On peut maintenant écrire une deuxième forme de l'algorithme :

Algorithme : permutation d'ordre x - forme 2

tableau (3:x+1) pc ; entier k {pile}	
entier c	
pc(x+1):=1 ; k:=x ; c:=1	1
itérer tantque k>2 faire	n_x
pc(k):=c ; k:=k-1 ; c:=1	fg_x
ftantque	
tantque k=c faire	n_x+1
k:=k+1 ; c:=pc(k)	fg_x+1
ftantque	
arrêt :k=x+1	fd_x+1
NOUVP ; c:=c+1	fd_x
fitérer	

c) Evaluation de cet algorithme :

Nous supposons que k et c sont gérés dans des registres, et nous évaluons l'algorithme comme nous l'avons fait au § 6.2. :
Nous obtenons un coût de environ $19x!$ (sans compter la gestion proprement dite de la permutation). Dans ce coût, la part respective des deux itérations internes est de :

- première itération interne : environ $8x!$ soit environ 42% du coût total
- deuxième itération interne :environ $7x!$ soit environ 37% du coût total.

Nous cherchons à simplifier l'algorithme de manière à réduire les coûts de ces itérations.

7.2.5. - Simplification de l'algorithme

a) Principe

On constate que la première itération interne a pour effet (lorsque $k > 2$ au départ) d'empiler la valeur initiale de c, puis de remplir la pile avec des 1 (entre $k-1$ et 3), la valeur finale de k étant 2 , sauf lorsque x vaut 1 (nous écartons ce cas dans la discussion qui suit).

Nous cherchons à exprimer cela autrement que par une itération :

. On peut éviter de mettre des 1 dans la pile, si on arrive à assurer qu'à tout instant, les positions du tableau pc non occupées par la pile comportent des 1 : ceci est assuré, d'une part en initialisant le tableau pcavec des 1 , d'autre part en accompagnant chaque opération *DEPILER* d'une remise à la valeur 1 de la position du tableau ainsi libérée. Nous proposons donc une modification de l'action *VIDER_PILE* , et une modification de l'action *DEPILER*. Le coût de l'initialisation est négligeable (environ $3x$ unités), le coût de la modification dans *DEPILER* correspond exactement aux opérations $pc(k):=c$ effectuées dans l'itération que nous étudions.

. Il reste à effectuer l'empilement éventuel de la valeur de c au début de l'itération. Ceci peut être fait par la séquence :

si k > 2 alors pc(k):=c ; k:=2 fsi

qui vient remplacer la première itération interne.
Une solution plus intéressante consiste à assurer l'invariant suivant : la position $pc(k)$ (prochaine position où l'on va empiler) est toujours égale à la valeur de c. Dans ces conditions, la séquence précédente devient :

k:=2

(rappelons que nous écartons le cas où x vaut 1).
Pour assurer cet invariant, il suffit d'implanter c dans le tableau pc dans la position suivant le sommet de pile. Ceci nous impose d'agrandir la taille de pc de 1 position.

. Finalement, le cas $x=1$ peut être traité par un test initial. En fait l'algorithme qui suit est tel que l'on n'a pas besoin de traiter de cas à part.

b) Algorithme

Nous donnons maintenant la dernière forme de l'algorithme que nous obtenons : on y représente une permutation à l'aide d'un tableau *p* de *x* entiers. Nous choisissons pour première permutation, la permutation qui trie les entiers en ordre croissant.

Algorithme : permutations d'ordre x - forme finale

```
tableau(1:x) entier p                          {permutation}
tableau(2:x+1) entier pc ; entier k            {pile}
entier f,t                                     {pourNOUVP}
{initialisation de p et de pc}
pour k:=x pas -1 jusqà 1 faire p(k):=k fpour ; imprimer (p)
pour k:=x+1 pas -1 jusqua 2 faire pc(k):=1 fpour
itérer k:=2
       tantque k=pc(k) faire pc(k):=1 ; k:=k+1 ftantque
arrêt : k=x+1
        {NOUVP}
        si impair(k) alors f:=1 sinon f:=pc(k) fsi
        t:=p(k) ; p(k):=p(f) ; p(f):=t
        imprimer(p)
        pc(k):=pc(k)+1
fitérer
```

c) Evaluation :

Evaluant le coût de l'algorithme comme précédemment, et sans tenir compte du coût de l'exécution de *NOUVP*, nous obtenons un coût d'environ *15x!* (le gain est moins important qu'on pouvait l'espérer à cause de l'opération *pc(k):=pc(k)+1*). On a gagné environ 21% par rapport à la forme précédente. Ce gain est évidemment beaucoup moins significatif si l'on tient compte de *NOUVP* qui est effectué *x!-1* fois.

7.2.6. - Conclusion

Oubliant le fait que la forme finale apporte sans doute un gain très minime par rapport à la forme initiale (d'autant plus que le problème traité reste académique dans la mesure où pour de relativement faibles valeurs de x, le temps d'exécution prend des valeurs très grandes), cet exemple nous a permis :

- de montrer comment le traitement arborescent s'intègre parfaitement à un processus de production de programmes par transformations successives, fournissant des éléments autant pour construire les algorithmes que pour les analyser.

. d'illustrer toutes les étapes du traitement arborescent (§ 6.4.2.).

Remarque :
On peut reprendre la construction que nous avons faite en partant de la version 1 de l'algorithme en ordre symétrique (cf.[Sch 79 b]). On obtient alors un algorithme analogue à celui proposé dans [Sed 78]. On peut toutefois souligner les différences sensibles dans la manière de construire ces algorithmes.

7.3. - EXEMPLE 2 - UN ALGORITHME DE TRI

Cet exemple est utilisé dans [Ber 78] pour illustrer une technique particulière de transformation automatique de programmes récursifs. Nous le reprenons ici car il illustre clairement comment on peut utiliser successivement le traitement arborescent et le traitement séquentiel.

7.3.1. - Algorithme de départ

Il s'agit d'un tri par insertion, forme particulière du "diminishing increment sort" ([Knu 73] pp. 84 et suivantes) attribuée par KNUTH à V. PRAT

a) Définition

On considère un tableau de x éléments que l'on désire trier selon une relation d'ordre notée avec les symboles habituels.
Un tableau T[1:X] est dit *"h-trié"* (h entier ≥ 1) si la condition suivante est vérifiée : $\forall i : 1 \leq i \leq x-h,\ T(i) \leq T(i+h)$
Trier un tableau au sens habituel revient à le "1-trier".

b) algorithme

L'algorithme de PRATT est alors basé sur le principe suivant :
. Etant donné un tableau T qui est à la fois *"2-trié"* et *"3-trié"* on peut le *"h-trier"* en appliquant l'algorithme suivant :

```
action htri (entier h) :
  {le tableau est au départ "2h-trié" et "3h-trié", h < x}
  {à la fin il est "h-trié"}
  {traitement séquentiel de toutes les listes de pas h.
   A chaque liste, on applique le principe du tri par insertion
   en tenant compte du fait que le tableau est à la fois
   "2h-trié" et "3h-trié"}
  pour i:=1 pas 1 jusqua h faire
    {i est la tête d'une liste de pas h}
    j:=i+h                                  {prochain élément à insérer}
    tantque j ≤ x faire
       {T est "2h-trié" et "3h-trié"
        la liste de pas h de tête i est triée jusqua j-h}
       {T est"2h-trié" donc :T(j) ≥ T(j-2h) : il suffit de comparer
        T(j) et T(j-h)}
        si T(j) < T(j-h)
           alors échanger T(j) et T(j-h) {comme T est "2h et 3h-trié"
                                          l'échange conserve cette
                 j:=j+2*h {T est "2h-trié"}          propriété }
           sinon j := j+h
        fsi
    ftantque
  fpour
faction htri
```

. On définit alors l'algorithme suivant qui permet de "h-trier" un tableau T de taille x

```
action tri (entier h) :
 {"h-trie" le tableau T quelque soit son état initial}
  si h < x alors
     tri(2*h) ; tri(3*h) {l'ordre entre ces deux opérations
     htri(h)              est arbitraire}
  fsi
faction tri
```

. Le tableau est alors trié ("1-trié") par l'appel :

```
tri(1)
```

7.3.2. - Application du traitement arborescent

a) définition de l'arbre des appels

A partir de l'algorithme récursif ci-dessus, on définit l'arbre des appels associés à l'appel *tri(1)* :

. un noeud est caractérisé par un entier

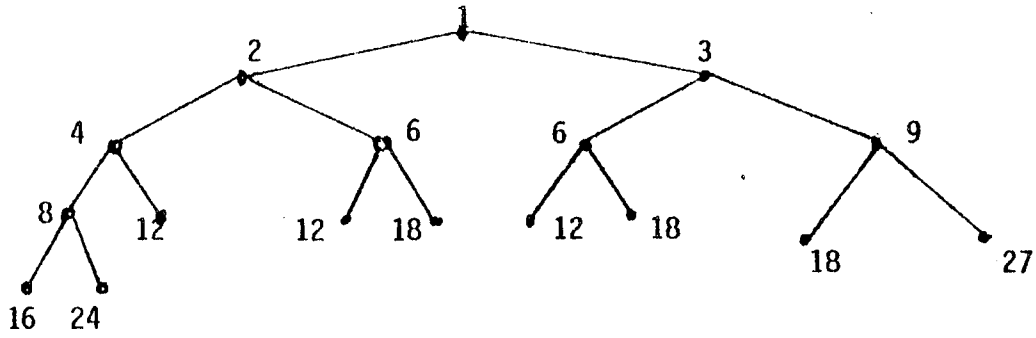
. les primitives qui définissent l'arbre sont :

RACINE : *1*
G(i) : *2*i*
D(i) : *3*i*
E_G(i) : *i* < *x*
E_D(i) : *i* < *x* {l'arbre est homogène}

Remarque :
On ne peut pas définir la fonction *ESTDROIT* et donc on ne peut pas définir la fonction *PERE*

b) Exemple :

L'arbre des appels engendrés par l'appel *tri(1)* avec *x=10* est le suivant :



c) Traitement de l'arbre

L'effet de l'appel *tri(1)* peut être interprété de la manière suivante : parcourir cet arbre en ordre terminal en appliquant l'action *VISITER* suivante :

. *VISITER(i)* = *htri(i)* si le noeud *i* n'est pas une feuille
. *VISITER(i)* = *vide* si le noeud *i* est une feuille

On peut donc produire un algorithme itératif en appliquant l'algorithme de parcours d'arbre binaire homogène adéquat (version 4 puisque les fonctions *PERE* et *ESTDROIT* ne sont pas définies, cf. § 6.2.4.).

7.3.3. - Simplification : une autre structuration en arbre

a) Analyse

Il est clair que cet algorithme n'est pas très efficace dans la mesure où parmi la succession d'appels de l'action *htri* qu'il engendre, un certain nombre de ces appels ont même valeur de paramètre au départ : l'effet de *htri(i)* est de *"i-trier"* le tableau *T*, de plus, une fois que *T* est *"i-trié"* (que que soit i), il le reste. Par conséquent, il est inutile de *"i-trier"* plusieurs fois le tableau *T* . Nous cherchons donc à supprimer tous ces appels inutiles de *htri*.

Examinons ce phénomène plus en détail :
la définition de l'arbre ci-dessus implique que la valeur i qui caractérise un noeud peut s'écrire sous la forme :

$$i = 2^p 3^q$$

où p est le nombre d'arcs gauche dans le chemin menant de la racine au noeud i

q est le nombre d'arcs droit dans le chemin menant de la racine au noeud i

A partir de cette remarque, il est facile de dénombrer les noeuds ayant la même valeur i: il y en a autant que de manières de combiner p éléments parmi $p+q$. Il y en a donc C^p_{p+q} (sauf pour le dernier niveau). De plus tous ces noeuds se trouvent au même niveau dans l'arbre, de valeur $p+q+1$. Nous cherchons maintenant à définir à partir de l'arbre ci-dessus, un nouvel arbre dans lequel tous les noeuds ont des valeurs distinctes et à définir la succession des appels de l'action *htri* nécessaires au tri du tableau T , en termes d'un parcours de ce nouvel arbre.

b) Traitement arborescent

Comme nous l'avons vu, tous les noeuds qui ont même valeur se trouvent au même niveau dans l'arbre de départ. Le parcours postfixé visite toujours les noeuds appartenant à un niveau donné dans l'ordre : gauche-droite. Dans un niveau $p+q+1$, parmi toutes les valeurs $i = 2^p 3^q$, nous allons conserver uniquement la plus à gauche, c'est-à-dire celle qui correspond au noeud atteint à partir de la racine par un chemin comportant d'abord p arcs gauche puis q arcs droit.

- Description de l'arbre

Ce nouvel arbre est alors défini de la manière suivante :

. un noeud est un entier i de la forme $i=2^p 3^q$
nous le caractérisons par les valeurs p et q

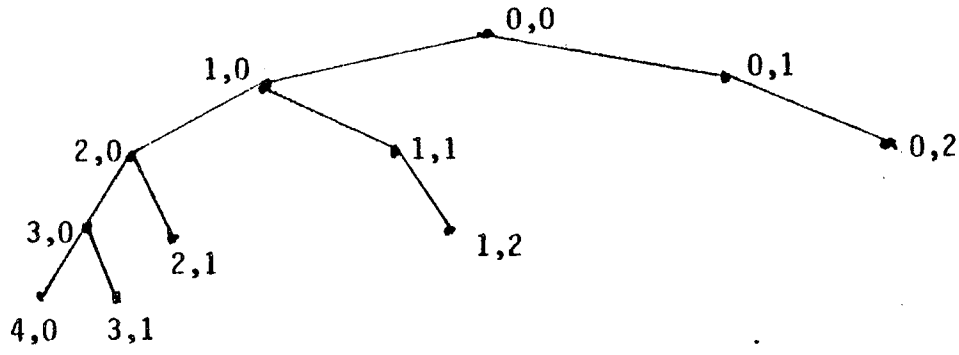
. Les primitives définissant le nouvel arbre sont alors :

RACINE'	: *0,0*	
G'(p,q)	: *p+1,q*	
D'(p,q)	: *p,q+1*	
E_G'(p,q)	: $q=0$ et $2^p 3^q < x$	
E_D'(p,q)	: $2^p 3^q < x$	
ESTDROIT'(p,q)	: $q \neq 0$	
PERE(p,q)	: *p, q-1*	*si* $q \neq 0$
	p-1,q	*si* $q=0$

Remarque :
On voit que la seule différence entre cet arbre et le précédent réside dans la condition d'existence d'un fils gauche.

c) Exemple :

Reprenant x=10, le nouvel arbre sera le suivant :
(nous mettons en face de chaque noeud les valeurs p et q)



Ainsi le nouvel arbre comporte l'ensemble des valeurs présentes dans l'ancien arbre, mais chaque valeur n'est représentée qu'une fois.

d) Algorithme récursif

Si l'on parcourt cet arbre en ordre terminal, on trie le tableau T dans la mesure où l'on applique la même action *VISITER* que précédemment. Nous pouvons prouver ceci sur la base de la forme récursive de ce parcours en ordre terminal :

appel : *ter(0,0)*

```
action ter(entier p,q)
  si q=0 et 2^p 3^q < x alors ter(p+1,q) fsi
  si 2^p 3^q < x alors ter(p,q+1) fsi
  si 2^p 3^q < x alors htri(2^p 3^q) fsi
faction ter
```

Remarque :
La forme de cette action peut évidemment être améliorée, mais nous nous préoccupons pour l'instant de prouver que l'appel initial trie le tableau T.

La preuve est faite en deux temps :

. tout d'abord, nous prouvons que l'action *htri* est toujours utilisée dans les bonnes conditions, c'est-à-dire que l'état initial de *htri(i)* est toujours "$i < x$ et T est *2i-trié* et *3i-trié*".

Ceci découle directement de la forme de l'action *ter* (on applique un raisonnement par récurrence basé sur le nombre d'appels de *ter*).

. Dans un deuxième temps, nous prouvons que l'appel *ter(p,q)* rend le tableau *"2^p3^q-trié"* . La preuve découle directement des spécifications de l'action *htri* et se base encore sur un raisonnement par récurrence
- c'est vrai pour *(p,q)* tel que $2^p3^q \geq x$
- on suppose que c'est vrai pour tout p,q tel que $p > p1$ ou $q > q2$ et on démontre alors aisément que c'est vrai pour *p1,q1*

. Finalement, il en découle que l'appel *ter(0,0)* trie le tableau *T*.

e) Algorithme itératif

Nous pouvons donc produire un algorithme itératif à partir des algorithmes du § 6.2.4. Nous adaptons la version 1 au cas présent : l'arbre est tel que tout noeud ayant un fils gauche a forcément un fils droit. D'autre part on ne visite pas les feuilles. Enfin, on sait que le premier noeud en ordre terminal est la première puissance de 2 supérieure ou égale à x.

Algorithme : tri-2-3. Version 1

```
{on maintient constamment la valeur i=2^p 3^q}
P:=0 ; q:=0 ; i:=1                                  {démarrer_racine}
tantque i<x faire p:=p+1 ; i:=2*i ftantque          {premier en ordre
tantque i≠1 faire                                    terminal}
 {remontée}                                         {i EST_RACINE}
  si q≠0                                            {-A_DROITE}
     alors i:=i/3 ; q:=q-1 ;                        {AVANCER_PERE}
           htri (i)
     sinon i:=i/2 ; p:=p-1                          {AVANCER_PERE}
           répéter i:=3*i ; q:=q+1                  {AVANCER_DROIT}
           jusqu'à i ≥ x
  fsi
ftantque
```

7.3.4. - Une_autre_forme_d'algorithme

a) Principe

La correction de l'algorithme ci-dessus nous assure que l'arbre que nous considérons comporte toutes les valeurs de noeuds nécessaires et suffisantes pour pouvoir trier le tableau donné *T* par une succession d'appels de l'action *htri*.

Les conditions d'utilisation de *htri(i)* sont

. $i < x$

. on a appliqué *htri(2*i)* et *htri(3*i)*

Clairement les noeuds $2*i$ et $3*i$ se trouvent au niveau directement supérieur au niveau de i.

b) Construction de l'algorithme

A partir de ces remarques, nous construisons une nouvelle forme d'algorithme :

. Nous conservons l'idée de parcours : chaque noeud est visité (la visite d'une feuille est vide).

. L'ordre dans lequel les noeuds sont visités est le suivant : on parcourt l'arbre par niveau, dans l'ordre des valeurs de niveau décroissantes (on commence par le niveau le "plus bas" et on remonte vers la racine). Dans un niveau, l'ordre dans lequel les noeuds sont visités importe peu.

Pour mettre en place cet algorithme, nous cherchons à énumérer les noeuds d'un niveau donné :
au niveau k on doit énumérer les couples (p,q) pour lesquels $p+q=k+1$, sachant que p et q varient tous deux entre 0 et $k-1$. Nous construisons un algorithme en appliquant le traitement séquentiel : nous ordonnons les noeuds d'un niveau sur la base de la relation suivante :

$$(p1,q1) < (p,q) \text{ si et seulement si } \quad p1 < p$$

La file des noeuds de niveau k est dont définie par les primitives :

$$\begin{aligned} &PREMIER_k && : k-1,0 \\ &SUIVANT_k(p,q) && : p-1,q+1 \\ &E\_S(p,q) && : p \neq 0 \text{ et } 2^{p-1}3^{q+1} < x \end{aligned}$$

ce qui peut s'exprimer plus simplement en changeant la manière de caractériser un noeud : à nouveau nous associons à un noeud sa valeur 2^p3^q . Dans ces conditions, la file des noeuds du niveau k est définie par :

$$\begin{aligned} &PREMIER_k && : 2^{k-1} \\ &SUIVANT_k(i) && : 3*i/2 \\ &E\_S_k(i) && : pair(i) \text{ et } 3*i/2 < x \end{aligned}$$

La file des niveaux est définie comme suit :

. un niveau est caractérisé par son premier noeud i

. le premier niveau traité est le premier non vide. Son premier élément a pour valeur 2^p où $p = \lfloor \log_2 x \rfloor$

. étant donné un niveau dont le premier élément est i , le niveau suivant a pour premier élément $i/2$

. le dernier niveau que l'on traite est celui de la racine : $i=1$

c) Algorithme

Finalement nous obtenons l'algorithme :

Algorithme :tri-2-3. Version 2

```
i:=1 ; tantque i < x faire i:=2*i ftantque
itérer i:=i/2
arrêt : i=0
        {traitement du niveau : l'ordre est arbitraire}
        j:=i
        répéter htri(j) ; j:=3*j/2
        jusquà j ≥ x ou impair(j)
        si j < x alors htri(j) fsi
fitérer
```

7.3.5. - Discussion

Ce deuxième exemple nous a permis d'illustrer l'application successive du traitement arborescent et du traitement séquentiel.

. dans un premier temps, le traitement arborescent nous permet d'analyser l'algorithme donné et d'étudier son comportement.

. une deuxième application du traitement arborescent permet d'obtenir un algorithme récursif plus simple.

. L'analyse de ce nouvel algorithme conduit alors à une forme itérative basée sur le "parcours par niveau" de l'arbre

. Finalement l'algorithme final est construit en appliquant le traitement séquentiel à 2 reprises.

Remarque :

Ici encore on peut souligner la différence dans la démarche suivie pour construire cet algorithme final par rapport à ce qui est fait dans l'article d'où nous avons tiré l'exemple [Ber 78].

7.4. - CONCLUSION

L'application du traitement arborescent conduit à un résultat original vis à vis du problème de transformation "récursif-itératif". Ce résultat vient compléter ce qui a été présenté au chapitre 4 est plus significatif étant donné le faible nombre de publications sur le sujet ([PaP 76], [Vei 76], [Bir 77]).

Sur un plan plus général, nous insisterons sur deux points :

. le traitement séquentiel et le traitement arborescent offrent un cadre unificateur dans lequel on peut traiter globalement la transformation "récursif-itératif".

. Ce cadre est parfaitement adapté à un processus de construction de programmes par transformations successives.

CONCLUSION : ASPECTS PRATIQUES ET METHODOLOGIQUES DU TRAITEMENT ARBORESCENT

Comme pour le traitement séquentiel, nous concluons la présentation de la méthode du traitement arborescent en soulignant ses aspects pratiques et en discutant de la démarche méthodologique qui a permis de la définir.

Nous avons décrit un modèle de processus de construction de programmes et nous l'avons illustré tout au long des trois chapitres (5 à 8) notamment lors de l'élaboration des schémas (§ 5.2.), lors de l'étude de la réduction logarithmique (§ 5.3.) et enfin dans le chapitre 7 à propos de la transformation d'algorithmes récursifs en algorithmes itératifs.

Chacune des applications que nous venons de citer constitue en elle-même un résultat intéressant et l'on peut souligner qu'à chaque fois le traitement arborescent est apparu comme un cadre unificateur dans lequel nous avons placé une démarche systématique conduisant à des résultats, connus pour les uns, nouveaux pour les autres.

Le chapitre 6 nous a permis de compléter la présentation du traitement arborescent et d'illustrer l'application du traitement séquentiel. Nous avons ainsi développé une construction systématique originale des algorithmes de parcours d'arbres. Ces algorithmes constituent un outil nécessaire à l'application du traitement arborescent (§ 6.3, § 7.2.3., § 7.3.2., § 7.3.3.)

Sur un plan méthodologique, nous avons confirmé les conclusions de la première partie en retrouvant les lignes directrices dans l'élaboration de la méthode : discrétisation de l'activité de programmation, recours à une structure de donnée abstraite pour définir le cadre de la méthode, définition précise des étapes du processus, et des modèles et des notations associés, élaboration d'une base de schéma.

TROISIEME PARTIE

NOTATIONS ALGORITHMIQUES ET PROGRAMMATION SYSTEMATIQUE

INTRODUCTION

Dans les deux premières parties, nous avons présenté des méthodes d'analyse de problèmes et les techniques permettant de les mettre en oeuvre. La question des supports d'expression algorithmique est un aspect complémentaire de la méthodologie de la programmation que nous abordons dans cette troisième partie.

La multiplicité et la diversité des langages de programmation est un obstacle majeur à la programmation systématique dans la mesure où elle constitue un frein indubitable au développement et à la promotion des méthodes de programmation qui sont proposées à l'heure actuelle. Dans le cas le plus général, même si le programmeur accepte le principe des méthodes, il les dissocie du travail de programmation qui reste toujours régi dans son esprit par la pratique d'un langage et d'un compilateur particuliers, si ce n'est d'une machine particulière. Une analyse plus fine de la situation amène à constater la distance importante existant entre les méthodes que l'on veut pratiquer (qui représentent une concrétisation d'une activité humaine) et les langages disponibles (quel que soit leur modernisme et leur degré d'évolution, ils restent liés à la machine ou du moins à un type de machine dont le fonctionnement reste loin de celui du cerveau humain ...). La question centrale est alors de savoir si l'on peut conserver les avantages d'une pratique systématique de méthodes précises d'analyse des problèmes, malgré cette distance et la multiplicité des langages de programmation.

Nous avons déjà répondu indirectement par l'affirmative à cette question, en montrant dans les chapitres précédents comment l'on peut associer à chaque méthode proposée des techniques de programmation systématique liées à un certain type et un certain niveau de langage. On a

notamment pu voir que d'une part le nombre de ces techniques et leur complexité dépendaient du niveau de langage auquel on se plaçait (récursif versus itératif par exemple), d'autre part que leur présentation dépendait fortement du choix de notations adéquates modélisant les diverses situations pratiques dans lesquelles on peut se trouver (cf. les modèles de "machines séquentielles", § 2.1.2. et de "machines arbres", § 6.1.2.). Sur un plan méthodologique, ceci est un aspect fondamental de notre démarche, et relativement original, car il conditionne la réussite de l'application d'une méthode dans un contexte pratique, aussi bien que la facilité de son enseignement.

Dans cette troisième partie, nous étudions cette question importante, en faisant abstraction des méthodes pratiquées et en regardant plutôt ce dont nous disposons pour communiquer avec l'ordinateur : les langages de programmation. Nous introduisons la notion de notation algorithmique et montrons comment cet intermédiaire entre méthode et langage permet de conserver une attitude systématique dans le processus de programmation. Par ailleurs, nous suggérons que, comme on pouvait s'y attendre, une partie de ce processus systématique peut être automatisé simplement en utilisant des techniques classiques de traduction.

Ces idées sont illustrées par l'exemple de la notation algorithmique que nous utilisons dans notre enseignement de programmation. Nous expliquons la démarche qui a conduit à sa définition et nous proposons des règles de traduction vers des langages de degré d'évolution différents. La mise en forme de cette notation a par ailleurs conduit à un langage expérimental, le langage MEFIA ([Gri 77], [CGS 78] , [SGC 78]) dans lequel ont été concrétisées des idées sur la sécurité et la fiabilité des programmes. La description du langage apparaît en détail dans la thèse de CUNIN ([Cun 79]). Un traducteur de la notation vers PL/1 a été réalisé à Grenoble dans le cadre de projets de DEA ([Dub 78], [Cas 79] , [Mor 79]) : il constitue le noyau d'un ensemble de logiciels expérimentaux d'aide à la programmation systématique que nous projetons de réaliser.

CHAPITRE 8

LANGAGES DE PROGRAMMATION ET NOTATIONS ALGORITHMIQUES

Notre objectif est d'assurer une unité dans la démarche de construction systématique des programmes malgré la diversité des matériels proposés et des logiciels qui les accompagnent. Cette unité nous semble être une clé dans la réussite de la promotion des méthodes d'analyse de problèmes, quelle que soit leur nature. Une brève étude des langages de programmation actuels nous permet de montrer qu'ils ne peuvent pour l'instant répondre à notre souci. Nous introduisons alors la notion de notation algorithmique, en la caractérisant par rapport aux langages de programmation, et en indiquant son rôle dans le processus de construction systématique des programmes.

8.1. - QUELQUES REMARQUES SUR LES LANGAGES DE PROGRAMMATION ACTUELS

Nous observons ce domaine de l'informatique dans l'optique de ce qui précède : les langages de programmation couramment disponibles et utilisés à l'heure actuelle ne peuvent assurer l'unité de la démarche que nous cherchons, car

. d'une part, quel que soit leur degré d'évolution, leur utilisation rend le programmeur encore trop dépendant de l'environnement informatique. Par exemple par le recours à une syntaxe fastidieuse, par la nécessité de préciser trop de détails (non pertinents du point de vue de la logique du problème)...

. d'autre part, une grande partie des langages les plus couramment utilisés n'ont pas été conçus dans une optique privilégiant la méthodologie de la programmation : cela est vrai de langages comme FORTRAN ou COBOL, flagrant pour PL1/. Malgré sa complexité apparente, ALGOL 68 a joué un rôle important du point de vue des méthodologies de conception des langages et de compilation [CSV 76] . On notera à ce propos, l'expérience intéressante

d'enseignement de la programmation utilisant ALGOL 68 comme support [Ger 77] Enfin PASCAL [Wir 73 a] semble à l'heure actuelle être le langage le plus proche d'un certain nombre de nos soucis, ceci par une certaine simplicité dans sa définition et par l'idée même de programmation systématique qui lui est sous jacente, et dont Wirth est l'un des précurseurs [Wir 73 et 76].

De manière générale, on peut faire deux types de critiques aux langages de programmation actuels :

a) Sur le plan conceptuel, il y a encore trop de contraintes dans l'expression

par exemple :

- la manipulation d'ensemble inexistante dans la majeure partie des langages courants
- l'expression de propriétés des objets des programmes.
- ...

En première analyse, on peut certainement expliquer ces contraintes par deux types de raisons : l'insuffisance des constructions linguistiques par rapport aux besoins rencontrés, les limites (de nature essentiellement économique) des techniques de compilation. En deuxième analyse, la critique peut être poussée beaucoup plus loin et conduire à remettre en cause le modèle même sur lequel sont fondés la plupart des langages de programmation à savoir le machine de VON NEUMANN.

Remarque :

L'avenir nous dira si J. BACKUS aura été un précurseur dans son discours de réception à l'ACM (1977 ACM Turing Award, cf. [Bac 78] comme il a pu l'être il y a bientôt trente ans en promouvant un nouvean langage appelé FORTRAN

b) Sur un plan pratique, il y a dans les langages de programmation encore trop de constructions dont les spécifications sont ambigues et dépendantes des machines utilisées. Ceci a pour conséquence de ruiner tout espoir d'assurer la portabilité des programmes par l'utilisation d'un langage unique, de quelque niveau qu'il soit. Une conséquence directe de cette constation est que la diversité des langages existant semble devoir subsister, obstacle majeur à l'unification des comportements des programmeurs vis à vis des environnements informatiques (même si cette diversité des langages peut présenter des avantages du point de vue des possibilités d'expression et de l'adéquation du langage à l'application informatique considérée).

Cette rapide critique des langages de programmation actuels est confirmée par l'examen des orientations actuelles de la recherche dans le domaine des langages de programmation, dont on peut espérer qu'un des buts à long terme est justement d'aboutir à une unification dans la communication entre l'homme et l'ordinateur : (cf. notamment [Schu 75] , et plus récemment [Wor 77] et la réunion du groupe Programmation et Langages de l'AFCET à Cargèse - mai 1979).

. Un premier type de projet (par exemple EUCLID [LHL 77] , MESA [GMS 77], SL5 [GrH 77], SESAME [CCK 76] , concurrent PASCAL [Bri 75] , SETL [KeS 75]...) est fondé sur l'intégration dans un cadre classique de langage, de constructions linguistiques nouvelles permettant d'accroître la puissance d'expression et/ou la fiabilité, et de concrétiser un certain nombre d'idées concernant la méthodologie de la programmation. Parmi ces constructions, on peut citer les plus fréquentes : modules, expression de la synchronisation, formes particulières de contrôle de l'exécution. Le souci de fiabilité est reflété soit par l'adjonction de moyens d'expression d'assertions (EUCLID par exemple), soit plus simplement par le soin particulier apporté au contrôle statique de la cohérence des programmes (système de types en MESA par exemple). Le langage MEFIA que nous avons développé en collaboration avec l'université de Nancy ([Gri 77], [CGS 78], [SGC 78],[Cun 79])est un projet de ce type. Nous présentons dans le chapitre qui suit quelques considérations d'ordre méthodologique qui ont conduit à sa définition.

. Les deux projets CLU [LSA 77] et ALPHARD [WLS 76] que nous séparons du groupe précédent, car ils semblent à la fois être les précurseurs et les plus ambitieux parmi ces projets, ayant été conçus directement comme le complément d'une étude méthodologique particulière : CLU semble plus centré sur les outils d'abstraction [LiZ 74] et sur des considérations pratiques (on notera notamment la notion de "bibliothèque CLU"), ALPHARD étant plus centré sur les problèmes de vérifications.

. Un troisième type de projet (LUCID [AsW 77], FFP Systems [Bac 78], MEDEE [Cas 78], EXEL [Ars 78]...) étudie de nouveaux types d'expression en déplaçant le problème, d'une expression des algorithmes, vers une expression des spécifications. Ce type de projet comporte en général un système intéractif de manipulation et de transformation de programmes ([DaB 76], CIP [Bau 78 EXELMANS [Ars 78]).

Ainsi les langages de programmation nous semblent devoir longtemps rester des outils de programmation plus proches des mécanismes qui définissent le fonctionnement de la machine que de ceux qui définissent celui du cerveau humain. Le troisième type de projet évoqué ci-dessus semble être le plus prometteur, du moins en ce qui concerne la diminution de la distance homme-machine ; mais est-il très réaliste d'envisager l'aboutissement et la diffusion de tels projets à court terme ? Il est clair que non. Dans cette attente, nous disposerons de langages de plus en plus sophistiqués dont les caractéristiques pratiques resteront les suivantes :

. Un langage de programmation présente toujours un caractère relativement général. Le plus souvent, il est conçu dans l'optique d'une utilisation par un très grand nombre personnes dont on ignore à priori les compétences

. Un langage de programmation est figé : ce caractère peut diminuer si l'on dispose de techniques permettant d'étendre le langage (langages extensibles, cf. par exemple [Jor 75], types génériques [Jac 78], types abstraits [GUTTAG 78]).

. La conception, l'implémentation, la diffusion d'un langage de programmation sont des opérations coûteuses en temps et en moyens : on trouve là un facteur fondamental d'inertie dans l'évolution des outils.

8.2. - NOTATIONS ALGORITHMIQUES : INTRODUCTION

C'est dans le contexte très riche en promesses que nous venons d'évoquer qu'il faut chercher aujourd'hui, pour répondre aux besoins immédiats, les moyens d'assurer une unité de démarche dans l'expression algorithmique, afin de conserver tous les bénéfices des méthodes de programmation que nous voulons pratiquer.

Nous proposons une démarche basée sur les deux aspects complémentaires suivants :

a) Le recours à des notations algorithmiques, conçues comme une étape intermédiaire entre les méthodes que l'on veut pratiquer et les langages que l'on doit utiliser. Une telle notation se distingue d'un langage par le fait qu'elle sert à exprimer des algorithmes pour un interlocuteur humain alors que le langage sert à exprimer des algorithmes pour un interlocuteur "machine". Une deuxième caractéristique des notations algorithmiques est qu'elles sont particulièrement adaptées à telle ou telle méthode : elles ont un caractère moins général qu'un langage. Enfin une troisième caractéristique est que contrairement à un langage qui est fixé de manière définitive (ne serait ce que pour pouvoir être implémenté), une notation algorithmique peut évoluer dans le temps ou dans l'espace au gré de l'évolution des méthodes des goûts et des désirs des individus les utilisant.

Une autre manière de souligner la différence entre notation algorithmique et langage (algorithmique) de programmation est d'examiner les conditions de leur utilisation :

- un langage de programmation est forcément attaché à une machine et doit être utilisé par un grand nombre de personnes aux qualifications et compétences différentes. La conception d'un nouveau langage est une opération très délicate dont le coût est très élevé (en temps, en moyens). Il en est de même pour son implémentation et sa diffusion.

- une notation algorithmique est attachée à une personne qui veut l'utiliser dans le contexte de nombreux envisonnements informatiques (machine + langages disponibles sur la machine). A la limite elle peut servir de norme de communication entre un petit nombre de personnes. Sa conception ne dépend que des méthodes de programmation que l'on veut pratiquer.

L'idée de notation algorithmique s'est beaucoup développée dans le contexte de l'enseignement de la programmation. COURTIN et VOIRON ont été précurseurs en ce domaine [CoV 74 a et b]. Nous l'avons nous mêmes présentée comme un élément fondamental pour la conception d'un tel enseignement [LuS76]On retrouve cette idée notamment dans [MeB 78].

b) La systématisation d'une partie du processus de production de programmes

Il s'agit de la partie du processus qui fait passer d'un algorithme décrit dans une notation algorithmique à un programme écrit dans un langage de programmation. Nous cherchons à développer une méthode de "comportement" vis à vis des langages de programmation basée sur les principes suivants :

. la (ou les) notation(s) que pratique un individu représente(nt) son expérience de programmation. Ces notations sont constamment enrichies par les nouvelles expériences de l'individu, éventuellement par l'examen des constructions proposées dans les langages nouveaux. L'interface avec l'environnement informatique dans lequel se trouve cet individu est réalisé en établissant un ensemble de règles de passage de la notation vers tel ou tel langage de cet environnement (ces règles peuvent d'ailleurs être considérées comme les règles que se donne l'individu pour utiliser le langage considéré). Les élaborer est relativement simple.

. La construction d'un programme comporte alors deux étapes : la description d'un algorithme dans une notation algorithmique à l'issue d'un travail de conception et d'analyse, puis la production d'un programme à partir de cet algorithme en appliquant de manière systématique les règles de passage de la notation vers le langage. Cette deuxième étape est éventuellement automatisée, mais alors la notation doit être figée sous forme d'un langage. Nous expérimentons ceci avec le langage MEFIA.

8.3. - NOTATIONS ALGORITHMIQUES : CARACTERES GENERAUX

Nous proposons ici une énumération de caractères généraux des notations algorithmiques. Nous faisons constamment référence ici aux langages de programmation dont les notations algorithmiques doivent écarter un certain nombre de défauts. Nous faisons abstraction des méthodes de programmation que ces notations doivent concrétiser (nous rentrerons dans les détails d'une notation au chapitre suivant). On pourra aisément reconnaître les caractéristiques des nombreuses notations qui sont utilisées soit pour décrire des résultats de recherche, soit pour présenter de manière didactique un ensemble de méthodes et/ou de techniques.

a) Forme externe :

On cherche à éviter un certain nombre de choix syntaxique hérités des langages de programmation qui sont autant d'obstacles à la compréhension de la sémantique des algorithmes et à l'application de règles de programmation systématique.

Nous proposons quelques exemples en référence à plusieurs langages de programmation :

. FORTRAN : faible liberté de mise en page, contraintes sur la taille des identificateurs, étiquettes numériques ...

. PL/1 : forme de l'appel de procédure, ambiguité entre l'opérateur de comparaison et le symbole d'affectation, manipulation de chaînes de caractères.

. de nombreux langages : abus dans l'emploi de séparateurs, le point virgule par exemple ; choix des mots clés

....

Il est évident que tout ceci n'est qu'un ensemble de détails dont les inconvénients peuvent être rapidement dépassés. Il n'empêche que cette accumulation de petits détails joue un rôle important dans le manque de lisibilité que peuvent avoir un certain nombre de notations algorithmiques (et notamment les langages de programmation si on les utilise en tant que notations). Très généralement, ce critère de lisibilité est lié à une expression la plus directe possible des concepts manipulés. Le degré de concision dans le formalisme peut varier d'une utilisation à l'autre (moins de concision dans une présentation didactique, plus de concision dans une utilisation répétée pour un problème précis). Enfin, on peut attacher une certaine importance au choix des mots qui pourront servir à une expression orale des algorithmes.

b) Sémantique

Clairement il faut que les primitives apparaissant dans la notation algorithmique reflètent les concepts qui sont à la base des méthodes pratiquées. Au niveau d'abstraction où nous nous plaçons il est difficile de donner des indications précises sur ce point. A titre d'exemple on peut comparer les notations utilisées dans des ouvrages tels que [AHU 74]("pidgin ALGOL") ou dans des articles décrivant des techniques de manipulation de programmes tels que [BuD 77].

Remarque :
Dans [Lus 76] nous donnons quelques indications supplémentaires concernant des notations utilisées pour l'enseignement de l'algorithmique

c) Aspect évolutif

Clairement une notation algorithmique n'est pas figée. La notation peut évoluer dans plusieurs types de circonstances :

- évolution du formalisme utilisé lors de l'analyse d'un problème. La notation sert aussi bien à décrire l'état d'analyse à un certain moment, qu'à décrire l'algorithme final. Plus on va vers cette forme finale, plus la notation devient rigoureuse. En aucun cas la forme séquentielle du texte décrivant l'algorithme final ne doit influer sur la démarche d'analyse (utilisation de dessins lors de l'analyse, ordre dans lequel sont abordés les problèmes, mise en place séparée des objets apparaissant progressivement lors de l'analyse).

- évolution de la notation suivant l'interlocuteur auquel est destiné l'algorithme. Nous avons déjà signalé une telle évolution en ce qui concerne la forme externe (variations dans la concision). Les variations peuvent aussi toucher les aspects sémantiques de la notation, notamment en fonction du degré de compétence (vis à vis du problème traité) de l'interlocuteur.

- évolution de la notation en fonction des problèmes traités, des méthodes appliquées (à titre d'exemple, on peut se référer au § 3.2. dans l'exposé du traitement séquentiel, où nous avons présenté une adaptation des notations générales de machines séquentielles, au cas du traitement de fichiers séquentiels).

8.4. - CONCLUSION

Une conséquence importante de l'évolution récente de la programmation est la remise en cause du rôle des langages de programmation dans la maîtrise du processus de construction de programmes (ce qui apparaît nettement dans les méthodes d'enseignement de la programmation). Nous avons critiqué les langages de programmation en ce sens et nous avons constaté que les projets actuels dans ce domaine confirment notre critique.

Nous sommes toutefois obligés de reconnaître que les moyens d'expression algorithmique jouent un rôle fondamental dans la pratique et la transmission des méthodes d'analyse des problèmes et des techniques de programmation systématique. C'est pourquoi nous cherchons à développer des notations algorithmiques qui d'une part reflètent les éléments fondamentaux des méthodes que l'on veut pratiquer et d'autre part restent proches des langages que l'on doit utiliser. A terme, ces notations doivent jouer un rôle essentiel dans le développement de l'idée de programmation systématique.

CHAPITRE 9

UN EXEMPLE DE NOTATION ALGORITHMIQUE

L'objectif de ce chapitre est d'illustrer sur un exemple la démarche que l'on peut suivre pour choisir les éléments d'une notation algorithmique. Pour cela nous reprenons les points principaux de la notation que nous avons utilisée tout au long de cette thèse, et qui par ailleurs nous sert dans notre enseignement de l'algorithmique.

Nous voulons résumer et refléter les méthodes de programmation que nous désirons pratiquer et nous voulons aussi tenir compte de l'état de l'art dans le domaine des langages de programmation (pour des raisons essentiellement pragmatiques, notamment dans un contexte didactique). C'est pourquoi nous restons à un niveau de formalisme dans lequel subsistent les notions de variable et d'affectation. A partir de ce choix initial, l'effort de constitution de la notation est guidé, outre les considérations relatives à la forme externe (souci de se libérer de certaines contraintes syntaxiques), par le désir de réunir un petit nombre de constructions linguistiques concrétisant les concepts fondamentaux de l'algorithmique.

Nous n'insistons pas sur les aspects syntaxiques de la notation : on reste très proche des langages de la famille ALGOL. On cherche à utiliser l'espace par des règles de présentation (notamment règles d'indentation, place des commentaires,...) que nous nous abstenons toutefois de donner ici (elles apparaissent d'elles-mêmes dans les exemples que nous avons donnés au cours de la thèse). Notons qu'on peut utiliser des conventions typographiques pour augmenter la richesse de la notation : par exemple, nous nous efforçons de réserver les majuscules pour les noms d'objets qui ne sont pas définis au même endroit que le texte en cours.

Dans ce qui suit, nous présentons les éléments de la notation (dont une autre présentation apparaît dans [Mor 79]) et les règles de traduction vers PL/1 (vis à vis duquel la notation joue plutôt le rôle d'un filtre

méthodologique) et vers BASIC (dont la notation est plutôt une extension). Le PL/1 employé est la version disponible sur l'IBM/360 et le BASIC est la version disponible sur la WANG 2200 . L'ensemble de ces règles de traduction peut être trouvé de manière plus détaillée dans [LuS 75] (vol. 2, "fiches PL/1") et dans [Ire 79] ("fiches BASIC").

9.1 - ABSTRACTION

Un élément fondamental de la programmation, notamment mis en valeur par les techniques d'analyse descendante, est le recours constant à l'abstraction. Cette abstraction peut porter sur les algorithmes, on introduit pour cela la notion d'*"action nommée"*, ou sur les informations, on introduit pour cela la notion d'*"information nommée"*. Les *"univers"* permettent d'exprimer les dépendances entre ces abstractions.

9.1.1. - Action nommée

a) Concepts

On fait abstraction de la manière dont un algorithme est réalisé et on ne retient que les spécifications de ses effets. L'abstraction est faite en utilisant un nom à la place de la description de l'algorithme, là où l'on veut faire référence à ses effets.

Nous utilisons le terme général d'*"action nommée"* (suivant en cela la terminologie proposée dans [Dij 71]), pour désigner ce type d'abstraction. Les règles de notation relatives à ce type d'abstraction sont guidées par deux considérations :

1 - La nature de l'effet :

. L'effet de l'algorithme est entièrement caractérisé par une modification d'un état (formalisé par un ensemble de variables) : nous conservons le terme d'*"action"*.

. L'effet de l'algorithme est entièrement caractérisé par une valeur ou un ensemble de valeurs (que l'algorithme calcule) : nous utilisons le terme de *"fonction"*. La référence à une fonction ne peut se situer que dans une expression (la fonction permet de faire abstraction de la valeur qu'elle calcule).

Remarque :
Nous utilisons aussi une forme intermédiaire d'action dont l'effet est caractérisé par une modification d'état, mais qui calcule aussi une valeur. Nous utilisons alors le terme d'*"action à résultat"*

2 - Le degré de dépendance de la description de l'action nommée vis à vis d'un contexte particulier.
Dans sa forme la plus primitive, une action nommée est entièrement dépendante du contexte dans lequel elle est utilisée : son effet dépend entièrement des variables de ce contexte qui apparaissent donc dans la description de l'action nommée.
Une forme plus élaborée d'action nommée permet de libérer sa description du contexte dans lequel elle est utilisée : on introduit la notion de *"paramètre"* et les notations associées. Dans la description de l'action nommée, on peut faire référence à des paramètres formels (qui correspondent aux variables libres de la logique ou aux paramètres muets en mathématiques), qui sont autant d'abstractions d'objets effectifs. La liste de ces paramètres formels apparaît au début de la description de l'action. Lors de l'utilisation d'une telle action nommée (que nous qualifions de *"paramétrée"*), on doit préciser quels sont les objets du contexte d'utilisation qui correspondent effectivement aux paramètres formels de la description : l'utilisation du nom de l'action nommée est accompagnée de la liste de ces objets appelés *paramètres "effectifs"* (mécanisme de liaison entre le contexte d'utilisation et le contexte de description).

Soulignons que nous interdisons de faire apparaître parmi les paramètres d'une action nommée, l'un des objets qui caractérisent l'effet de l'action, c'est-à-dire l'un des objets modifiés par l'action. En d'autres termes, un paramètre ne peut jamais être modifié par une action. Cette interdiction sévère est justifiée par des raisons d'ordre méthodologique (il paraît peu logique de faire abstraction d'une partie de l'effet d'une action, i.e. l'objet qu'elle modifie) et aussi par des raisons

de sécurité (notamment la possibilité de modifier les paramètres est à l'origine de nombreux "effets de bord", cf. [CGS 78].

b) Notations

. actions :

action nom (liste des paramètres formels)
{commentaire indiquant . l'effet de l'action
. le domaine de validité des paramètre
description de l'action
faction nom

. fonctions :

fonction nom (liste des paramètres formels) → type de la valeur calculée
{commentaire indiquant . la nature de la valeur calculée
. le domaine de validité des paramèt
description de la fonction
ffonction nom

. actions à résultat :

C'est une combinaison des deux constructions précédentes.

Les mots clés sont *acres* et *facres*.

Remarques :
. pour plus de détails sur les notations cf. [Mor 79].
. les fonctions et actions à résultat peuvent délivrer des n-uplets de valeurs.

c) Traduction

c.1 en PL/1

On utilise le mécanisme de procédure PL/1 avec un certain nombre de règles supplémentaires : notamment, l'obligation de déclarer tous les noms de procédure par une instruction *DECLARE* (afin d'éviter de nombreux problèmes dus au système de conversion automatique et à l'attribution automatique des attributs de précision).

Remarque :
Les n-uplets résultats sont traduits sous forme de paramètres résultats, mis à jour au moment de quitter la procédure.

c.2 en BASIC :

On utilise le mécanisme de sous-programme de BASIC (instruction *GOSUB*') : la notion de paramètre est partiellement restituée (les "paramètres" BASIC ne sont pas formels : ils jouent plus un rôle de convention de liaison qu'un rôle de paramètre permettant de réaliser une certaine abstraction). La transmission des valeurs calculées par une fonction est réalisée à l'aide d'une variable attachée à la fonction et réservée à cet effet. (on peut aussi utiliser un mécanisme général de pile).

d) Remarques :

La traduction d'une action nommée lors du passage d'un algorithme à un programme dans un langage donné, ne passe pas nécessairement par les constructions linguistiques directement associées au concept d'action, à savoir le mécanisme général de procédure ou de sous-programme. Citons plusieurs variantes :

. dans les langages à structure de bloc, on peut remplacer l'action par un bloc, plutôt que par une procédure : le remplacement des paramètres formels par les paramètres effectifs est alors fait au moment de la traduction. La structure de bloc permet de garder le bénéfice de la localité pour la description de l'action (dans notre terminologie, un tel bloc est une *"action anonyme"*).Le nom de l'action et ses paramètres peuvent être rappelés au niveau d'un commentaire.

. de manière plus générale, l'action peut tout simplement être traduite par la séquence d'instructions correspondant à sa description. Là encore, il faut effectuer la liaison paramètres formels-paramètres effectifs, au moment de cette traduction. Il faut de plus assurer la gestion des variables locales à l'action (éventuellement en rebaptisant tous les noms locaux). Cette technique peut être préférée soit pour des raisons de concision du programme obtenu, soit pour des raisons d'efficacité.

9.1.2. - Information nommée

a) Concepts

On fait abstraction de la "valeur" d'une information et de la manière dont elle est formalisée et représentée. On ne retient que les caractéristiques de l'information, ce que l'on appelle le *"type"* de l'information. On trouvera une approche systématique de la notion de type dans [Hoa 72] reprise en partie dans [Lus 75].

L'abstraction est faite en utilisant un nom à la place de l'information là où l'on veut la faire intervenir dans un algorithme.

a.1 - Le traitement d'une information se ramène à deux types élémentaires de traitements : la *consultation* de l'information (totale ou partielle) et la *modification* de l'information (totale ou partielle). Le premier type de traitement est exprimé par l'utilisation du nom de l'information à l'intérieur d'une expression, le deuxième type de traitement est exprimé par l'apparition du nom de l'information en partie gauche d'une instruction d'affectation.

Remarque :

Nous voulons souligner ici la complexité de la notion de variable telle qu'elle apparaît couramment dans les langages de programmation. Nous pensons que cette complexité est due au fait que la même construction linguistique, l'utilisation d'"identificateurs de variables", est le carrefour de plusieurs concepts très différents : sur un plan algorithmique, information et représentation de l'information, sur un plan technique, allocation mémoire et sécurité (vérification de cohérence en fonction des types des objets manipulés). Le terme "variable" semble abusif et du moins ambigu, dans la mesure ou l'on doit distinguer deux types de variations : la variation du contenu d'un (ou d'un groupe de) emplacement mémoire, et l'évolution d'une information dans le temps au cours du traitement décrit par l'algorithme étudié. C'est le premier sens qui a longtemps prévalu, du à une assimilation entre l'identificateur et l'emplacement mémoire lui correspondant (le même emplacement mémoire pouvant d'ailleurs servir dans le temps à représenter plusieurs informations bien distinctes au niveau algorithmique : ceci a longtemps été la base de la programmation dite "astucieuse"). A l'heure actuelle, on a tendance à revenir sur ce sens et l'on préfère associer à un nom d'identificateur l'information qu'il représente. Dans ces conditions la notion de variable change et devient plus particulièrement associée à la notion d'itération. On trouvera un exemple d'illustration de ces idées dans [Ger 77].

a.2 - La description proprement dite de l'information est réalisée à l'aide d'une construction linguistique particulière appelée *"déclaration"*, qui permet d'associer un nom et un type d'information. Cette description doit être mise en parallèle avec la description d'une action que nous avons évoquée au paragraphe précédent.

Les formalismes algorithmiques actuels sont peu avancés du point de vue de la description des informations et de leurs propriétés : en effet, la description d'une information dans un langage classique se réduit à la description de sa représentation au niveau machine (en

termes de types standards et de constructeurs), ce qui reste un niveau très bas (c'est-à-dire très dépendant des machines). Nous pensons que le maximum d'effort doit être porté dans ce domaine afin de faire progresser les notations algorithmiques. Ce travail doit tirer parti des recherches effectuées sur les types abstraits ([LiZ 74],[GHM 78], [Jac 78]). On trouvera une présentation didactique des problèmes de spécification formelle dans [Liv 78]).
Mais on peut aussi s'inspirer de travaux bien antérieurs tels que ceux d'IVERSON [Ive 62] (qui ont conduit à la définition du langage APL) ou des travaux plus récents qui ont conduit à la définition du langage SETL [KeS 75].

Nous travaillons dans ce domaine et avons obtenu quelques résultats en ce qui concerne l'expression de propriétés. CUNIN développe dans [Cun 79] les problèmes posés par l'implémentation des quelques constructions linguistiques que nous avons introduites dans le langage MEFIA. Mais ces constructions nécessitent d'être soumises à de nombreuses expérimentations avant que l'on puisse tirer des conclusions sur un plan méthodologique et les répercuter au sein d'une notation algorithmique.

b) Notations

b.1 - types standards

Nous utilisons les quatre types standards suivants :

entier, réel, logique, caractère

à chaqun de ces types est associée la convention habituelle de notation de constantes.

b.2 - déclaration

Une déclaration a la forme suivante :

<description><nom>

la description est réalisée en combinant l'utilisation de types standards, de types nommés et de constructeurs.

b.3 - Constructeurs

Nous utilisons un constructeur de *structure* et un constructeur de *tableau*.

. le constructeur de structure : l'objet que l'on déclare est un ensemble non ordonné de champs. Chaque champ est désigné par un nom de champ.

Une description utilisant un constructeur de structure a la forme :

structure (<liste de champs>) {le séparateur est la virgule

Dans la liste de champs, chaque champ comporte un nom et une description :

<nom de champ>:<description>

Pour désigner un champ d'un objet structuré, on utilise une "notation qualifiée" où apparaissent le nom de l'objet et le nom du champ.

. le constructeur de tableau : l'objet que l'on déclare est un ensemble ordonné d'éléments de même type.

La description a la forme :

tableau (<dimension>)<description>

où l'on fait apparaître la dimension du tableau et la description du type commun à tous les éléments du tableau (qui peut évidemment utiliser un constructeur de tableau : on définit ainsi des tableaux à plusieurs dimensions).

La dimension du tableau peut être indiquée de manière classique en donnant une borne inférieure et une borne supérieure, ou bien en donnant un nom d'intervalle (cf. b.4 ci-dessous).

Pour désigner un élément d'un tableau, on utilise une notation indicée : le nom du tableau est suivi d'un "indice", entier déclaré comme appartenant à l'intervalle de définition du tableau.

b.4. - Intervalles

Un intervalle correspond à un intervalle fermé sur l'ensemble des entiers relatifs. On introduit un nom d'intervalle par une déclaration dont la forme est :

intervalle<nom d'intervalle> *est* <borne inférieure>:<borne supérieure>

Les intervalles permettent d'une part de dissocier la description d'un tableau de la définition de ses bornes et éventuellement de relier plusieurs tableaux définis avec la même dimension, d'autre part d'attacher les entiers qui vont servir à indexer les tableaux à leurs intervalles de définition par une déclaration d'indice :

indice (<nom d'intervalle>)<nom de l'indice>

b.5 - type nommé

On peut donner un nom à une description d'objet, ce nom peut alors être utilisé partout où l'on attend une description. On a ainsi un "*type nommé*" dont la définition a la forme :

type<nom de type> *est*<description>

Les types nommés permettent de dissocier la description d'un objet de sa déclaration. C'est une première étape dans notre recherche de constructions linguistiques permettant d'aider la manipulation d'abstractions relatives aux informations traitées par un programme.

Remarque :
Une forme plus élaborée de définition de type permet d'introduire des paramètres formels correspondants aux dimensions des tableaux intervenant dans la description du type. Ce mécanisme permet notamment d'exprimer des relations entre des objets déclarés avec des types différents (cf. [Cun 79]).

c) Traduction

c.1 en PL/1

. on utilise le mécanisme de déclaration d'attributs de PL/1, et les constructeurs de tableau et de structure que ce langage comporte.

. la notion de type nommé disparaît. Tout au plus on peut faire apparaître les noms de ces types en commentaire.

Remarque :
on peut en fait restituer la notion de type nommé en utilisant la construction *LIKE* de PL/1.

. la notion d'intervalle et d'indice disparaît. On peut utiliser le mécanisme de *ON CONDITION* pour forcer la vérification d'appartenance des indices à l'intervalle de définition des tableaux.

c.2 - en BASIC

Les possibilités sont faibles, du fait

. de la limitation sur le nombre de lettres des identificateurs. On peut gérer manuellement une table de correspondance entre les noms logiques apparaissant dans l'algorithme et les identificateurs du programme.

. de l'inexistence du constructeur de structure :
un objet structuré sera traduit par une collection d'objets simples.

Remarque :
Bien que BASIC ne comporte pas de construction de "déclaration" nous suggérons de restituer ces déclarations sous forme de commentaires.

9.1.3. - Dualité algorithmes-informations : notion d'univers

a) Concept

Ce qui précède a montré les liens étroits existant entre les deux mécanismes d'abstraction que nous avons présentés :

. l'effet d'une action est entièrement spécifié par les modifications apportées à une (ou un groupe d') information elle-même décrite en termes de variables. Cet effet peut de plus dépendre de l'état d'informations appartenant au contexte dans lequel l'action est utilisée.
Dans le cas des fonctions (qui ne modifient pas d'information) la valeur calculée peut aussi dépendre de l'état d'informations du contexte d'utilisation.

. réciproquement, la description d'une information et notamment sa représentation dépendent beaucoup des traitements que cette information subit au cours de l'exécution de l'algorithme.

Cette dualité bien connue entre les algorithmes d'une part et les informations d'autre part apparaît très nettement au cours de l'analyse d'un problème : on se trouve souvent dans une situation où les décisions relatives à la description d'une action dépendent de choix faits à propos de la représentation d'une information et vice-versa. Il est donc nécessaire de regrouper les abstractions (d'algorithmes et d'informations) avant de poursuivre l'analyse du problème.

La notation algorithmique fournit avec les *"univers"* un mécanisme de regroupement d'abstraction. Un univers regroupe toutes les actions et fonctions spécifiées en termes d'un ensemble d'informations, et ces informations. Ce mécanisme est dans l'immédiat très proche de la notion de *"module"* développée notamment dans [Par 72 a et b]. (cf. aussi la notion de "machine abstraite" développée dans [Gal 77] et [GaM 78]).

Un deuxième aspect de la notion d'univers apparaît si l'on se place du point de vue de l'algorithme que l'on obtient à l'issue d'une analyse : l'ensemble des abstractions que cette analyse a fait apparaître se trouve codifié dans l'algorithpe d'une manière ou d'une autre. La description des actions et fonctions est faite en termes des variables qui décrivent les informations. Les univers permettent de regrouper dans une même région du texte du programme les variables et les instructions qui le modifient. Ce mécanisme permet ainsi d'assurer une certaine protection des variables : en dehors de l'univers auquel elle appartient, on ne peut utiliser le nom d'une variable, que pour la consulter. Ces variables ne peuvent être modifiées à l'extérieur de leurs univers, qu'indirectement par l'intermédiaire des actions de l'univers qui les modifient.

Enfin, un troisième aspect de la notion d'univers apparaît si l'on considère un univers indépendamment du contexte de son utilisation : il regroupe des actions nommées et des informations nommées interdépendantes. On peut alors considérer l'univers comme étant lui même une abstraction : on fait abstraction de l'information qu'il comporte et on ne retient que les primitives qui permettent de la consulter ou de la modifier. Nous aboutissons ainsi sur le concept de type abstrait.

En résumé, la notion d'univers permet d'exprimer simultanément trois types de démarches qui interviennent à des moments différents dans le processus de production de programmes :

. le regroupement d'abstractions au cours d'une analyse
. la protection de variables au cours du codage d'un programme
. la mise en évidence d'un nouveau type d'information.

Au niveau de notre notation, nous restons pour l'instant encore très prudents : nous utilisons une forme très primitive du mécanisme qui ne comporte aucun moyen de paramétrer les univers (cette question est néanmoins étudiée dans [Cun 79] au travers des *"méta-modules"* de MEFIA). Par ailleurs nous ne prévoyons pour l'instant aucune imbrication d'univers.

b) Notation :

Un univers est encadré par les mots clés *univers* et *funivers* éventuellement suivis du nom choisi pour l'univers. Il regroupe :

- un ensemble de déclarations de variables et constantes (dites de l'univers) caractérisant l'information de l'univers. Les noms de ces objets ne peuvent être utilisés à l'extérieur de l'univers qu'en lecture (partie droite d'une affectation et dans la mesure où leur déclaration est préfixée par le mot clé *"accés"*
- un ensemble d'actions et de fonctions, les primitives de l'univers, qui permettent de manipuler l'information de l'univers, en dehors de l'univers.

Remarque :
Un programme est constitué d'un "algorithme" rédigé en termes des primitives d'un ensemble d'univers (le "répertoire" du programme).

c) Traduction

c.1 - traduction en PL/1

Elle peut être basée sur l'utilisation de blocs procédures : à chaque univers correspond une procédure. Les primitives de l'univers sont autant de points d'entrée dans cette procédure. Les variables de l'univers sont déclarées avec l'attribut STATIC ce qui permet de leur donner une durée de vie égale à celle du programme. Lorsqu'une variable de l'univers fait intervenir un élément dynamique (dimension de tableau ou longueur de chaîne dépendant d'une variable), la variable est déclarée avec l'attribut CONTROLLED, ce qui permet de gérer son allocation à l'exécution au moment où l'élément dynamique de la déclaration est connu. On trouvera les détails de cette technique et une discussion sur d'autres méthodes possibles dans [Dub 78].

c.2 - traduction en BASIC

Cette traduction est basée sur un simple regroupement des instructions.

9.1.4. - Discussion

Les mécanismes d'abstraction que nous utilisons dans notre notation correspondent à des notions maintenant classiques.

En ce qui concerne les actions, nous avons fait un effort de classification lié à la manière de définir l'effet des algorithmes dont on veut faire abstraction. On arrive ainsi à une définition précise des actions et des fonctions dont l'intérêt s'est avéré particulièrement net dans un contexte d'enseignement : la notion de fonction et son utilisation a toujours été un point délicat auprès d'étudiants n'ayant pas une formation scientifique poussée. Cette difficulté est d'ailleurs révélatrice du mal qu'ils ont à appréhender les mécanismes d'abstraction sous-jacents au mécanisme de procédure, trop souvent présenté par ses seuls aspects techniques : outil permettant de n'écrire qu'une fois un ensemble d'instructions (l'effort de présentation étant axé sur les aspects du mécanisme de passage de paramètres et sur la puissance que les divers modes de passage permet de donner). Notre travail de réflexion sur l'enseignement de la programmation s'est efforcé dès le départ (1973) de résoudre cette question. C'est ainsi que nous avons mis en place une initiation à la programmation incluant les procédures (sous la forme du concept d'action) dès les premières séances (cf. [LMS 75]).

La restriction que nous imposons sur les paramètres (impossibilité de les modifier) est justifiée dans la mesure où elle va dans un sens de clarification des concepts. Elle est contraignante par rapport aux habitudes, mais l'expérience montre que les cas où cette restriction est difficile à dépasser correspondent à des situations où l'on a en fait besoin d'un mécanisme de type abstrait (plutôt que d'enfreindre les règles).

Comme nous l'avons dit, nous pensons que le principal effort d'imagination doit porter sur la recherche de mécanismes de description des informations. Les types nommés et les univers indiquent deux directions dans lesquelles nous travaillons. Là aussi, nous pensons avoir fait un pas dans le sens d'une clarification des concepts, en proposant des mécanismes

permettant d'indiquer clairement les abstractions qui sont manipulées et leurs relations, et en rattachant leur présentation à des concepts méthodologiques plus qu'à des aspects techniques liés au processus de compilation.

9.2 - REDUCTION DES PROBLEMES - COMPOSITION DES ACTIONS

Une technique générale d'analyse des problèmes consiste à "réduire" la complexité d'un problème en le ramenant à des sous-problèmes plus simples en termes desquels l'on peut résoudre le problème initial. Dans un deuxième temps, on construit un algorithme en "composant" les sous-problèmes à l'aide de "schémas de composition". Cette technique devenue relativement classique à l'heure actuelle est à la base de toutes les méthodes de programmation actuelle (on peut en trouver les premières présentations significatives dans [Wir 71 b] et dans [Dij 72]). Soulignons qu'elle est intrinsèquement liée à un souci de certification progressive de la correction des programmes et à un désir d'axiomatisation des primitives de programmation [Hoa 69] : l'importance des schémas de composition de base (séquence, choix, itération) est liée à l'axiomatisation qui a pu leur être associée. Nous passons sous silence ici tous les aspects des techniques de preuve progressive des programmes renvoyant le lecteur aux ouvrages didactiques présentant le sujet (par exemple [Wir 73] ou plus récemment [MaW 77], [Bol 77] [Dah 77], [Liv 78]).

Nous présentons ici les différents schémas de composition en les rattachant à des méthodes de réduction de problèmes. Pour chaque schéma nous donnons une forme de base, éventuellement des formes particulières. Puis nous discutons de leur traduction en BASIC et en PL/1.

9.2.1. - Processus et composition simple

Le schéma de composition le plus simple est la "séquence". Il correspond à la notion même de processus, c'est-à-dire à la discrétisation dans le temps d'un évènement. Le point virgule est le symbole consacré par ALGOL 60 pour indiquer la séquence (ce n'est pas toujours le cas. Ainsi en BASIC, le "deux point" joue ce rôle).

Un autre schéma de composition simple (plus récent) est la composition collatérale (cf. ALGOL 68 et [Ger 77]). Nous la dénotons à l'aide de la virgule. Elle revêt une certaine importance pour nous dans la mesure où nous cherchons à rester, au niveau de la description algorithmique, le plus près possible de la logique du problème traité : nous voulons pouvoir ne pas imposer d'ordre entre l'exécution de deux actions, si la logique ne l'impose pas elle-même.

Remarque :
Le point-virgule joue un rôle supplémentaire de séparateur syntaxique dans les langages de programmation (utile pour la description de la grammaire et pour la récupération d'erreurs syntaxiques), parfois à un point tel qu'il perd totalement son rôle de composition séquentielle. Au niveau de notre notation algorithmique, nous pouvons diminuer l'utilisation des séparateurs dans la mesure où la lecture par un humain est beaucoup plus globale que l'analyse d'un compilateur. C'est ainsi que lorsqu'il n'y a aucune ambiguité possible, nous omettons le point virgule (notamment lorsque c'est le dernier caractère significatif d'une ligne, ou qu'il est directement suivi d'un autre séparateur syntaxique). (dans un contexte pédagogique, cette démarche peut être contestable, du moins lors de l'initiation, car elle risque de provoquer un grand nombre d'erreurs syntaxiques lors des premiers passages de programmes).

9.2.2. - Analyse par cas et schéma de choix

a) concepts

L'analyse par cas est une technique simple de réduction des problèmes qui consiste à réaliser une partition sur l'ensemble des cas possibles et à associer à chacun des sous-ensembles ainsi obtenu, un sous-problème particulier. Les points importants dans cette démarche sont les suivants :

. on doit effectivement examiner tous les cas possibles

. on doit effectivement réaliser une partition sur cet ensemble de cas.

Le deuxième point conditionne le déterminisme des algorithmes que l'on écrit. Là encore on peut (toujours dans le souci de ne pas exprimer dans l'algorithme plus que la logique du problème), relacher cette contrainte et admettre qu'un cas soit rattaché à plusieurs sous-problèmes (ceci est largement discuté dans [Dij 76], où il est montré notamment que dans certains cas le non déterminisme facilite la preuve des programmes, si ce n'est leur construction). Dans l'absence de réflexion personnelle très

approfondie sur le non déterminisme et en attendant une concrétisation plus nette des recherches entreprises dans ce domaine, nous essayons en général de conserver le déterminisme de nos constructions.

b) notations

b.1 - forme générale

```
choix
   {commentaire : description du domaine de référence
    qui conditionne l'examen des cas}
    e.l.1. : action_1
    e.l.2. : action_2
    ---
    e.l.n. : action_n
fchoix
```

. Les e.l.i. sont des expressions logiques qui formalisent la partition réalisée : tout le domaine de référence doit être couvert, et les expressions logiques doivent s'exclure mutuellement.

. les action_i sont les actions associées à chacun des groupes de cas possibles.

b.2 - formes particulières :

Nous admettons trois formes particulières de cette construction :

- la première permet d'indiquer lorsqu'on le désire un ordre dans l'examen des cas qui contrôlent le choix : cet ordre est celui dans lequel ils apparaissent dans la construction. Le mot clef *choix* est alors suivi du qualificatif *séquentiel* . On admet dans cette construction de remplacer la dernière expression logique par le mot clé *autrement*.

- la deuxième forme est l'action conditionnelle classique :

```
si e.l.
  alors action_1
  sinon action_2
fsi
```

qui est une "séquentialisation" de :

```
choix
    e.l. : action_1
   ¬e.l. : action_2
fchoix
```

- enfin la troisième forme est un cas particulier de la seconde dans laquelle l'action_2 est vide (plus que d'un choix, il s'agit du repérage d'une condition exceptionnelle) :

```
si e.l. alors
   action_1
fsi
```

c) traduction

c.1 - traduction en PL/1

Les deux dernières formes particulières(*si...alors...sinon...* et *si...alors...fsi*) sont traduites directement par les instructions équivalentes de PL/1.
En ce qui concerne la forme générale, on peut la séquentialiser soit à l'aide d'une succession de *si...alors...sinon...* imbriqués, soit à l'aide de l'instruction de branchement.

c.2 - traduction en BASIC

Seul le *si...alors...fsi* a son équivalent direct. Les autres constructions sont séquentialisées en termes du *si...alors...fsi* : le schéma général de choix devient par exemple :

```
      rem choix
      if ¬ e.l.1 then et_2
      action_1 : goto et_fchoix
et_2  if ¬e.l.2 then et_3
      action_2 : goto et_fchoix
et_3  if ¬e.l.3 then et_4
      ---
et_n  action_n
et_fchoix rem fchoix
```

où *et_2*, *et_3*,..., *et_n* , *et_fchoix* sont des étiquettes numériques adéquates.

Remarque :
On trouvera dans [Cun 79] une étude des problèmes d'implémentation du schéma *choix* , relatifs à la vérification de la validité et de la cohérence des expressions logiques.

9.2.3. Traitement_séquentiel_et_itération

a) Forme générale

La première partie de cette thèse a montré comment la méthode de traitement séquentiel est attachée au schéma d'itération. Nous ne précisons ici qu'un certain nombre d'éléments de notation.

La forme principale de schéma d'itération que nous considérons est le schéma *itérer* présenté notamment dans [Knu 74] en relation avec un certain nombre de problèmes d'efficacité posés par l'élimination par trop systématique du *GOTO*.

```
itérer
  action_1
arrêt : E.1.
  action_2
fitérer
```

b) Formes particulières

Ce schéma admet deux formes particulières qui correspondent aux cas où l'une des deux actions est vide : si l'action_1 est vide on obtient le schéma *tantque* , si l'action_2 est vide on obtient le schéma *répéter* . Nous utilisons ces deux formes particulières afin de mettre en évidence le fait que, contrairement au cas du schéma général *itérer* , tout ce qui est répété l'est le même nombre de fois. Le schéma *répéter* a de plus pour propriété que l'action interne est répétée au moins une fois.

formes particulières d'itérations	
tantque ¬e.1. *faire* action_2 *ftantque*	*répéter* action_1 *jusqua* e.1.

Nous avons déjà discuté dans la première partie (§ 2.2.2.) du schéma de contrôle *pour* , indiquant que nous réservions son usage au cas où tout le contrôle de l'itération, i.e. toute la définition de la file caractéristique du traitement séquentiel, était entièrement exprimé dans le *pour*.

c) Traduction

c.1 traduction en PL/1

le *tantque* et le *pour* ont leur équivalent en PL/1. La traduction des schémas *itérer* et *répéter* peut se faire soit en utilisant le *tantque* (ce qui entraîne une duplication de l'action_1), soit en utilisant l'instruction de branchement.

c.2 traduction en BASIC

On utilise l'instruction *si..alors...* :

```
et_itérer rem itérer
          action_1
          si e.l. alors et_fitérer
          action_2 : goto et_itérer
et_fitérer rem fitérer
```

d) Discussion

L'étude des formes d'itération a connu un essor particulier au moment de la controverse sur l'instruction de branchement [Lea 72]. Dans un premier temps, on s'est accordé à ne retenir qu'une forme d'itération, le *tantque* (et/ou la forme voisine *répéter*) : ceci allait dans le sens de la clarification demandée par le courant de programmation structurée (limitation du nombre de schémas de base, effort d'axiomatisation). Parallèlement à la critique du *GOTO* , apparaissait d'ailleurs une critique de certaines formes d'itération (en FORTRAN par exemple). La forme de *pour* du langage ALGOL W (par rapport à ALGOL 60) est un signe de cette évolution.

L'expérimentation d'une utilisation systématique des principes de la programmation structurée (trop souvent confondue avec une recherche de l'élimination systématique des *GOTO*) a conduit à certaines réserves, notamment du point de vue de l'efficacité [Knu 74] . La tendance a alors été de trouver de meilleures (plus pratiques) formes d'itération, tout en étant extrêmement prudent (ne serait-ce que par l'aspect exemplaire que peut prendre toute discussion sur les schémas de contrôle) : cette prudence se manifeste notamment par un souci constant d'apporter en même temps qu'une forme de schéma d'itération, les éléments nécessaires à la preuve (de terminaison et de correction) des algorithmes l'utilisant (cf. la synthèse proposée dans [LeM 75]). Outre le schéma *itérer* que nous avons présenté et qui est largement accepté à l'heure actuelle sous diverses formes syntaxiques, plusieurs types de constructions ont été

proposées : nous en énumérons quelques unes ici (renvoyant le lecteur à la bibliographie, pour en découvrir les détails), en essayant de les classifier en fonction de la motivation qui semble être à leur origine.

Une première motivation est celle, dont nous avons déjà parlé, d'accroître le confort et l'adéquation à des soucis d'efficacité des schémas proposés. Les constructions étudiées par ZAHN dans [Zah 74] (reprises dans [Knu 74]) sont significatives de cette tendance. On peut ranger dans la même catégorie, les tentatives de traiter certains problèmes en utilisant des mécanismes différents tels que les coroutines. ([DaH 72],[BNR 77]).

Une deuxième motivation plus récente est de libérer l'expression algorithmique d'une trop grande séquentialisation imposée par la nature des machines. Ceci apparaît dans des constructions liées à certaines représentations des données manipulées, notamment lorsqu'il s'agit de tableaux ou d'ensembles ordonnés ([Bol 77], [Cun 79]). On notera comment ces idées se trouvent déjà dans le langage APL avec le mécanisme général de compression. Mais les formalismes les plus intéressants de ce point de vue sont ceux qui font intervenir la notion d'ensemble : le langage SETL [KeS 75] est très intéressant dans ce domaine car il intègre l'utilisation des quantificateurs existentiel et universel de la logique du premier ordre. On notera à ce propos un article récent [Lam 79] présentant des notations "ensemblistes" adaptées à une syntaxe de style ALGOL.

Enfin, nous évoquerons une dernière motivation, proche de la précédente, qui va dans le sens d'un enrichissement des mécanismes d'abstraction disponibles : les constructions proposées dissocient dans une itération l'aspect contrôle, de l'aspect traitement (ce qui correspond aux idées que nous avons développées en présentant le traitement séquentiel). On trouve de tels schémas d'itération notamment dans les langages EUCLID [LHL 77], CLU [LSA 77], ALPHARD [SWL 77], HELOISE [Rou 78]. Par ailleurs, on retrouve ces idées dans les formalismes applicatifs ([AsW 77], [Pai 77 b] [Cas 78]

Nous concluons cette discussion en indiquant brièvement les orientations que nous suivons dans cette recherche de formalismes : comme nous venons de le souligner, les notations dissociant l'aspect contrôle de l'aspect traitement d'une itération, reflètent bien les idées de base de la méthode de traitement séquentiel. A ce titre nous cherchons à développer une notation plus générale que le simple schéma *itérer*, qui nous permette d'exprimer de manière plus adéquate, les schémas proposés dans les deux premières parties (chapitres 2 et 6). Cette notation peut se fonder sur le mécanisme de coroutine, comme par exemple ce qui est proposé dans CLU. [Mor 79]. Par ailleurs, nous voulons pouvoir mieux intégrer les schémas fondamentaux du traitement séquentiel. C'est pourquoi, nous nous efforçons de dissocier les primitives d'itération en deux groupes : celles permettant d'exprimer une recherche associative (opérateur existentiel dans SETL, opérateur *MU* dans les notations d'ARSAC, instruction *FIRST* dans ALPHARD) et celles permettant d'exprimer un parcours (opérateur universel dans SETL, instruction *FOR* dans ALPHARD).

9.3. - CONCLUSION

Nous avons présenté une notation algorithmique et des règles de traduction vers deux langages, et nous avons discuté des concepts qui nous paraissent fondamentaux.

La notation algorithmique est caractérisée par une recherche de la simplicité et de la clarification des concepts. Ceci apparaît par exemple dans les constructions relatives à la notion d'action (§ 9.1.1.), mais aussi dans une construction apparemment très simple comme le *choix*.

Nous avons intégré avec prudence des constructions issues des concepts modernes liés à la manipulation d'abstraction. L'aspect évolutif de la notation nous permettra d'intégrer des constructions plus sophistiquées au fur et à mesure d'une validation méthodologique très délicate.

En proposant des règles de traductions vers deux langages de degré d'évaluation différents, nous avons pu montrer que les concepts importants de la construction des programmes peuvent être utilisés quel que soit l'environnement informatique disponible. On retrouve la base essentielle de la notion même de notation algorithmique.

Enfin, nous avons essayé d'organiser la discussion des concepts fondamentaux sur la base même du principe d'analyse descendante des problèmes : une réduction des problèmes, suivie de la description d'algorithmes en termes de schémas de composition et d'objets abstraits (issus de la réduction).

CONCLUSION : LE PROJET MEFIA POINT DE DEPART D'UN SYSTEME D'AIDE A LA PROGRAMMATION SYSTEMATIQUE

La notation algorithmique que nous venons de présenter est un élément important dans notre travail de recherche :

. sur un plan conceptuel, elle nous permet de faire constamment la synthèse sur les mécanismes intellectuels et les démarches d'analyse qui nous paraissent fondamentaux. Toute évolution sur ces points est aussitôt reflétée dans la notation. Ceci apparaît nettement dans les diverses publications et ouvrages didactiques que nous avons écrits dans les dernières années.

. sur un plan pratique, elle nous permet d'assurer le lien entre notre recherche sur la méthodologie de la programmation et une pratique quotidienne de cette activité (le "retour" obtenu étant primordial dans la conduite de cette recherche), que ce soit dans un contexte d'enseignement ou dans le cadre d'une étude des algorithmes fondamentaux.

L'évolution constante des idées et le caractère expérimental et pragmatique de nos travaux, nous ont appris à privilégier le critère de simplicité dans la définition des éléments composant la notation. Cette simplicité est obtenue en recherchant constamment les concepts fondamentaux sous-jacents aux constructions envisagées et en rejettant assez systématiquement une trop grande sophistication qui nous semble plus du ressort d'un langage de programmation. Nous pouvons ainsi facilement modifier la notation et tout aussi facilement l'adapter aux environnements informatiques actuels. Enfin tous ces points sont autant de facteurs qui favorisent une diffusion rapide dans le milieu professionnel des idées issues des recherches méthodologiques.

Nous nous sommes efforcés de concrétiser les idées présentées dans ce chapitre et d'en évaluer la cohérence, en développant leur mise en forme dans le cadre plus strict d'un langage de programmation.

Le cadre de cette concrétisation est le projet MEFIA : à partir d'expériences diverses dans plusieurs domaines liés à la programmation (méthodologie et didactique dont nous venons de parler, conception et compilation des langages, cf. [Gri 74], [CSV 76]), nous avons construit le langage MEFIA après avoir établi :

. les concepts fondamentaux de l'algorithmique qu'il doit permettre d'exprimer Nous venons d'en donner les détails dans ce chapitre.

. les propriétés importantes que tout programme doit vérifier afin d'assurer leur fiabilité. Cette discussion est amorcée dans [Gri 76a] puis reprise dans [Gri 77].

. le rôle important que doit jouer le compilateur du point de vue de la vérification des propriétés des programmes [Gri 76 b].

Dans une forme initiale simple, le langage MEFIA ne comporte que des éléments classiques (correspondant à peu près à ceux que nous avons énumérés dans les paragraphes précédents). On peut en souligner les caractéristiques techniques importantes : c'est un langage fortement typé (afin de pouvoir multiplier les vérifications statiques), dans lequel on s'efforce de chasser les possibilités d'effet de bord par des contraintes sur l'utilisation des procédures, et qui comporte un mécanisme de regroupement de procédures permettant d'assurer une certaine modularité. Soulignons par ailleurs, la disparition des blocs d'ALGOL. (cf. [CGS 78] et [SGC 78]).

Dans une deuxième forme plus sophistiquée, CUNIN a développé certains des aspects du langage. Soulignons notamment :

. La paramétrisation des univers et leur imbrication, qui conduisent à la notion de "méta module". Ceci s'inscrit dans la ligne des constructions proposées à l'heure actuelle pour concrétiser la notion de type abstrait.

. L'examen des possibilités d'expression de propriétés des valeurs manipulées dans un programme et des relations qu'elles vérifient entre elles. L'approche prise est relativement originale : citons CUNIN [Cun 79]p. 17)

"Nous avons pris à ce sujet deux décisions importantes nous démarquant totalement des autres langages prototypes :

- une propriété s'exprime comme une fonction à résultat booléen. Du fait de cette liberté d'expression, nous ne cherchons pas à savoir statiquement à quoi correspond une propriété... nous ne nous intéressons statiquement qu'à la conservation et à la propagation de la totalité de cette propriété.

- une propriété ou une relation caractérise les valeurs prises par un ou plusieurs objets. Pour définir une propriété, il n'est donc pas nécessaire de déclarer un nouveau type avec ses opérations associées".

...

Notre souci de concrétisation des idées ne s'arrête pas à la définition du langage MEFIA (dont les spécifications formelles vont être étudiées à NANCY). Nous basant sur la première forme simple du langage que nous avons évoquée ci-dessus, nous avons entrepris la réalisation d'une série de traducteurs permettant de passer de MEFIA à plusieurs langages de degré d'évolution différent. Cette étude se passe dans le cadre de projets de DEA à l'Université de Grenoble.

Un premier traducteur permettant de produire des programmes PL/1 est en cours de finition ([Mor 79][Cas 79]). Il a une structure extrêmement modulaire basée sur une forme interne reflétant la sémantique profonde des programmes (cf. les "arbres abstraits" définis dans [CSV 76]). La production de cette forme interne est faite à l'aide d'un générateur d'analyseurs syntaxiques [CuG 78]. Deux modules distincts permettent d'une part d'assurer les vérifications statiques, d'autre part d'appliquer les règles de traduction vers PL/1. Le traducteur comporte de plus un modèle original d'édition des programmes MEFIA assurant l'indentation des programmes, l'édition d'une table de références croisées et l'édition des commentaires du programme en fonction de leur position syntaxique (cette édition permet notamment de distinguer des commentaires généraux, de commentaires spécifiant les actions et les fonctions, de commentaires indiquant le rôle de certaines variables).

A partir de cette première expérience, nous envisageons un projet plus large comportant les éléments suivants :

. extension limitée du langage de base : notamment étude de l'intégration de primitives plus élaborées d'itération (cf. la discussion du § 9.2.3.).

. étude d'un autre traducteur, par exemple vers le langage FORTRAN : cette étude répond à deux objectifs : (cf. [Cas 79]) :

- réaliser un nouveau traducteur, vers un langage dont la sémantique est nettement moins riche que celle de MEFIA (alors que PL/1 est suffisamment riche pour qu'il y ait peu de problèmes de traduction à proprement parler, le traducteur jouant surtout un rôle de vérificateur et de filtre [Gri 76 b]

- étudier les éléments communs aux deux traducteurs et en déduire des règles générales pour la conduite de tels projets, et si possible des outils de génération automatique de composants des traducteurs (voir l'étude prospective dans [ScC 76]).

. étude d'une "banque" de schémas, regroupant les schémas que nous avons présentés dans les deux premières parties de cette thèse (et éventuellement d'autres à définir pour traiter d'autres types de problèmes). Cette étude aurait un double intérêt :

- la question du formalisme dans lequel introduire les schémas, et les protocoles permettant de le faire
- la question des protocoles d'utilisation des schémas.

Il est clair qu'une telle étude peut amener à une expérimentation concrète sur l'utilisation de constructions linguistiques reflétant la notion de type abstrait.

. à terme intégration de tous ces éléments dans un premier système d'aide à la programmation systématique pouvant comporter en plus :

- des outils d'aide à la programmation descendante (on pourra se baser sur des expériences telles que PEARL [Sno 72] ou TOPD [HSG 75]).

- des outils d'aide à la vérification dans la ligne de projets tels que [AAO 73] : ce type d'outil permet de pratiquer des preuves par manipulation d'assertions. En fait il engendre des théorèmes que l'utilisateur doit démontrer lui-même. Une partie intéressante d'un tel projet est la manipulation formelle de formules, leur simplification et leur édition. Un tel outil aurait surtout un intérêt dans un contexte pédagogique. Il nécessite d'intégrer au langage de base, un langage simple d'assertions.

- des primitives de gestion de fichiers permettant de maintenir constamment l'historique des programmes écrits, les relations entre les modules, afin de pouvoir notamment engendrer automatiquement un état des répercussions entraînées sur un ensemble de programmes par la modification de l'un d'entre eux (cf. notamment [Hen 77]).

- un système d'édition "intelligent" de programmes permettant à l'utilisateur de travailler au niveau de la syntaxe profonde (cf. par exemple le projet MENTOR [KHM] développé autour de PASCAL). Un tel système semble intéressant pour la manipulation formelle des programmes, notamment pour pouvoir pratiquer des transformations automatiques de programmes. Là encore, les applications pédagogiques sont évidentes.

Il est remarquable que toutes les idées présentées ici peuvent être menées à bien sans trop de difficultés techniques d'importance. Le point essentiel fédérateur d'un tel projet est l'existence d'une notation de base, formalisée sous forme d'un langage de programmation. Le langage MEFIA, même dans sa forme simple évoquée ci-dessus, ouvre ainsi la possibilité de réunir dans un vaste projet un ensemble de logiciels d'aide à la programmation systématique, dont l'étude fournira autant de points de départ vers de nouvelles voies de recherches, et autant de supports pour l'expérimentation de méthodes de programmation systématique.

CONCLUSION

Nous concluons cette thèse en énumérant quelques principes méthodologiques qui nous ont guidés et en imaginant quelques prolongements à nos travaux :

. définir des problèmes généraux qui modélisent un grand nombre de problèmes traités concrètement. Nous avons étudié la question du traitement d'un ensemble d'informations et nous en avons déduit les deux méthodes proposées.

Peut-on définir d'autres classes de problèmes ? en déduire des méthodes générale ?

. fournir un modèle du processus de construction de programmes : nous avons donné des étapes précises et associé un ensemble d'outils à chacune de ces étapes (modèles d'analyse et de description, techniques de raffinement et de simplification).

Bien que nous ayons tenté d'être systématique dans la description de ce processus, nous sommes restés très informels : la question importante à laquelle nous devons répondre maintenant est celle de la formalisation de ce processus de construction de programmes.

. établir le cadre du processus de construction de programmes en tirant parti des connaissances relatives aux structures de données concrètes. Nous avons étudié les files et les arbres. Notre prochaine étape concernera les graphes, espérant ainsi définir une troisième méthode sur le même modèle que le "traitement séquentiel" et le "traitement arborescent".

. fournir une classification précise des algorithmes relevant d'une méthode particulière, en décrivant les étapes de leur construction et les choix dont ils sont issus. L'aspect fastidieux de cette énumération pourra t'il diminuer dans le cadre d'une automatisation, de la création d'une "banque de schémas" ?

. libérer l'expression des algorithmes des contraintes imposées par les environnements informatiques. Nous proposons un modèle de comportement vis à vis des langages de programmation fondé sur la notion de "notation algorithmique". L'objectif principal que nous poursuivons est d'augmenter l'indépendance des "programmeurs" en favorisant les critères de "portabilité" (des programmes, des programmeurs).

Pourra-t-on développer des langages et des systèmes d'aide à la programmation qui rapprochent la machine de l'homme, afin d'éviter à l'homme de trop avoir à se rapprocher d'elle ?

ANNEXE A

EVOLUTION DE L'ENSEIGNEMENT DE L'ALGORITHMIQUE

ET DE LA PROGRAMMATION

Nous retraçons l'évolution de l'enseignement de la programmation par un choix d'ouvrages didactiques et d'articles spécialisés. Nous montrons notamment que les universités françaises ont été particulièrement actives en ce domaine durant les cinq dernières années.

1. REVUES SPECIALISEES

- groupe SIGCSE de l'ACM : voir notamment les actes des Congrès Annuels de 1974 à 1976

- groupe "enseignement de l'informatique" de l'AFCET :
  réunion de Quiberon (mai 1976), ateliers aux congrès AFCET 1976 et 1978
  Colloque sur l'unité de la démarche informatique ([IRE 78]).

2. ECOLES

- Ecole d'Eté de l'AFCET :
  Ales (1971) cf. [Pai 71], Rabat (1975) cf. [CGL 75], Montréal (1977) cf. [Bol 77]

- Ecoles Internationales :
  Toulouse (1977) cf. [CCE 77], Breau Sans Nappe (1978) cf. [BRE 78], Marktoberdorf (1978) cf. [MOD 78]

3. CONTRIBUTION DES UNIVERSITES FRANCAISES

On trouvera une synthèse dans [Com 78].

. Grenoble : [CoV 74 a et b], [Vei 74], [LuS 75], [Sch 77 b], [Luc 78] [Sch 79 a]

. Nancy : [BHP 77], [BFH 78]

. Toulouse : [Che 76], [Gal 77]

. Rennes : Travail autour du langage ALGOL 68 tout d'abord puis en utilisant [Ger 77] et finalement sur la base d'une notation algorithmique.

J. BARRE, A. COUVERT, J. RISTORI :
La programmation ? Mais c'est très simple !
Version 2 - UER Mathématiques et Informatique - Rennes 1978

. St Etienne: [Mah 73]

. Besançon : G.R. PERRIN
Cours et exercices de programmation à l'usage des étudiants de Maîtrise SMI
Université de Franche Comté - Besançon 1978

. Lille : G. JACOB
L'enseignement de la programmation en DEUG à Lille 1
Ecole d'Eté de Monastir 1979

G. COMYN
Algorithmique
Notes de cours 1978-79 - IUT Informatique - Lille

. Aix en Provence :
D. FENEUILLE
50 heures d'algorithmie à l'IUT
IUT d'Aix en Provence - 1978

...

4. LIVRES, MONOGRAPHIES

. Français :[PaG 77], [Ars 77], [Ger 77], [MaB 78], [MeB 78]

. Etrangers : [Knu 69 et 73], [Dij 71 et 76], [Wir 73 et 76], [Kep 74 et 76 [AHU 74], [Jack 75], [Baa 78]

ANNEXE B

CLASSEMENT PAR THEMES DES OUVRAGES CITES EN BIBLIOGRAPHIE

1. TRAVAUX PRESENTES DANS CETTE THESE

1.1 - Traitement séquentiel

[LuS 75], [LSV 77], [Sch 78 a et b], [Luc 78]

1.2 - traitement arborescent

[Sch 76], [Sch 77 b]

1.3 - Notation algorithmique , MEFIA

[ScC 76], [LuS 76], [Gri 77], [CGS 78], [Dub 78], [SGC 78], [Cun 79] [Cas 79], [Mor 79]

2. OUVRAGES DIDACTIQUES D'INTERET GENERAL

2.1 - Construction de programmes

[Wir 73], [Jack 75], [Dij 76], [Ars 77], [Ger 77], [Fin 79], [Hug 79]

2.2 - Structures de données et algorithmique

[Knu 69 et 73], [Mah 73], [AHU 74], [Kun 76],[Wir 76] , [PaG 77], [MaB 78], [MeB 78], [Baa 78]

2.3 - Logique et théorie de la programmation

[MaW 77], [Liv 78]

2.4 - Méthodologie de la programmation

[Wir 71 b], [DDH 72], [Par 72 a et b], [DAT 73], [CS 74], [KeP 74], [KeP 76],[Pai 77 a], [Gal 77], [Yeh 77], [Pai 78], [Fin 79]

2.5 - Congrès, écoles

[CGL 75], [Wor 77], [CCE 77], [IRE 78], [Rob 78], [BRE 78], [MOD 78], [AFC 78]

3. LANGAGES ET TRADUCTEURS

3.1 - Langages

[Mac 60], [Mac 61], [Ive 62], [DMN 68], [Wir 71 a], [Bri 75], [KeS 75] [WLS 76], [CCK 76], [BKL 77], [GrH 77], [LHL 77], [AsW 77], [GMS 77], [Rou 77], [Cas 78]

3.2 - Synthèses sur les langages

[Schu 75], [BaS 76], [Wor 77], [Bac 78]

3.3 - Types abstraits

[LiZ 74], [Jor 75], [WLS 76], [LSA 77], [GHM 78], [Jac 78]

3.4 - Itérations et itérateurs

[Zah 74], [LeM 75], [LSA 77], [Lam 79], [SWL 77], [Pai 77 b], [Bol 78] [Rou 78], [BFH 79],

3.5 - Traducteurs, compilation

[GoH 74], [Bro 76], [Gri 76 b], [CSV 76], [CuG 78]

4. PREUVE DE PROGRAMMES

[Mac 62], [Nau 66], [Flo 67 a], [Hoa 69], [Bur 69], [Dij 72], [Wir 73] [MaW 77], [Liv 78]

5. LOGICIELS D'AIDE A LA PROGRAMMATION

[Sno 72], [AAO 73], [HSG 75], [KHM 77], [Ken 77], [Ars 78], [GaM 78], [Com 79],[Bau 78]

6. TRANSFORMATIONS DE PROGRAMMES

6.1-Programmation par transformations successives :

[Dar 72], [Gri 76 a], [Bau 76], [Weg 76], [BuF 77], [PPW 78]

6.2 - Techniques de transformation

[Knu 74], [Mon 76], [BPP 76], [Bau 76], [Cha 77], [Ars 79]

6.3 - Transformations "récursif itératif"

[Coo 66], [Vei 76], [PaP 76], [Bir 77], [Ber 78], [ArK 79]

6.4 - Systèmes de transformations

[DaB 76], [BuD 77], [Bau 78], [Ars 78]

7. TECHNIQUES DIVERSES

7.1 - "backtracking" et "Branch and Bound"

[Flo 67], [Bir 75], [GrH 77], [Sak 79]

7.2 - Algorithmes divers, études de cas

[Hea 63], [Wil 64], [HeS 72], [Sed 77], [Dij 78]

BIBLIOGRAPHIE

AAO 73 M. AJENSTAT, B. AMY, F. OUABDESSELAM
S.V.P. Système de validation de programmes. Un système intéractif pour la conception de programmes corrects.
Rapport de Recherche - ENSIMAG - Grenoble 1973

AFC 78 Congrès de l'AFCET
Théorie et techniques de l'informatique
Actes du congrès - 13-15 novembre 1978 - Editions Hommes et Techniqu

AHU 74 A.V. AHO, J.E. HOPCROFT, J.D. ULLMAN
The design and analysis of computer algorithms
Addison Wesley - 1974

ArK 79 J. ARSAC, Y. KODRATOFF
Some methods for transformation of recursive procedure into iterative ones
RR 79-2 LITP Institut de Programmation - Paris 7 - 1979

Ars 77 J. ARSAC
La construction de programmes structurés
Dunod Informatique 1977

Ars 78 J. ARSAC
Manuel d'utilisation du programme EXM, adaptation de ce système à la MITRA 15/LSE
Rapport n° 78-45 , LITP Paris Novembre 1978

Ars 79 J. ARSAC
Syntactic source to source transforms and program manipulation
Communications of the ACM - V ol. 22 n° 1 January 1979

AsW 77 E.A. ASHCROFT , W.W. WADGE
LUCID, a non prodecural language with iteration
Communication of the ACM vol. 20 n° 7 July 1977

Baa 78 S. BAASE
Computer algorithms : introduction to design and analysis
Addison - Wesley Publishing Company - 1978

Bac 78 J. BACKUS
Can programming be libetared from the von Neumann style ?
A functional style and its algebra of programs.
1977 ACM Turing Award lecture - Communication of the ACM, vol 21, n° 8, august 1978

BaE 74 [Cf. GoH 74]

BaS 76 F.L. BAUER, K. SAMELSON editors
Language hierarchies and interfaces
Lecture notes in computer Science n° 46 Springer Verlag 1976

Bau 76 F.L. BAUER
Programming as an evolutionary process
Bericht Nr 7617 - T.U.M. - Munich 1976

Bau 78 F.L. BAUER
Towards a wide spectrum language to support program specification and program development
SIGPLAN Notices - 1978

Ber 78 P. BERLIOUX
Application des propriétés des compositions séquentielle et parallèle des instructions à la transformation des programmes récursifs
RR.100 Laboratoire IMAG-USMG Grenoble 1978 aussi dans [Rob 78]pp.187-

BFH 78 F. BELLEGARDE, J.P. FINANCE, B. HUC, J.M. PIERREL, A. QUERE, J.L. REMY
Initiation à une construction méthodique de programmes
Rapport CRIN 78E81 Nancy 1978

BFH 79 F. BELLEGARDE, J.P. FINANCE, B. HUC, J. JARAY
Quelques prolongements de la méthode de programmation déductive
GROPLAN - Groupe programmation et langages de l'AFCET n° 6 1979

BHP 77 F. BELLEGARDE, V. HUC, J.M. PEIRREL, A. QUERE
Initiation à une construction méthodique de programmes
Publication CRIN n° 77R035 - IUT Informatique - Nancy 1977

Bir 77 R.S. BIRD
Notes on recursion elimination
Communications of the ACM - vol 20 n° 6 June 1977

BiR 75 J.R. BITNER, E.M. REINGOLD
Backtrack Programming Techniques
Communication of the ACM , vol 18, n° 11 Nov. 1975

BKL 77 J. BEZIVIN, W.H. KAUBISCH, A. LEROY, J.L. NEBUT, R. RANNOU
SIMONE : manuel de référence
Publication IRIA/SFER - BP105 78150 LE Chesnay - 1977

BNR 77 J. BEZIVIN, J.L. NEBUT, R. RANNOU
Another view of coroutines
Publication interne n° 76 IRISA - RENNES 1977

BoL 77 J.C. BOUSSARD, O. LECARME
Didactique de la programmation
Cours de l'Ecole d'Etat - AFCET-Montréal 1977
Informatique Mathématiques Automatique - Nice Juillet 1977

BPP 76 F.L. BAUER, H. PARTSCH, P. PEPPER, H. WOESSNER
Techniques for Program Development
Interner Bericht - Technische Universität München
Institut für Informatick - Septembre 1976

BRE 78 Ecole d'été internationale IRIA, EDF, CEA
BREAU Sans Nappe 10-28 juillet 1978

Bri 75 P. BRINCH HANSEN
The purpose of Concurrent Pascal
Proceedings of ICRS - 1975

Bro 76 P.J. BROWN (editor)
Software Portability, an advanced course
Combridge University Press - 1976

BuD 77 R.M. BURSTALL and J. DARLINGTON
A Transformation System for developing recursive programs
JACM Vol. 24 n° 1 pp. 44-67 - January 1977

BuF 77 R. BURSTALL, M. FEATHER
Program development by transformation : an overview
dans [CCE 77]

Bur 69 R.M. BURSTALL
Proving properties of programs by structural induction
Computing Journal , vol 12, n° 1pp. 41-48 - February 1969

Cas 78 CASTOR
MEDEE : a type of language for constructing program
Rapport CRIN n° 78R074 - Nancy 1978

Cas 79 C. CASERY
Construction méthodique de programmes : réalisation d'un traducteur de la notation MEFIA, vers PL/1 - Généralisation à d'autres langages
Rapport de DEA - USMG Grenoble 1979 -

CCE 77 Cours de la Commission des Communautés Européennes
Les fondements de la programmation
Toulouse 1977 - Edité par l'IRIA

CCK 76 J.L. CHEVAL, F. CRISTIAN, S. KRAKOWIAK, Ma. LUCAS, J. MONTUELLE? J. MOSSIERE
Un système d'aide à l'écriture des systèmes d'exploitation
Congrès AFCET - Panorama de la nouveauté informatique en France
Gif/Yvette 1976

CGL 75 B. CHERBONNEAU, M. GALINIER,J.P. LAGASSE, H. MASSIE, B. MATHIS, J.L. PAUL
Programmation structurée
Ecole d'Eté de l'AFCET - Rabat 1975 - Rapport n° 112 UER Informatique Université Paul Sabatier - Toulouse 1975

CGS 78 P.Y. CUNIN, M. GRIFFITHS, P.C. SCHOLL
Aspects fondamentaux du langage MEFIA
Journées d'études sur la fiabilité des programmes dans les applications industrielles (IRIA-EDF-Chapitre français de l'ACM)
Clamart 26-27 avril 1978

Cha 77 P. CHATELIN
Self-Redefinition as a program manipulation strategy
Proceedings of Symposium on Artificial Intelligence and Programming Languages - ACM SIGPLAN notices and SIGART Newsletter - August 1977 pp. 174-179

Che 76 B. CHERBONNEAU
Conception d'un projet étudiant par machines abstraites
Congrès AFCET 1976, Atelier Enseignement de l'informatique
Novembre 1976

Com 78 G. COMYN
Orientation actuelle de l'enseignement de la programmation
dans [IRE 78]

Com 79 D. COMER
MAP : A Pascal Macro Preprocessor for large program development
Software-Practice and Experience, vol. 9 pp. 203-309 - 1979

Coo 66 D.C. COOPER
The equivalence of certain computation
Comp. J. pp. 45-52 1966

CoV 74 a J. COURTIN, J. VOIRON
Introduction à l'algorithmique et aux structures de données
IUT B cours - Grenoble 1974

CoV 74 b J. COURTIN, J. VOIRON
Traduction des schémas de programmes en FORTRAN
IUT Informatique - USSG Grenoble 1974

CS 74 ACM Computing Surveys
Special Issue : Programming
ACM Computing surveys, vol 6, n° 4 December 1974

CSV 76 P.Y. CUNIN, M. SIMONET, J. VOIRON
Méthodologie d'écriture de compilateurs,. Une expérience du langage ALGOL 68
Thèse de docteur ingénieur - USMG, INPG Grenoble 1976

CuG 78 P.Y. CUNIN - M. GRIFFITHS
Générateur d'analyseurs LL(1) sur IRIS 80
Rapport 78-R-008 CRIN - Nancy 1978

Cun 79 P.Y. CUNIN
Fiabilité et sécurité des programmes : propositions autour d'un langage d'essai
Thèse - CRIN, INPL - Nancy 1979

DaB 76 J. DARLINGTON, R.M. BURSTALL
A system which automatically improves programs
Acta Informatica, 6, pp. 41-60, Springer-Verlag 1976

DaH 72 O.J. DAHL, C.A.R. HOARE
Hierarchical program structures
Dans [DDH 72]

Dah 77 O.J. DAHL
Can program proving be made practical
dans [CCE 77]

Dar 72 J. DARLINGTON
A semantic approach to automatic program improvment
Ph.D. Thesis - Department of Machine Intelligence - University of Edinburgh 1972

DAT 73 Revue DATAMATION
Vol. 19, n° 12 1973

DDH 72 O.J. DAHL, E.W. DIJKSTRA, C.A.R. HOARE
Structured Programming
Academic Press 1972

Dij 65 E.W. DIJKSTRA
Programming considered as a human activity
Proceedings IFIP COngress 1965
North Holland Publishing Company, Amsterdam 1965

Dij 71 E.W. DIJKSTRA
A short introduction to the art of programming
EWD 316 - Department of Mathematics - Eindhoven 1971

Dij 72 E.W. DIJKSTRA
Notes on structured programming
dans [DDH 72]

Dij 76 E.W. DIJKSTRA
A discipline of programming
Prentice Hall series in automatic computation - 1976

Dij 78 E.W. DIJKSTRA
In honour of Fibonacci
EWD 654 - Plataanstraat 5 - 5671 Al Nuenen - The Netherlands

DMN 68 O.J. DAHL, B. MYHRHAUG, K. NYGAAR
The SIMULA 67 common base language
Publication S22, Norvegian Computing Centre - Oslo 1968

Dub 78 J.P. DUBOURREAU
Construction méthodique de programmes : étude du langage MEFIA et d'un traducteur
Rapport de DEA - INPG - Grenoble 1978

Fin 79 J.P. FINANCE
Réflexions sur la construction de programmes : outils et méthodes de spécifications et de résolution de problèmes
Thèse d'Etat - CRIN - Nancy 1979

Flo 67 R.W. FLOYD
Non deterministic algorithms
Journal of the ACM, vol. 14, n° 4, october 1967, pp. 636-644

Flo 67 a R.W. FLOYD
Assigning meanings to programs
Proceedings fo Symposium in applied mathematics
Vol. 19 (J.T. SCHWARTZ Editor) American Mathematical Society
Providence R.I. pp. 19-32 - 1967

FPS 76 G. FAFIOTTE, S. PAINVIN, P.C. SCHOLL
Insertion de l'enseignement assisté par ordinateur dans l'enseignement secondaire , formation à l'E.A.O. : analyse et propositions
RR 30, Laboratoire IMAG - Grenoble 1976

Gal 77 M. GALINIER
A software design methodology and associated tools
dans [CCE 77]

GaM 78 M. GALINIER, A. MATHIS
Les machines abstraites : unités de conception de logiciel fiable
Congrès sur la fiabilité des programmes dans les applications industrielles. Chapitre français de l'ACM, EDF,IRIA, 26-27 avril 1978

Ger 77 A. GERBIER
Mes premières constructions de programmes
Lecture No es in Computer Sciences n° 55 - Springer Verlag 1977

GHM 78 J.V. GUTTAG, E. HOROWITZ, D.R. MUSSER
Abstract data types and software validation
Communication of the ACM vol. 21 n° 12 décembre 1978

GMS 77 C.M. GESCHKE, J.H. MORRIS Jr, E.H. SATTERTHWAITE
Early experience with MESA
Communications of the ACM - vol. 20, n° 8 august 1977

GoH 74 G. GOOS, J. HARTMANIS (editors)
Compiler construction - an advanced course
Lecture notes in computer sciences, n° 21 Springer Verlag mars 1974

GrH 77 R.E. GRISWOLD, D.R. HANSON
Language facilities for programmable backtracking
Proceedings of the symposium on artificial intelligence and programming languages. SIGPLAN Notices Vol. 12, n° 8 SIGART Newletter n° 64 - August 1977

Gri 73 M. GRIFFITHS
Exercices de programmation
Maîtrise d'Informatique - C4 - ENSIMAG 2°A. USMG-INPG
Grenoble 1973

Gri 74 M. GRIFFITHS
LL(1) grammars and analysis - Introduction to compiler compilers
dans [GoH 74]

Gri 76 a M. GRIFFITHS
Program production by successive transformation
dans [BaS 76]

Gri 76 b M. GRIFFITHS
Verifiers and Filters et Translation between high level languages
dans [Bro 76]

Gri 77 M. GRIFFITHS
Language support for program construction
dans [CCE 77]

Hea 63 B.R. HEAP
Permutations by interchanges
Computer J. 6, pp. 293-294 - 1963

Hen 77 P. HENDERSON
Relational data models for describing system structure
dans [CCE 77]

HeS 72 P. HENDERSON, R. SNOWDON
An experiment in structured programming
BIT 12, 1 - January 1972

Hoa 69 C.A.R. HOARE
An axiomatic basis of computer programming
Communications of the ACM vol 12, n° 10, pp. 576-580, 583 octobre 1969

Hoa 72 C.A.R. HOARE
Notes on data structuring
Dans [DDH 77]

HSG 75 P. HENDERSON, A.R. SNOWDON, J.D. GORRIE, I.I. KING
The TOPD System
Technical report series n° 77
Computing Laboratory - University of Newcastle upon Tyne - Setp. 1975

Hug 79 J.W. HUGHES
A formalization and explication of the Michael Jackson method of program design
Software Practice and experience, vol 9, pp. 191-202- 1979

IRE 78 congrès IREM-AFCET-INRP, Toulouse janvier 1978
Unité de la démarche informatique, du super ordinateur à la calculatrice de poche
Bulletin AFCET Informatique Enseignement, VOl.2 n° 2 Juin 1978
(on peut se procurer ces actes auprès des IREM).

IRE 79 Groupe Informatique de l'IREM
Fiches BASIC
IREM-USMG Grenoble 1979

Ive 62 K.E. IVERSON
A programming language
John Wiley and Sons - New York 1962

Jac 78 P. JACQUET
Les types génériques : propositions pour un mécanisme d'abstraction dans les langages de programmation
Thèse 3ème cycle - USMG. INPG Grenoble 1978

Jack 75 M.A. JACKSON
Principles of Program Design
Academic Press - London 1975

Jor 75 P. JORRAND
Contribution au développement des langages extensibles
Thèse d'état USMG, INPG - Grenoble 1975

KeP 74 B.W. KERNIGHAN, P.J. PLAUGER
The elements of programming style
Mc Graw Hill , 1974

KeP 76 B. KERNIGHAN, P.J. PLAUGER
Software tools
Addison Wesley Publishing Company - 1976

KeS 75 K. KENNEDY, J. SCHWARTZ
An introduction to the set theoretical language SETL.
Comp. and Maths with Appls, vol. 1 pp. 97-119 - 1975

KHM 77 G. KAHN, G. HUET, P. MAURICE
Environnement de programmation PASCAL
Manuel d'utilisation sous SIRIS 7/8 - novembre 1977

Knu 69 D.E. KNUTH
The art of computer programming : vol1. Fundamentals algorithms
Addisson Wesley Publishing Compagny 1969

Knu 73 D.E. KNUTH
The art of computer programming : vol. 3 sorting and searching
Addisson Wesley Publishing Company 1973

Knu 74 D.E. KNUTH
Structured programming with GOTO statement
dans[CS 74]et aussi dans [YEH 77]

KuM 77 J. KUNTZMANN, J. MESSULAM
Retombées informatiques dans l'enseignement des mathématiques
RR 79 - Laboratoire IMAG - Grenoble - Septembre 1977

Kun 76 J. KUNTZMANN
Manipulation de l'information
Projet de cours en Maîtrise d'Enseignement de Mathématique
1°A. (partie D) USMG 1976

LAM 79 L. LAMPORT
A general construction for expressing repetition
SIGPLAN Notices (ACM) vol. 1 n° 3 - march 1979

Lea 72 B.M. LEAVENWORTH
Programming with(out) the GOTO
Proceedings of the ACM annual conference, Boston (MASS)
pp. 782-86 - August 1972

LeM 75 H.F. LEDGARD, M. MARCOTTY
A genealogy of control structures
Communications of the ACM, 18, 11 novembre 1975- pp. 629-639

LHL 77 W. LAMPSON, J.J. HORNING, R.L. LONDON, F.G. MITCHELL, G.J. POPEK
Report on the programming language EUCLID
SIGPLAN Notices (ACM) vol 12, n° 2 - février 1977

Liv 78 C. LIVERCY
Théorie des programmes
Dunod 1978

LiZ 74 B. LISKOV, Z. ZILLES
Programming with abstract data types
Proceedings of ACM SIGPLAN Conference on very high level languages
SIGPLAN Notices vol. 9 n° 4 April 1974

LMS 75 M. LUCAS, J. MOSSIERE, P.C. SCHOLL
Initiation à la programmation : réflexions et propositions
RAIRO 9° année - B-1 pp. 5-27 mars 1975

LSA 77 B. LISKOV, A. SNYDER, R. ATKINSON, C. SCHAFFERT
Abstraction mechanisms in CLU
Communications of the ACM - Vol. 20, n° 8 August 1977

LSV 77 M. LUCAS, P.C. SCHOLL, J. VOIRON
Apprentissage et utilisation du traitement séquentiel pour la construction de programmes
Rapport de recherche RR 74 - Laboratoire IMAG-USMG- Grenoble 1977

Luc 78 M. LUCAS
Algorithmique et représentation des données - notions élémentaires
Maîtrise d'Informatique - C2 - ENSIMAG 1° A. USMG-INPG Grenoble 1978

LuS 75 M. LUCAS, P.C. SCHOLL
Propositions pour une initiation à l'algorithmique
(2 volumes) Laboratoire IMAG-USMG Grenoble 1975

LuS 76 M. LUCAS, P.C. SCHOLL
Un exemple d'intégration de concepts méthodologiques dans l'enseignement de la programmation
Atelier algorithmique et programmation - Congrès AFCET "panorama de la nouveauté informatique en France"
Bulletin AFCET Informatique enseignement - VOL.1 n° 3 novembre 1976

LuS 79 M. LUCAS, P.C. SCHOLL
Un exemple de construction systématique de programmes
Rapport de recherche IMAG - A paraitre 1979

MaB 78 R. MAHL,J.C. BOUSSARD
Algorithmique et structures de données
Laboratoire d'informatique - Université de Nice 1978

Mac 60 J. Mc. CARTHY
Recursive functions of symbolic expressions and their computation by machine, Part. 1
Communications of the ACM, avril 1960

Mac 61 J. Mc CARTHY
A basis for a mathematical theory of computation
Séminaire IBM Blaricum (Hollande) 1961 - dans "Computer Programming anf Formal Systems" P. BRAFFORT and D. HIRSCHBERG Editors
North Holland Publishing Cimpagny - 1963

Mac 62 J. Mc CARTHY
Toward a mathematical Science of Computation
Proceedings IFIP Congress 1962, Muchih (Germany) North Holland Publishing Company Amsterdam 1963

Mah 73 R. MAHL
Algorithmique et structures de données
Cours d'informatique software - ENS Mines - Saint Etienne 1973

MaW 77 Z. MANNA, R. WALDINGER
The logic of computer programming
Technical Note 154 Artificial Intelligence Center Stanford Research Institute - August 1977

MeB 78 B. MEYER, C. BAUDOIN
Méthodes de programmation
Collection de la D.E.R. d'EDF - Eyrolles 1978

MOD 78 International summer school on program construction
Marktoberdorf - Juillet-Août 1978

Mon 76 B. MONT-REYNAUD
Removing trivial assignments from programs
STAN-CS-76-544 Computer Science Department Stanford University -March

Mor 79 P. MORAT
Construction méthodique de programmes : réalisation d'un traducteur de la notation MEFIA, étude de quelques primitives de contrôle
Rapport de DEA - INPG - Grenoble 1979 -

Nau 66 P. NAUR
Proof of algorithms by general snapshots
BIT6, pp. 310-316 - 1966

PaG 77 C. PAIR, M.C. GAUDEL
Les structures de données et leur représentation en mémoire
IRIA - JUIN 1977

Pai 71 C. PAIR
Les structures de l'information
Cours de l'Ecole d'Eté de l'AFCET - Alès 1971

Pai 77 a C. PAIR
La construction des programmes
Rapport CRIN 77-R-019 - Nancy 1977
Publié aussi dans RAIRO Informatique vol. 13- n°2 pp. 113-138

Pai 77 b C. PAIR
Mise en évidence de l'ensemble de départ dans les itérations en programmation déductive
A tout crin - Bulletin de liaison n° 6 CRI Nancy octobre 1977

Pai 78 C. PAIR
La programmation : de l'énoncé au programme
dans [AFC 78]

PaP 76 H. PARTSCH, P. PEPPER
A family of rules for recursion removal related to the towers of Hanoï problem'
Bericht n° 7612 - T.U.M. Munich 1976

Par 72 a D.L. PARNAS
A technique for the specification of software modules with examples
Communications of the ACM, vol. 15, n° 5, pp. 330-336 May 1972

Par 72 b D.L. PARNAS
On the criteria to be used in decomposing systems into modules
Communication of the ACM, vol. 15 n° 12, pp. 1053-58 - december 1972

PPW 78 P. PEPPER, H. PARTSCH, H. WOESSNER, F.L. BAUER
A transformational approach to programming
dans [Rob 78] pp. 248-262

Rob 78 B. ROBINET (editor)
Transformations de programmes
3ème colloque international sur la programmation - Paris 28-30 mars 1978 - DUNOD Informatique - phase recherche

Rou 77 R. ROUSSEAU
HELOISE : un langage et une méthode pour la description fonctionnelle des grands logiciels séquentiels
Thèse de spécialité - IMAN T7 - Nice 1977

Rou 78 R. ROUSSEAU
Type abstrait de structure de contrôle : un concept utile pour la flexibilité des programmes ?
bulletin GROPLAN n° 4 (AFCET-TTI) 1978

Sak 79 M. SAKAROVITCH
Notes de cours
Cours d'Optimisation Combinatoire - USMG - Grenoble 1979 - pp. 14-36

ScC 76 P.C. SCHOLL, M. CABRIC
Outils informatiques d'aide à l'enseignement et l'apprentissage de l'algorithmique : analyse prospective
Séminaire de programmation - Laboratoire IMAG - mars 1976

Sch 70 P.C. SCHOLL
Un interpréteur APL pour le CII 90-80
Thèse de docteur ingénieur - Toulouse 1970

Sch 72 P.C. SCHOLL
An experiment in french computer assisted instruction final report
Institute for Mathematical Studies in the Social Sciences
Stanford University 1972

Sch 76 P.C. SCHOLL
Interprétation de programmes comme le traitement d'arbres : un aspect de la production de programmes par transformations successive
Rapport de recherche RR 54 - Laboratoire IMAG Grenoble 1976

Sch 77 a P.C. SCHOLL
Méthodologie de la programmation : une étude de cas : construction d'un index
dans [LSV 77] Grenoble 1977

Sch 77 b P.C. SCHOLL
Introduction à la récursivité et aux arbres
Support de cours - Institut de Programmation - USMG Grenoble 1977

Sch 78 a P.C. SCHOLL
Etude du traitement séquentiel et d'une méthode de construction de programmes associée
dans [IRE 78]

Sch 78 b P.C. SCHOLL
Le traitement séquentiel : une classe de problèmes et une méthode de construction de programmes
congrès AFCET-TTI Gif sur Yvette - Novembre 1978

Sch 79 a P.C. SCHOLL
Notes introductives au traitement de fichiers séquentiels
IREM - USMG Grenoble 1979

Sch 79 b P.C. SCHOLL
Notes de cours
Institut de Programmation - USMG - Grenoble 1978-79

Schu 75 S.A. SCHUMAN (editor)
New directions in algorithmic languages
IFIP Working Group on ALGOL - IRIA Rocquencourt 1975

ScS 72 P.C. SCHOLL, Ma. SCHOLL
Une expérience d'enseignement assisté par ordinateur du français
Congrès AFCET "les techniques de l'informatique - Grenoble 1972

Sed 77 R. SEDGEWICK
Permutation Generation Methods
Computing surveys vol. 9, n° 2 June 1977

SGC 78 P.C. SCHOLL, M. GRIFFITHS, P.Y. CUNIN
Construction méthodique et vérification systématique de programmes : éléments d'un langage
Congrès AFCET-TTI Gif sur Yvette - Novembre 1978

Sno 72 R.A. SNOWDON
PEARL : an interactive system for the preparation and validation of structured programs
SIGPLAN Notices vol. 7 n° 3 mars 1972

SWL 77 M. SHAW, W.A. WULF, R.L. LONDON
Abstraction and vérification in ALPHARD : defining and specifying itération and generators
Communications of the ACM - vol 20, n° 8 August 1977

Vei 74 G. VEILLON
Algorithmique
Cours 2ème A. ENSIMAG-C3 Logique et Programmation - USMG-INPG
Grenoble - décembre 1974

Vei 76 G. VEILLON
Transformation de programmes récursifs
RAIRO Informatique - vol.10 n° 9 septembre 1976

Weg 76 B. WEGBREIT
Goal directed program transformation
Third ACM symposium on principles of programming languages
pp. 153-170 Janvier 1976 -
aussi dans IEEE transactions on software engineering vol. SE-2
n° 2 pp. 69-80 1976

Wil 64 J.W.J. WILLIAMS
Heapsort - Algorithme 232
Communications of the ACM, 7, pp. 347-348 - Juin 1964

Wir 71 a N. WIRTH
The programming language PASCAL
Acta Informatica, vol. 1n N° 1, pp. 35-63

Wir 71 b N. WIRTH
Program development by stepwise refinement
Communications of the ACM, vol. 14 n° 4, pp. 221-27 - April 1971

Wir 73 N. WIRTH
Systematic programming : an introduction
Prentice Hall series in automatic computation - 1973
voir aussi la traduction française par O. LECARME

WIR 73 a N. WIRTH
Introduction a la programmation systématique
traduction française par O. LECARME
Monographies AFCET - Masson 1977

WIR 76 N. WIRTH
Algorithms + data structures = Programs
Prentice-Hall series in Automatic Computation 1976

WLS 76 W.A. WULF, R.L. LONDON, M. SHAW
Abstraction and verification in ALPHARD : introduction to language and methodology
Technical Report Carnegie Mellon University - 1976

Wor 77 D.B. WORTMAN (editor)
Proceedings of an ACM conference on Language Design for Reliable Software
SIGPLAN Notices Vol. 12, n° 3 mars 1977 - Cinq articles présentés à cette conférence apparaissent dans les "communications of the ACM"de juillet 1977.

Yeh 77 R.T. YEH (editor)
Current trends in programming methodology , vol. 1 : software specification an design
Prentice Hall, Englewoods Cliffs, N.J. 1977

Zah 74 C.T. ZAHN Jr
A control statement for natural top-down structured programming
Symposium on Programming Languages - Paris 1974

dernière page de la thèse

AUTORISATION DE SOUTENANCE

VU les dispositions de l'article 5 de l'arrêté du 16 Avril 1974,

VU les rapports de Messieurs :

- M. GRIFFITHS, Professeur à l'Université de NANCY II.
- M. SINTZOFF, Professeur à BRUXELLES.
- M. VEILLON, Professeur à l'Institut National Polytechnique de GRENOBLE.

Monsieur Pierre-Claude SCHOLL

est autorisé à présenter une thèse en soutenance pour l'obtention du grade de DOCTEUR D'ETAT, discipline SCIENCES.

Grenoble, le 14 Juin 1979

Le Président de l'U.S.M.G.

Le Président de l'I.N.P.G.

Ph. TRAYNARD
Président
de l'Institut National Polytechnique