Etude d'un modèle abstrait pour une machine LISP et de son implantation

Augustin Lux

THESE

présentée à

UNIVERSITE SCIENTIFIQUE ET MEDICALE DE GRENOBLE

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

POUR OBTENIR LE GRADE DE
DOCTEUR DE TROISIEME CYCLE
INFORMATIQUE

Augustin LUX

Etude d'un modèle abstrait pour une machine LISP et de son implantation.

Soutenue le 19 mars 1975 devant la Commission d'Examen

Président : L. BOLLIET

Examinateurs :
- M. GRIFFITHS
- C.H.A. KOSTER
- Y. SIRET
- G. VEILLON

UNIVERSITE SCIENTIFIQUE ET MEDICALE DE GRENOBLE		INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE
M. Michel SOUTIF	Présidents	M. Louis NEEL
M. Gabriel CAU	Vice-Présidents	MM. Lucien BONNETAIN Jean BENOIT

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

PROFESSEURS TITULAIRES

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

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

PROFESSEURS ASSOCIES

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

PROFESSEURS SANS CHAIRE

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

MAITRES DE CONFERENCES ET MAITRES DE CONFERENCES AGREGES

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

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

MAITRES DE CONFERENCES ASSOCIES

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

CHARGES DE FONCTIONS DE MAITRES DE CONFERENCES

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

Fait à Saint Martin d'Hères, OCTOBRE 1974.

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

PROFESSEURS TITULAIRES

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

PROFESSEUR ASSOCIE

M.	BOUDOURIS Georges	Radioélectricité

PROFESSEURS SANS CHAIRE

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

MAITRES DE CONFERENCES

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

MAITRE DE CONFERENCES ASSOCIE

M.	LANDAU Ioan Doré	Automatique

CHARGE DE FONCTIONS DE MAITRES DE CONFERENCES

M.	ANCEAU François	Mathématiques appliquées

Fait à Saint Martin d'Hères JANVIER 1974

Je tiens à remercier

Monsieur L. BOLLIET, Professeur à l'Institut Universitaire de Technologie de Grenoble, qui a bien voulu me faire l'honneur de présider le jury de cette thèse,

Monsieur G. VEILLON, Professeur à l'Institut National Polytechnique de Grenoble, qui dirige l'équipe au sein de laquelle ce travail a été effectué et qui au cours de nombreuses discussions m'a aidé par des suggestions et des critiques constructives,

Messieurs M. GRIFFITHS, Maître de Conférences à l'Université Scientifique et Médicale de Grenoble, et C.H.A. KOSTER, Professeur à l'Université Technique de Berlin, dont les critiques bienveillantes m'ont été précieuses,

Monsieur Y. SIRET, Ingénieur à l'Institut Laue Langevin, qui s'est intéressé à mon travail et qui a bien voulu le juger.

Je remercie également tous ceux, nombreux, qui m'ont aidé à améliorer la rédaction, en particulier Anne CHASTANIER qui a eu la patience de lire toutes les versions du manuscrit. Bien entendu, les erreurs qui restent sont à mettre entièrement à mon compte ainsi que les germanismes que le lecteur voudra bien excuser.

Je remercie Mesdames G. BICAIS, F. BLANC et M. TREVISAN qui ont assuré la dactylographie de ce texte avec une grande compétence et avec beaucoup de rapidité, Monsieur D. IGLESIAS et le personnel du service reproduction du C.I.C.G. qui ont assuré la réalisation matérielle de cette thèse dans des délais très rapides et avec le soin habituel.

Augustin LUX

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

Le travail présenté dans cette thèse a été abordé dans le cadre de plusieurs projets faisant appel au langage LISP.

Il s'est avéré que le système LISP/360 dont nous disposions avait de nombreux défauts dans le contexte du système CP/CMS, notamment en ce qui concerne les entrées/sorties et le traitement d'erreur.

Il n'a pas été possible d'effectuer des corrections limitées : LISP/360 est programmé en un seul module dont la structure interne porte les traces de nombreuses retouches. Toute nouvelle modification nécessite un assemblage long et entraîne souvent des problèmes ridicules d'adressabilité.

C'est pourquoi j'ai réorganisé la structure interne de LISP/360, en définissant des nouvelles conventions de liaison et d'adressabilité, pour rendre le système modulaire. Il s'est avéré que cela était possible en introduisant des simplifications qui ont en plus entraîné une légère amélioration des performances.

Ensuite, j'ai incorporé au système une interface complète avec CMS. Cette version de LISP, nommée LISP/CMS, est décrite dans un manuel d'utilisation. Elle offre maintenant de nombreuses possibilités qui n'existent pas dans LISP/360 et dont les plus importantes sont décrites dans la deuxième partie de cette thèse.

Bien que le compilateur, composant important de LISP/360,soit utilisé sans modification majeure en LISP/CMS, son étude a abouti à la conclusion qu'il est souhaitable de le réécrire. La réflexion sur les principes d'un nouveau compilateur a amené à la définition d'un compilateur extrêmement simple (I.2.4). Le code produit par ce dernier définit une machine (I.2.3) qui permet aussi la description d'une implantation de LISP relativement indépendante d'une machine physique, y compris la définition de l'interpréteur.

Parallèlement à ces développements, l'implantation de REDUCE qui est l'application la plus volumineuse de LISP/CMS a exercé une forte influence. D'une part elle a constitué un test sérieux de LISP/CMS, d'autre part elle a amené à la définition de RLISP. RLISP est le nom que j'ai donné à un sous-ensemble de REDUCE qui définit un langage assez proche de MLISP [41] . Ce langage, maintenant disponible en LISP/CMS, est sémantiquement équivalent à LISP et permet l'écriture de programmes LISP sous une forme inspirée d'ALGOL.

Avec la notion de S-graphe, élaborée pour des raisons didactiques, ces divers éléments permettent une nouvelle présentation de LISP, dans laquelle j'essaie d'éviter certaines insuffisances de la présentation classique. En effet, celle-ci, fondée sur la notion de S-expression et la définition de l'interpréteur en LISP, n'est pas satisfaisante pour plusieurs raisons :

La structure de données traitée en LISP est plus générale que les S-expressions; l'explication de certaines propriétés de LISP, en particulier des fonctions RPLACA, RPLACD, ne peut se faire qu'à l'aide de dessins qui ne rentrent pas dans le formalisme.

L'autodéfinition de LISP a des avantages - elle est une référence concise pour l'utilisateur, elle donne des possibilités de génération téléscopique à l'implantation - mais aussi des inconvénients bien connus : en particulier, elle est inutile pour le débutant, elle ne donne pas d'instructions à l'implanteur, et elle est incomplète du point de vue formel. Par exemple, ni le type de passage de paramètres ni la sémantique exacte de l'expression conditionnelle ne sont précisés.

La présentation proposée ici est donnée dans la première partie de l'exposé, qui est ainsi une synthèse de l'ensemble des notions abordées tout au long du travail. C'est une définition de LISP qui recouvre non seulement la syntaxe et la sémantique du langage, mais aussi certains aspects importants d'une implantation.

Le premier chapitre donne la définition des S-graphes, graphes binaires qui sont l'unique structure de données traitée en LISP, et la définition des sept fonctions CAR, CDR, CONS, ATOM, EQ, RPLACA, RPLACD qui forment la base du langage LISP.

Le langage est présenté au deuxième chapitre : d'abord par la définition de RLISP qui est considéré ici comme un format externe de LISP et qui est sémantiquement équivalent à l'écriture de LISP sous forme de listes. Un algorithme de traduction dans les deux sens est décrit.

Une première définition de la sémantique de LISP est donnée par l'interpréteur en LISP de Mc CARTHY.

Ensuite, je présente une Machine Intermédiaire pour l'implantation de LISP, MIL. Cette machine à piles représente un modèle abstrait d'une implantation de LISP et définit des notions utilisées dans la deuxième partie pour la description de LISP/CMS.

Le langage MIL permet une deuxième définition de l'interpréteur, dont la compréhension ne demande pas de connaissance préalable de LISP.

Le lien entre LISP et MIL est formalisé par un compilateur (2.4).

Ces trois éléments - interpréteur, machine MIL, compilateur - sont intimement liés entre eux, mais étant donné que le sujet de ce travail n'était pas l'étude de la nature de ces liens (qui se placent au niveau d'une théorie plus générale de la sémantique des langages de programmation) je me limiterai dans le paragraphe 2.4. à une présentation simple et descriptive du compilateur, accompagnée de quelques idées encore embryonnaires sur une poursuite possible des recherches.

Les développements exposés au deuxième chapitre sont appelés LISP pur; ils représentent un langage de programmation ayant des propriétés bien formalisables.

Le troisième chapitre introduit LISP1.5 qui est dérivé de LISP pur par une suite d'extensions, notamment les entrées/sorties (3.1) qui permettent de faire le lien entre les S-graphes et les S-expressions et l'adjonction d'une mémoire permanente (3.3).

Le quatrième chapitre esquisse une nouvelle technique d'implantation d'un système LISP qui procède en deux étapes : implantation de MIL en langage machine et implantation de LISP en MIL. La deuxième étape étant indépendante de la machine cette implantation est relativement transportable.

Alors que la première partie définit les grandes lignes d'une implantation de LISP, la deuxième partie présente des problèmes plus spécifiques dans le cadre d'une réalisation qu'est LISP/CMS. J'essaierai de dégager les principes, les détails étant décrits dans un manuel d'utilisation et dans une notice d'implantation.

Après un rappel bref de l'évolution et de la situation actuelle de LISP/CMS, contenu dans le premier chapitre, le deuxième chapitre aborde le problème de la gestion de la mémoire. La solution de ce problème est l'élément critique de l'implantation d'un langage, déterminant les applications possibles et les limitations inhérentes à un système. En LISP la question centrale de la gestion de la mémoire est celle du ramasse-miettes dont l'existence vient du fait que nous devons réaliser une source inépuisable d'éléments de listes dans une mémoire finie.

Une analyse qualitative du coût du ramasse-miettes montre que la solution choisie en LISP/360 est relativement mauvaise et qu'il en existe de meilleures.

Les possibilités offertes par LISP/360 pour l'introduction rapide de grandes masses de données sont très insuffisantes. Le troisième chapitre décrit les "outils de construction de système" disponibles en LISP/CMS en donnant plusieurs solutions à ce problème, chacune avec des caractéristiques (efficacité, souplesse d'emploi) différentes. La fonction LOAD qui commande le chargement d'un fichier de type TEXT constitue l'élément central de ces outils.

Le contenu de ce chapitre a un caractère relativement technique, mais il décrit des outils essentiels grâce auxquels LISP devient un langage de programmation effectivement utilisable.

Si l'on veut pleinement exploiter le caractère conversationnel de LISP, il faut intégrer au système des outils d'aide à la mise au point, dont un éditeur est le premier et le plus important. Le quatrième chapitre présente l'éditeur intégré à LISP/CMS à l'aide duquel la modification d'une fonction peut se faire de façon commode sans quitter l'environnement courant de LISP.

Le cinquième chapitre décrit très brièvement l'implantation du langage de calcul formel REDUCE.

PREMIERE PARTIE

"DEFINITION DE LISP"

1. S-graphes

1.1. Définitions

Un S-graphe est un graphe binaire dont la définition a été inspirée par les structures utilisées dans l'inplantation des listes en LISP. Intuitivement un noeud d'un S-graphe correspond a une cellule de mémoire contenant deux pointeurs.

Nous définissons d'abord la notion de classe de S-graphes qui correspond à une mémoire de pointeurs contenant un ensemble de S-graphes.

Définition : Une classe de S-graphes C est définie par :

* deux ensembles :

  A un ensemble d'atomes. Un atome est caractérisé par son nom qui est une chaîne de caractères quelconque.

  N l'ensemble des noeuds non-atomiques.

* et deux applications :

  car, cdr : $N \rightarrow N \cup A$

Une classe de S-graphes est donc un bi-graphe dont les feuilles sont les atomes.

L'ensemble s(r) des noeuds connectés à un noeud r par une chaîne de car et de cdr est récursivement défini par l'expression :

s(r) := if atom(r) then {r}
else {r} ∪ s(car(r)) ∪ s(cdr(r)).

Définition : Un S-graphe dans C avec la racine r est défini par

* une racine $r \in N \cup A$
* le sous-graphe complet de C ayant l'ensemble de noeuds s(r).

1.2. Représentation

1) Représentation graphique

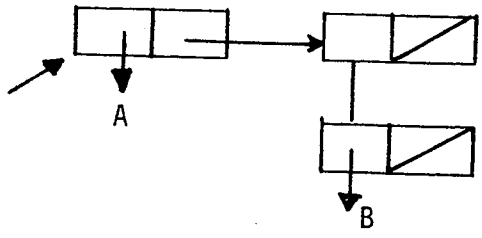
Nous appelons S-graphe atomique un S-graphe composé d'un seul atome. Il est représenté par un pointeur (arc) sur le nom de cet atome :

Un S-graphe non-atomique de racine r est désigné par un pointeur sur un rectangle à deux cases, la case gauche contenant le S-graphe de racine car(r), la case droite contenant le graphe de racine cdr(r).

2) Représentation linéaire

Etant donné le caractère encombrant de la représentation graphique nous sommes obligés d'utiliser les représentations linéaires définies au § 3.1, que nous introduisons ici à l'aide de quelques exemples :

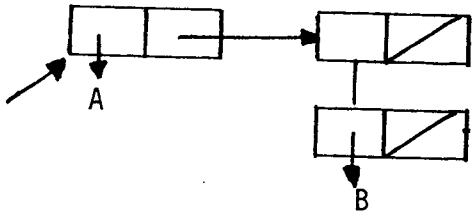
représentation graphique



représentation linéaire

La S-expression pointée
(A.((B.NIL).NIL))

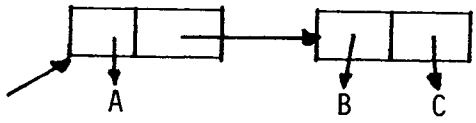
Les S-expressions pointées sont proches de la représentation graphique, mais cette notation n'est jamais utilisée car elle est extrêmement fastidieuse. Une représentation par listes est nettement plus claire :



La liste (A (B))

Cependant, l'écriture sous forme de listes est possible seulement si le cdr de chaque noeud non-atomique est lui-même non-atomique ou la liste vide (synonyme avec l'atome NIL).

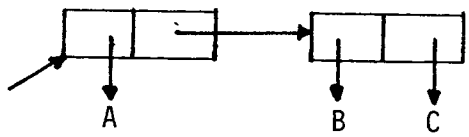
Le graphe



La S-expression pointée
(A . (B . C))

n'a pas une structure de listes.

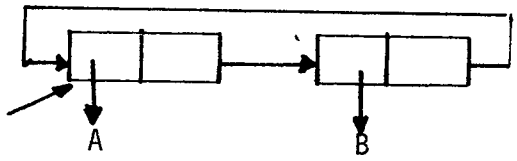
La structure linéaire la plus générale, les S-expressions, permet l'écriture de ce genre de graphes avec un minimum de points :



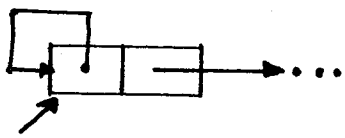
La S-expression (A B . C)

D'après les définitions du § 3.1, les S-expressions pointées et les listes sont des sous-ensembles de S-expressions.

Bien entendu, les S-graphes avec circuits, comme:



ou :



n'ont pas de représentation linéaire.

Remarques : 1) Notre nomenclature est différente de celle de Mac Carthy dans [30], où S-expression désigne ce que nous appelons S-expression pointée, alors que les S-expressions (dans notre sens) n'ont pas de nom propre. Nous avons changé la terminologie afin de la rendre conforme à l'usage.

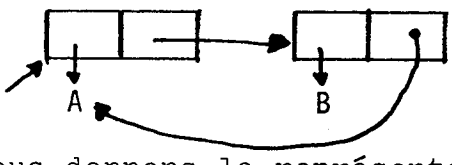
2) Les S-graphes sont une structure légèrement plus générale que les listes parce que :
a) ils peuvent comporter des circuits,
b) un cdr peut être atomique.

Il n'est donc pas tout à fait exact d'appeler LISP un langage de traitement de listes; en particulier, la fonction CONS (paragraphe suivant) est une fonction totale. Cependant, dans la majorité des applications on utilise seulement les listes.

1.3. L'unicité des atomes

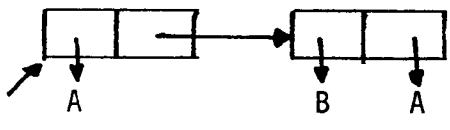
A étant un ensemble, chaque atome est unique et ne peut figurer qu'une fois dans un graphe. Comme cela nuit souvent à une représentation agréable des graphes nous prenons la liberté de dessiner un même atome plusieurs fois.

Par exemple, au lieu de :



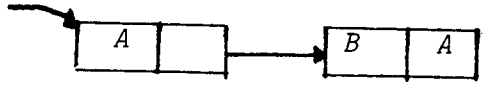
(A B . A)

nous donnons la représentation :



Remarques : *1) Nous insistons sur ce point pour une raison pratique : à l'unicité des atomes dans la théorie correspond leur implémentation à une adresse unique en machine, grâce à laquelle la fonction EQ peut être réalisée par une simple comparaison d'adresses.*

2) Notre représentation d'un S-graphe atomique est moins courante que la représentation sans pointeur, comme dans l'exemple :



La différence n'est pas très importante, mais nous préférons l'écriture avec pointeurs pour plusieurs raisons :

a) elle permet, au moins en principe, de tenir compte de l'unicité des atomes.

b) elle traduit mieux la situation en machine, où les noeuds non atomiques ne contiennent que des pointeurs.

1.4. Fonctions de base sur les S-graphes

1) Les applications car et cdr induisent des application sur les S-graphes.

CAR, CDR : {S-graphes non-atomiques} → {S-graphes}

où le CAR (CDR) d'un S-graphe de racine $r \in N$ est le S-graphe de racine car(r) (cdr(r)).

Dans la suite nous appellerons S l'ensemble de S-graphes, la classe C sera toujours sous-entendue, elle intervindra explicitement dans le ramasse-miettes seulement (II.2.1.2.) Par ailleurs, nous confondrons systématiquement les S-graphes avec les noeuds de C (les deux ensembles étant isomorphes). Ainsi, nous disons que

$$S = N \cup A$$

et nous ne distinguons pas CAR et car, CDR et cdr.

2) ATOM(X), EQ(X,Y)

Les fonctions ATOM et EQ peuvent prendre deux valeurs : ⇝T et ⇝NIL.

ATOM(X) = ⇝T si X ∈ A, ⇝NIL dans tous les autres cas.
EQ(X,Y) = ⇝T si X et Y sont le même élément de S
⇝NIL dans les autres cas.

Remarque : La définition de EQ est un peu problématique : Pour la cohérence du formalisme il serait préférable de définir EQ seulement pour des atomes, puisque c'est la seule définition possible pour les listes (§ 3.1.2.). Par contre, EQ est implémenté par une simple comparaison d'adresses et toujours utilisé comme fonction totale (ce qui simplifie la programmation). Cela nous a amené à donner la définition ci-dessus.

CAR, CDR, ATOM et EQ sont des fonctions à caractère statique, c'est-à-dire des fonctions au sens classique du terme.

Les autres primitives ont un caractère dynamique en ce sens qu'elles modifient les éléments de la structure algébrique même. L'effet de ces fonctions se prête difficilement à une analyse mathématique.

La définition d'une classe de S-graphes par quatre données élémentaires fournit une classification de ces fonctions :

*Modification de N :

La fonction CONS définit une extension de N et des applications car et cdr.

L'exécution de CONS(X,Y) rajoute un nouveau noeud non-atomique n à la classe C, qui est tel que :

car(n) = X
cdr(n) = Y

Le nouveau noeud constitue le résultat de CONS.

L'utilisation de CONS suppose donc une source inépuisable de noeuds.

Alors que l'extension de N est commandée explicitement par l'utilisateur, la diminution de N se fait implicitement et ne peut pas directement alimenter la source de noeuds. Dans un environnement à ressources limitées cela pose le problème de gestion de mémoire, approfondie au paragraphe II.2.1.

* Modification de car et cdr

Les fonctions RPLACA(x,y) et RPLACD(x,y) modifient les applications car et cdr au point x.

Après l'exécution de RPLACA(x,y) l'application car est telle que :

CAR(x) EQ y c'est-à-dire EQ(CAR(x),y) = vrai

et après l'exécution de RPLACD(x,y)

CDR(x) EQ y c'est-à-dire EQ(CDR(x),y) = vrai

Ces deux fonctions introduisent des effets de bord dans le langage, qui les rendent assez délicates à manipuler.

C'est pourquoi ces fonctions ne sont pas incluses dans LISP pur; elles détruiraient la plupart des propriétés mathématiques du langage sans en augmenter la puissance. Les cinq premières primitives suffisent déjà pour construire un langage universel.

* Modification de A.

En LISP pur on suppose fixe l'ensemble des atomes. Dans les systèmes réels la gestion de A est en général implicite: lors d'une lecture la fonction RDOBJ() (définie au paragraphe 3.1) rajoute des nouveaux atomes à A en fonction des données. La fonction GENSYM() génère un nouvel atome. Les atomes sont enlevés ou bien implicitement par le ramasse-miettes ou bien explicitement par la fonction REMOB(a).

2 - LISP pur

2.1 - Syntaxe

LISP pur est un langage de description de fonctions partielles sur des S-graph

$$f : S \to S$$

Le mécanisme de définition de fonctions est celui des λ-expressions, avec les cinq primitives CAR, CDR, CONS, ATOM, EQ, et a été développé par Mac Carthy [28,29,30]

Comme les S-graphes sont également utilisés pour représenter les fonctions, fonctions et données ont une même représentation en LISP. Cela est une caractéristique fondamentale qui implique la facilité du traitement de fonctions en tant qu'objets (par exemple la création, transformation, interprétation) et qui distingue LISP de la plupart des langages de haut niveau, parmi les 120 langages décrits dans le livre de J. Sammett (1) LISP est le seul à avoir cette propriété.

Cette représentation de LISP sous forme de S-graphes est bien adaptée au traitement en machine, mais elle a un inconvénient majeur : les programmes ont un caractère cryptique pour le néophyte, qui répugne beaucoup d'informaticiens, et la lisibilité est médiocre même pour le programmeur averti. La "jolie impression" (II.4.1) peut certes atténuer ce problème, mais elle ne le résout pas. C'est pourquoi on a toujours utilisé une autre représentation de fonctions LISP, parallèlement à la forme en listes : dans sa première définition [28] Mc Carthy parle de M-expressions (Meta-expressions), mais celles-ci étaient seulement une commodité d'écriture (leur syntaxe partage d'ailleurs beaucoup des défauts de la syntaxe des S-expressions). Elles n'étaient pas complètement formalisées ni traitées automatiquement.

1) Programming Languages : History and fundamentals.
Prentice Hall, 1969

L'idée d'utiliser effectivement un langage ressemblant à l'ALGOL comme langage d'entrée de LISP et de le traduire automatiquement en S-expressions fut réalisée nettement plus tard : d'abord pour LISP2 [1] et ensuite pour MLISP [41] qui est beaucoup utilisé aujourd'hui et qui a donné suite à des développements fort intéressants (MLISP2 [42]).

Nous suivons cet exemple en utilisant le langage RLISP pour la représentation externe desfonctions LISP.

2.1.1 - RLISP

Le langage RLISP tel que nous le présentons dans cette thèse est un sous-ensemb du langage utilisé dans la définition de REDUCE [19] et un parent proche de MLISP.
Il a la propriété importante que la traduction entre RLISP et LISP est facilemer possible dans les deux sens. Cela donne entre autre la possibilité d'utiliser ur éditeur fondé sur le même principe que l'éditeur LISP (II.4.2). Nous utilisons actuellement les deux traducteurs : le programme de traduction de RLISP en LISP disponsible sous CMS est extrait de la génération de REDUCE. Un traducteur très simple et élégant a récemment été donné par M. Nordstrom [36].
Le traducteur de LISP en RLISP est utilisé pour traduire des programmes en LISP déjà existant afin de leur donner une forme plus claire et concise.

La plupart des fonctions figurant dans cette thèse seront données en LISP et en RLISP ce qui permettra au lecteur une comparaison des deux formes.

2.1.2 - Grammaires

La page 17 donne la grammaire de LISP sous forme de listes -c'est la définition classique-, la page 16 contient la grammaire de RLISP. Ces deux grammaires sont sémantiquement équivalentes, et la traduction d'une fonction d'un format dans l'autre est définie par la règle (informelle) suivante :

Etant donné l'arbre syntaxique construit à l'aide d'une des deux grammaires, construire de haut en bas un arbre avec les règles de la même ligne de l'autre grammaire, en respectant la remarque 3 pour la traduction de la forme conditionnelle. Le mot des feuilles est la traduction.

Grammaire élémentaire de RLISP

```
 1  <forme> ::= <S-expression>|
 2              <atome> |
 3              <fonction>(<arglist>)|
 4              IF <forme> THEN <forme> ELSE <forme>
 5  <arglist> ::= <arg>{,<arg>}* | vide
 6  <arg>  ::= <forme> |
*7              FUNCTION (<fonction>)
 8  <fonction> ::= <atome> |
(9)                (LAMBDA (<atome>{,<atome>}*);<forme>)|
                                ou <vide>
10                 LABEL (<atome>,<fonction>)|
*11                <forme>
```

<S-expression> ::= <SXP>, voir § 3.1.4.

REMARQUES :

1) Une forme *correspond à la notion d'expression dans d'autres langages, c'est-à-dire une* forme *est une "expression de type S-graphe".*

*2) Les règles dont le numéro est précédé d'une * sont propres à LISP1.5 et n'existent pas en LISP pur.*

3) Traduction de la forme conditionnelle :

IF c1 THEN f1 {ELSE IF c_i then f_i}* $\left[\text{ELSE } f_n\right]$

(COND (c1 f1) $\{(c_i\ f_i)\}^*$ $\left[(T\ \ f_n)\right]$

facultatif

Grammaire LISP

			Commentaires
1	<forme>	::= (QUOTE<S-expression>) |	*constante*
2		<atome> |	*variable*
3		(<fonction><arglist>) |	*appel de fonction*
4		(COND {(<forme><forme>)}*)	*forme conditionnelle*
5	<arglist>	::= {<arg>}*	
6	<arg>	::= <forme> |	*cas normal*
*7		(FUNCTION <fonction>)	*argument fonctionnel*
8	<fonction>	::= <atome> |	*nom de fonction*
9		(LAMBDA ({<atome>}*)<forme>) |	*définition par λ-expression*
10		(LABEL<atome><fonction>) |	*liaison nom - fonction*
*11		<forme>	*rarement utilisé*

Extensions de RLISP

<forme> ::= (<forme>)
<bloc>
<forme spéciale>

La sémantique des BLOCs est définie au paragraphe 3.5.

<bloc> ::= BEGIN{<dcl>}* {<inst>}* END
<DCL> ::= SCALAR <atome>{,<atome>}*;
<inst> ::= <atome>: *étiquette*
<forme>; *; avant END facultatif*
<forme spéciale> *défini par l'utilisateur (par exemple WHILE)*

Format d'un bloc en LISP

(PROG ({<atome>}*){<forme>}*)
dcl variables locales

Une <forme> atomique définit une étiquette.

2.1.3. - Extensions syntaxiques de RLISP

La page 16 donne seulement un noyau de la syntaxe de RLISP. Des extensions syntaxiques seront introduites à plusieurs occasions par la suite.
Voici deux extensions qui diminuent énormément le nombre de parenthèses :

1) Préfixes : si une fonction n'a qu'un argument, celui-ci peut ne pas être entouré de parenthèses. Au lieu de

```
ATOM(CAR(X))
```

on peut écrire

```
ATOM CAR X.
```

2) Expressions infixées : pour un certain nombre de fonctions (modifiable par le programmeur) il existe une forme infixée. C'est en particulier le cas des fonctions suivantes :

fonction	forme infixée
CONS	.
EQ	EQ
EQUAL	=
SETQ	:=

et de toutes les fonctions arithmétiques habituelles.

Les priorités sont définies par la liste suivante :

(:= OR AND NOT MEMBER = NEQ EQ >= > <= < + - * / ** .)

L'ordre d'évaluation des opérateurs est de gauche à droite, sauf pour . (CONS) où il va de droite à gauche.

Exemples :

RLISP	*LISP*
X.Y.NIL	(CONS X (CONS Y NIL))
CAR X EQ CAR Y	(EQ (CAR X) (CAR Y))
A + B * C	(PLUS A (TIMES B C))

En mettant des parenthèses on peut modifier cet ordre :

(A+B) * C)	(TIMES (PLUS A B) C)

On constate que les parenthèses ont une signification entièrement différente en LISP et en RLISP :

* en LISP elles indiquent la structure d'une S-expression
* en RLISP elles peuvent
  a) explicitement délimiter des sous-expressions
  b) délimiter des listes d'arguments.

Remarque : Alors que NIL et ( ) sont synonymes en LISP, ils ne le sont pas en RLISP où NIL représente une expression et ( ) une liste vide :
FN NIL est traduit en (FN NIL) (appel d'une fonction à un argument qui a la valeur NIL)
alors que FN( ) est traduit en (FN) (appel d'une fonction sans argument).

2.2 - La fonction universelle

La sémantique de LISP est formellement définie par l'interpréteur APPLY (aussi appelé fonction universelle) qui peut lui-même être écrit en LISP. Une telle "autodéfinition" de LISP, donnée p.20,109 est utile pour la documentation et même pour l'implémentation (voir 4), mais elle est incomplète du point de vue informatique et certainement inutile pour le débutant.

Le chapitre 2.3 donnera une deuxième définition de l'interpréteur dans un langage extérieur à LISP, dont la compréhension ne demande pas de connaissance préalable de LISP. Nous nous contentons ici de donner quelques commentaires facilitant la compréhension globale de l'interpréteur.

Forme conditionnelle :
L'expression conditionnelle est bien connue (du moins dans sa forme RLISP) ; il faut toutefois préciser quand une condition est vraie puisque la notion d'expression booléenne n'existe pas en LISP. L'interpréteur ne nous renseigne pas à ce sujet. La règle est :

- l'atome NIL représente la valeur faux
- toute autre S-graphe représente la valeur vrai.

Bien entendu, cette définition est arbitraire, mais elle rend très aisée l'uti-

```
SYMBOLIC PROCEDURE APPLY(FN,X,A);
IF ATOM FN THEN IF FN EQ 'CAR THEN CAAR X
    ELSE IF FN EQ 'CDR THEN CDAR X
    ELSE IF FN EQ 'CONS THEN CAR X . CADR X
    ELSE IF FN EQ 'ATOM THEN ATOM CAR X
    ELSE IF FN EQ 'EQ THEN CAR X EQ CADR X
    ELSE APPLY(EVAL(FN,A),X,A)
  ELSE IF CAR FN EQ 'LAMBDA THEN EVAL(CADDR FN,PAIRLIS(CADR FN,X,A))
  ELSE IF CAR FN EQ 'LABEL THEN APPLY(CADDR FN,X,(CADR FN . CADDR FN).
A ) ;

SYMBOLIC PROCEDURE EVAL(E,A);
IF ATOM E THEN CDR ASSOC(E,A)
  ELSE IF ATOM CAR E THEN IF CAR E EQ 'QUOTE THEN CADR E
    ELSE IF CAR E EQ 'COND THEN EVCON(CDR E,A)
    ELSE APPLY(CAR E,EVLIS(CDR E,A),A)
  ELSE APPLY(CAR E,EVLIS(CDR E,A),A) ;

SYMBOLIC PROCEDURE EVLIS(M,A);
IF NULL M THEN NIL ELSE EVAL(CAR M,A) . EVLIS(CDR M,A) ;

SYMBOLIC PROCEDURE EVCON(C,A);
IF NULL C THEN NIL
   ELSE IF EVAL(CAAR C,A) THEN EVAL(CADAR C,A)
   ELSE EVCON(CDR C,A) ;
```

L'interpréteur LISP pur de Mc CARTHY [30] en RLISP.

(L'annexe 1 donne le même programme en LISP)

```
SYMBOLIC PROCEDURE CAPPLY(FN);
IF ATOM FN THEN IF FN EQ 'CAR THEN ATTACH('(CAR))
    ELSE IF FN EQ 'CDR THEN ATTACH('(CDR))
    ELSE IF FN EQ 'CONS THEN ATTACH('(CONS))
    ELSE IF FN EQ 'ATOM THEN ATTACH('(ATOM))
    ELSE IF FN EQ 'EQ THEN ATTACH('(EQ))
    ELSE ATTACH(LIST('LINK,FN))
  ELSE IF CAR FN EQ 'LAMBDA THEN
    BEGIN
       IF CADR FN THEN ATTACH('ARGS . CADR FN) ;
       CEVAL(CADDR FN);
       IF CADR FN THEN ATTACH(LIST('FREE,LENGTH(CADR FN))) ELSE NIL
    END
  ELSE IF CAR FN EQ 'LABEL THEN BEGIN ATTACH(CADR FN);CAPPLY(CADDR FN)
END ;

SYMBOLIC PROCEDURE CEVAL(E);
IF ATOM E THEN ATTACH(LIST('PUSHV,E))
  ELSE IF ATOM(CAR E) THEN
         IF CAR E EQ 'QUOTE THEN ATTACH(LIST('PUSHQ,CADR E))
    ELSE IF CAR E EQ 'COND THEN (LAMBDA (ETI);
           BEGIN CEVCON(CDR E,ETI); ATTACH ETI END)(GENSYM())
    ELSE BEGIN CEVLIS(CDR E);CAPPLY(CAR E) END
  ELSE BEGIN CEVLIS(CDR E);CAPPLY(CAR E) END ;

SYMBOLIC PROCEDURE CEVLIS(M);
IF NULL M THEN NIL ELSE BEGIN CEVAL(CAR M);CEVLIS(CDR M) END ;

SYMBOLIC PROCEDURE CEVCON(C,ETI);
IF NULL C THEN NIL
  ELSE IF CAAR C EQ 'T THEN CEVAL(CADAR C)
  ELSE (LAMBDA (X);
    BEGIN
       CEVAL(CAAR C);
       ATTACH(LIST('BRNIL,X));
       CEVAL(CADAR C);
       ATTACH(LIST('GO,ETI));
       ATTACH X;
       CEVCON(CDR C,ETI)
    END)(GENSYM()) ;

SYMBOLIC PROCEDURE ATTACH(I);
LISTING := APPEND1(LISTING,I);

SYMBOLIC PROCEDURE COMPILE(N);
  COMMENT N EST LE NOM DE LA FONCTION A COMPILER;
  BEGIN SCALAR LISTING;
  COMMENT LISTING SERA LE RESULTAT DE LA COMPILATION;
     LISTING := N . NIL;
     CAPPLY(GET(N,'EXPR));
     COMMENT ON COMPILE LA DEFINITION DE LA FONCTION;
     ATTACH('(RETURN));
     LPRIN LISTING;  COMMENT IMPRESSION;
     RETURN LISTING
  END;
```

Le compilateur du paragraphe 2.4 écrit en RLISP.

(L'annexe II donne le même programme en LISP)

lisation des expressions conditionnelles, puisque toute expression peut servir de condition ; la comparaison (très fréquente) avec la liste vide est simplifiée au lieu de :

IF X EQ NIL THEN A ELSE B

on peut écrire

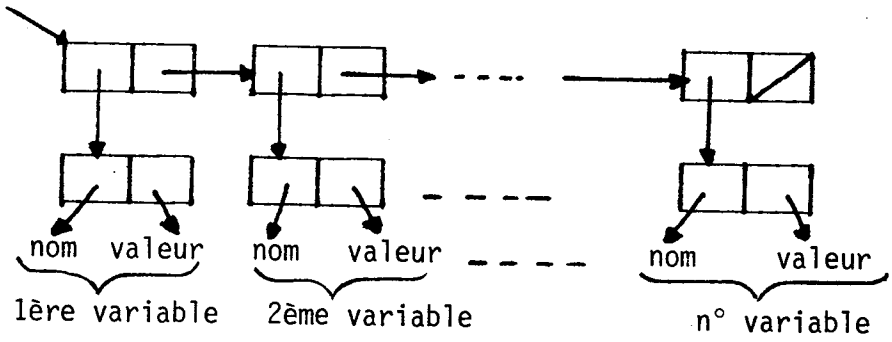
IF X THEN B ELSE A

LAMBDA et LABEL

L'atome LAMBDA caractérise la définition d'une fonction (règle (9)), qui consiste en la donnée d'une liste des noms des paramètres et d'une expression, mais une telle fonction n'a pas de nom. Un nom de fonction est introduit par LABEL. Cette particularité de LISP pur n'est jamais utilisée dans la pratique, où on définit les fonctions de façon permanente dans la liste de propriété du nom (voir 3.3.2.).

Contexte et ALIST :

Lors de l'interprétation le symbole LAMBDA lie des variables à des valeurs. Ceci se fait dans une liste d'association ou ALIST. qui a la forme



et qui est créée par la fonction PAIRLIS.

La fonction ASSOC(X,A) délivre le couple associé au nom X dans la ALIST A.

Exemple :

ASSOC(X, ((A.B) (X.Y) (X.Z))) => (X.Y)

Si un nom figure plusieurs fois dans la ALIST, la première occurence est choisie. Ainsi la ALIST forme le contexte d'une exécution ; lors de l'appel hiérarchique de fonctions son fonctionnement ressemble à celui d'une pile. La situation plus complexe en LISP.1.5 est décrite au chapitre 3.

L'argument A des quatre fonctions de l'interpréteur est un contexte d'évaluation sous forme d'une ALIST.
La structure de ces fonctions reflète la structure de la grammaire S de la page 17.

* la fonction EVAL(E,A) calcule la valeur de la <FORME>E dans le contexte A.
* la fonction APPLY(FN,X,A) calcule la valeur de la <FONCTION> FN avec la liste d'arguments X dans le contexte A.
* EVLIS et EVCON calculent la valeur d'une liste d'arguments et d'une forme conditionnelle resp., correspondant aux règles (5) et (4) de la grammaire.

Chaque test fait par ces fonctions détermine une alternative de la grammaire (indiquée à gauche du programme) et l'action effectuée définit la sémantique de la règle.

Par exemple, une <FORME> qui est un atome -règle (2)- est un identificateur dont la valeur est à chercher sur la ALIST. Si le premier élément d'une <FORME> est QUOTE -règle (1)- on a affaire à une constante donnée par le deuxième élément de la forme.

2.3 - La machine MIL

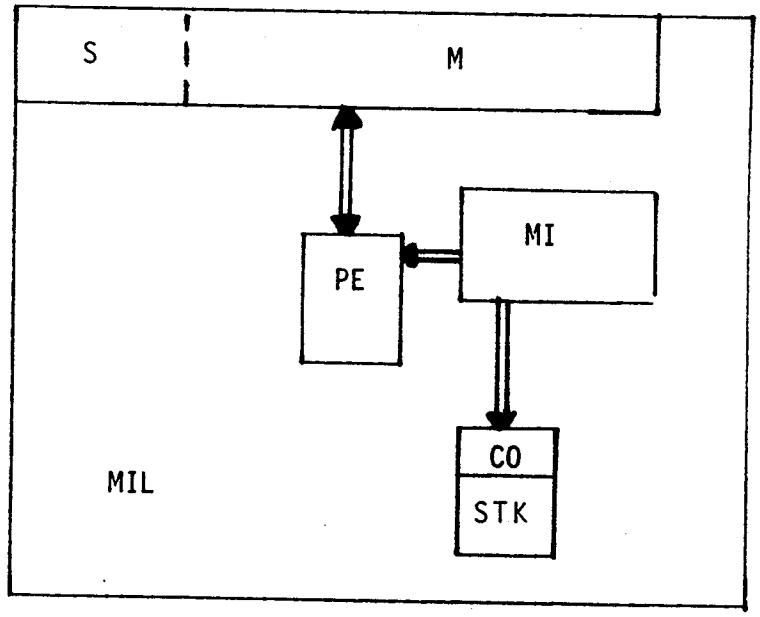
Dans ce chapitre, nous définissons une Machine Intermédiaire pour l'implantation de LISP, MIL, qui permet d'exprimer toute fonction LISP ; en particulier nous pouvons programmer la fonction universelle APPLY et ainsi réaliser une machine LISP.

Du point de vue logique nous présentons avec cette machine un deuxième formalisme universel, équivalent à LISP ; chacun des deux formalismes permet facilement d'exprimer l'autre. La deuxième définition ne se justifie donc pas par des raisons purement théoriques, mais seulement par des motifs pratiques. La machine MIL est un formalisme plus simple en ce sens qu'il est plus proche de la structure d'un ordinateur (et aussi de la machine de Turing) et qu'il permet ainsi de modeler un interpréteur LISP avec des primitives faciles à réaliser sur une machine réelle. La machine MIL fournit donc non seulement une définition de LISP dans un autre langage, mais aussi un modèle d'implantation d'un interpréteur LISP.

2.3.1 - Organisation globale

La machine MIL est définie par un ensemble de dispositifs physiques (Fig. 1) et par un ensemble d'actions (§ 3.3).

FIGURE 1 : Eléments de mémorisation et principaux chemins de données de MIL



	LISP	MIL
acteur	interpréteur	partie contrôle
donnée interprêtée	S-graphe	instruction
action élémentaire	interprétation d'un "doublet" fonction-liste d'arguments	cycle de base : exécution d'une instruction et mise à jour de CØ
résultat d'une action	S-graphe	nouvel état de la machine (M,PE,STK)
contexte d'exécution	LISP pur : ALIST LISP 1.5 : ALIST + banque de données	état défini par (M,PE,STK)
liaison de variables	associative (par nom) dans la ALIST	emplacement fixe dans PE

Figure 2 : Comparaison de la machine LISP et de la machine MIL

La machine est organisée autour de deux mémoires et deux piles :

* une mémoire M contenant une classe de S-graphes et une source de noeuds S.
* Une mémoire MI contenant des instructions.
* Une pile PE contenant des pointeurs sur des S-graphes dans M.
* Une pile STK contenant des adresses dans MI. Le sommet de STK est le compteur ordinal CO.

La machine fonctionne suivant le principe classique : un cycle de base est constitué par l'exécution de l'instruction dans MI adressée par CO, et CO est mis à jour. Plus précisément, l'exécution d'une instruction est son interprétation par la partie contrôle de MIL, qui n'est pas représentée dans la figure 1.
Il y a donc une analogie de la machine MIL avec une machine LISP, résumée dans la figure 2.

Une fonction dans MI (ou programme) est définie par la donnée simultanée d'un ensemble d'instructions contenues dans MI et d'une valeur de CO (point d'entrée) (l'ensemble des instructions consiste en toutes les instructions par lesquelles peut passer l'exécution à partir du point d'entrée).

Etant donnée une fonction F, une exécution de F est une suite de cycles de base partant du point d'entrée. Une exécution peut être finie ou non.

PE est une pile qui assure pour les fonctions dans MI un rôle analogue à celui de la ALIST pour les fonctions interprétées par LISP : elle définit les valeurs des variables. Il y a cependant une différence importante : alors que dans la ALIST la valeur d'une variable est liée à un nom, dans PE elle est liée à un emplacement fixe et ne porte pas de nom à l'exécution (le nom est traduit en une adresse par l'assembleur, paragraphe suivant).

Le rôle de contexte d'exécution joué par PE est étroitement lié au rôle dans le passage de paramètres selon les conventions de liaison qui définissent les notions d'argument et de résultat d'une fonction dans MI : quand une fonction reçoit le contrôle, elle trouve ses arguments au sommet de la pile PE. A la

sortie (instruction RETURN), les arguments doivent avoir été enlevés et remplacés par le résultat. Ceci est symbolisé par le schéma donné pour l'instruction LINK ci-après.

Variables libres

La différence entre la ALIST et la pile PE a une conséquence importante pour la compilation (2.4) : contrairement à une fonction LISP un programme en MIL ne peut contenir des variables libres. (Une variable libre est une variable qui n'est pas liée dans la fonction dans laquelle elle est utilisée. Son adresse n'est pas connue à la compilation). Le problème est résolu en LISP1.5 (§ 3.3.5). Cette limitation n'est pas gênante, d'autant plus que les variables libres ne sont pas de bon style en programmation. Elle n'est pas non plus en contradiction avec le caractère universel de MIL : si l'on veut programmer une fonction avec des variables libres, il suffit de modifier l'algorithme (pour les éliminer).

2.3.2. Instructions

En fonction de leur effet sur la machine, nous répartissons les instructions de MIL en trois classes :

1) gestion de PE
2) gestion de STK, y compris CO } instructions de contrôle
3) transfert d'information M <=> PE (fonctions primitives)

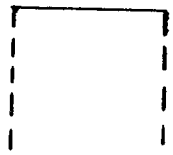
Nous représentons ces instructions dans un langage d'assemblage afin de leur donner une forme facile à comprendre et pour éviter des détails techniques : nous faisons notamment abstraction du codage des adresses, en utilisant des adresses symboliques (étiquettes) et ceci non seulement pour référencer des instructions, comme c'est le cas de tous les assembleurs, mais

aussi pour désigner les éléments de PE. Les noms des éléments de la pile sont définis par la pseudo-instruction suivante :

ARGS n1 ...ni On déclare que les i éléments au sommet de PE portent les étiquettes n1 ... ni.

Ces étiquettes ont un caractère temporaire et sont enlevées par l'instruction FREE (ci-après).

La sémantique de la plupart des instructions est définie par leur effet formel (la notion d'exécution formelle est reprise au § 2.3.5) sur PE, donné sous forme d'un schéma dans lequel le sous-schéma



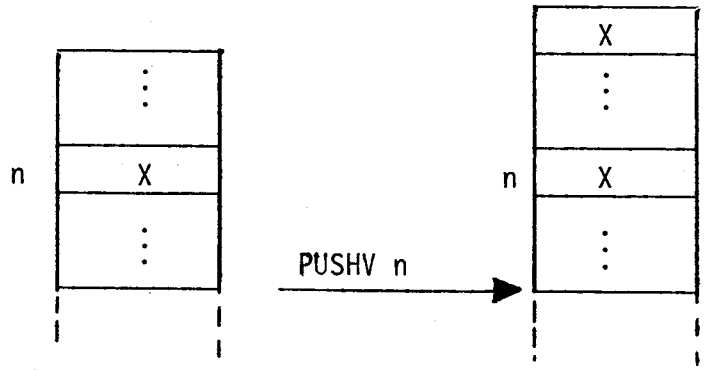
désigne la partie inférieure de la pile qui n'est pas affectée par l'instruction. Cette partie inférieure peut être absente alors que les cases dessinées avec des traits pleins

sont obligatoirement présentes.

Si un élément de la pile porte une étiquette lors de l'assemblage, nous l'indiquons à gauche de la case correspondante.

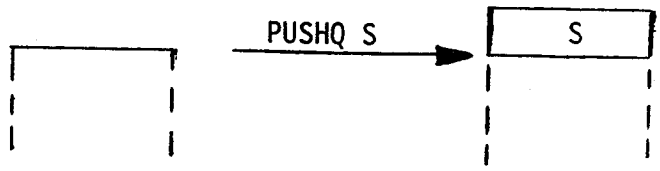
2.3.2.1. Instructions de gestion de PE

PUSHV n la valeur de l'élément de PE portant l'étiquette n est recopiée au sommet de PE :



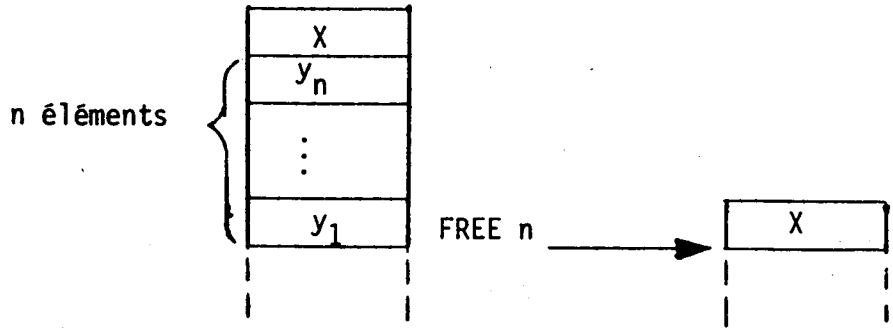
Remarque : *L'étiquette n est transformée par l'assembleur en une adresse qui peut être exprimée par un déplacement par rapport au sommet de PE ou par rapport à une base.*

PUSHQ s Empile (un pointeur sur) la S-expression s, qui est une constante. Cette instruction est la seule à effectuer un transfert d'information de MI vers PE.



FREE n Les n éléments juste en dessous du sommet de PE sont enlevés, le sommet descend de n positions .

Cette instruction sert au dépilement des arguments d'une fonction, exigé par les conventions de liaison.



Remarque : *Cette instruction est aussi l'inverse de la pseudo-instruction ARGS. Si un élément dépilé portait une étiquette, celle-ci disparaît.*

2.3.2.2. Instructions de gestion de STK

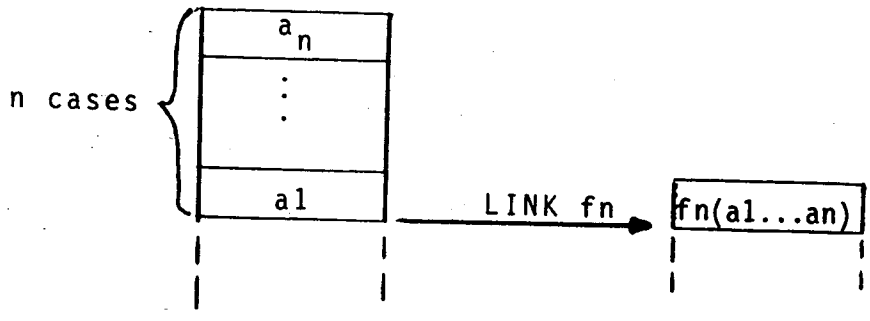
Les instructions de branchement affectent seulement la valeur de CO :

BRNIL eti Branchement conditionnel à l'adresse eti; on saute si le sommet de PE est égal à NIL (sinon on continue en séquence). Le sommet est dépilé.

GO eti Branchement inconditionnel à l'adresse eti.
L'instruction GO est redondante, puisque GO ETI peut être remplacé par PUSHQ NIL
BRNIL ETI

Les instructions d'appel de fonction empilent et dépilent le compteur ordinal dans STK.

LINK fn L'adresse de l'instruction suivante est empilée dans STK et le contrôle est passé à la fonction fn.
En LISP pur le point d'entrée de la fonction fn est définie par une étiquette d'assemblage. La situation est plus complexe en LISP1.5 (§ 3.3.4). En tenant compcompte des conventions de liaison, l'effet formel de l'instruction LINK sur PE est donné par le schéma :



RETURN Retour au programme appelant :

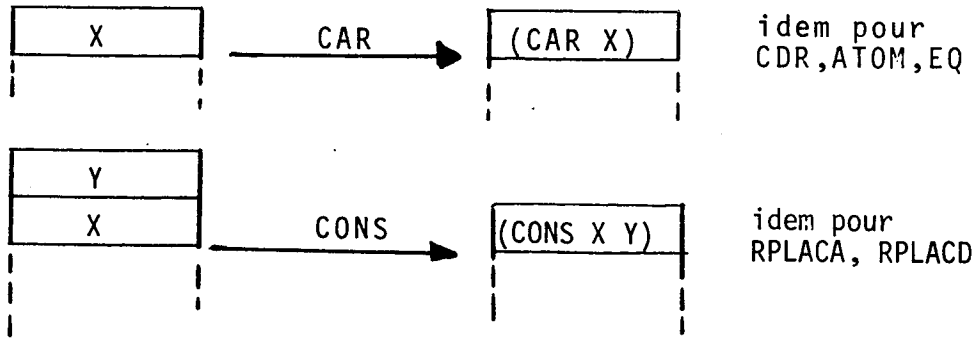
CO <= pull(STK)

On revient à l'instruction qui suit le dernier LINK

Si STK est vide la machine s'arrête.

2.3.2.3 Les instructions LISP

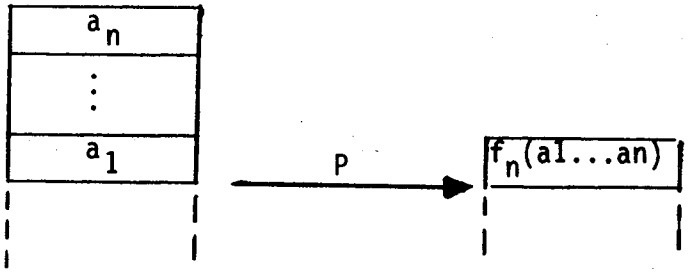
Chaque fonction de base de LISP constitue une instruction de transfert de données entre M et PE. Bien entendu, ces fonctions respectent les conventions de liaison.



2.3.3. Programmation d'une fonction LISP

Nous disons qu'une fonction LISP fn à n paramètres est réalisée par un programme P pour la machine MIL si toute exécution de P avec des paramètres a1...an, conforme aux conventions de liaison, donne un résultat isomorphe à fn(a1...an), lorsque cela est défini.

Nous exprimons cela par le schéma suivant, symbolisant l'effet formel sur la pile :



Remarque : *Si fn(a1 ...an) est indéfini, l'exécution de P peut ne pas s'arrêter ou donner un résultat aléatoire. (Un système comme LISP1.5 peut aussi détecter une erreur).*

Exemple : La fonction

```
PAIR(A,B) := IF A THEN (CAR A . CAR B) . PAIR(CDR A, CDR B)
                  ELSE NIL;
```

qui s'écrit en LISP :

```
(LABEL PAIR  (LAMBDA (A B) (COND
                  (A (CONS
                        (CONS (CAR A) (CAR B))
                        (PAIR (CDR A) (CDR B))))
                  (T NIL))))
```

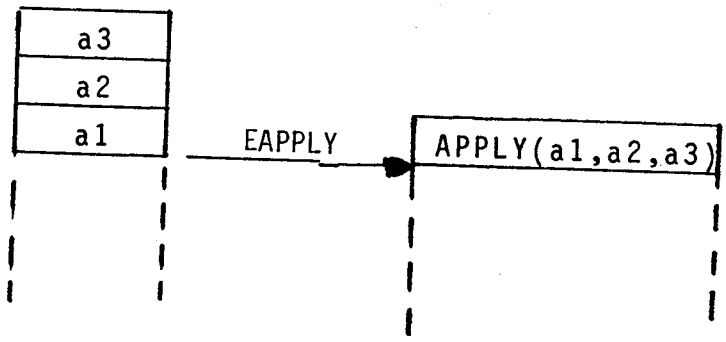
est réalisée par le programme suivant :

```
                  EPAIR (ARGS A B)
                        (PUSHV A)
                        (BRNIL SYMB2)
                        (PUSHV A)
                        (CAR)
                        (PUSHV B)
                        (CAR)
                        (CONS)
                        (PUSHV A)
                        (CDR)
                        (PUSHV B)
                        (CDR)
                        (LINK PAIR)
                        (CONS)
                        (GO SYMB1)
                  SYMB2 (PUSHQ NIL)
                  SYMB1 (FREE 2)
                        (RETURN)
```

2.3.4. Réalisation d'une machine LISP

Les pages 33, 34 donnent un programme qui réalise la fonction universelle dont il constitue une deuxième définition.

En utilisant les définitions du paragraphe précédent, nous énonçons donc la propriété :



La technique de validation ci-dessous permet de vérifier cette propriété. Pour la démontrer il faudrait définir un méta-formalisme dans lequel on peut définir un programme écrit en LISP et un programme écrit en langage d'assembleur. Mais cela déplacerait le problème sans le résoudre car on ne pourra pas démontrer la validité du codage des programmes dans le méta-formalisme.

```
EAPPLY  (ARGS FN X A)
        (PUSHV FN)
        (ATOM)
        (BRNIL SYMB2)
        (PUSHV FN)
        (PUSHQ CAR)
        (EQ)
        (BRNIL SYMB4)
        (PUSHV X)
        (CAR)
        (CAR)
        (GO SYMB3)
SYMB4   (PUSHV FN)
        (PUSHQ CDR)
        (EQ)
        (BRNIL SYMB5)
        (PUSHV X)
        (CAR)
        (CDR)
        (GO SYMB3)
SYMB5   (PUSHV FN)
        (PUSHQ CONS)
        (EQ)
        (BRNIL SYMB6)
        (PUSHV X)
        (CAR)
        (PUSHV X)
        (CDR)
        (CAR)
        (CONS)
        (GO SYMB3)
SYMB6   (PUSHV FN)
        (PUSHQ ATOM)
        (EQ)
        (BRNIL SYMB7)
        (PUSHV X)
        (CAR)
        (ATOM)
        (GO SYMB3)
SYMB7   (PUSHV FN)
        (PUSHQ EQ)
        (EQ)
        (BRNIL SYMB8)
        (PUSHV X)
        (CAR)
        (PUSHV X)
        (CDR)
        (CAR)
        (EQ)
        (GO SYMB3)
SYMB8   (PUSHV FN)
        (PUSHV A)
        (LINK EVAL)
        (PUSHV X)
        (PUSHV A)
        (LINK APPLY)
SYMB3   (GO SYMB1)
SYMB2   (PUSHV FN)
        (CAR)
        (PUSHQ LAMBDA)
        (EQ)
        (BRNIL SYMB9)
        (PUSHV FN)
        (CDR)
        (CDR)
        (CAR)
        (PUSHV FN)
        (CDR)
        (CAR)
        (PUSHV X)
        (PUSHV A)
        (LINK PAIRLIS)
        (LINK EVAL)
        (GO SYMB1)
SYMB9   (PUSHV FN)
        (CAR)
        (PUSHQ LABEL)
        (EQ)
        (BRNIL SYMB10)
        (PUSHV FN)
        (CDR)
        (CDR)
        (CAR)
        (PUSHV X)
        (PUSHV FN)
        (CDR)
        (CAR)
        (PUSHV FN)
        (CDR)
        (CDR)
        (CAR)
        (CONS)
        (PUSHV A)
        (CONS)
        (LINK APPLY)
        (GO SYMB1)
SYMB10
SYMB1   (FREE 3)
        (RETURN)
```

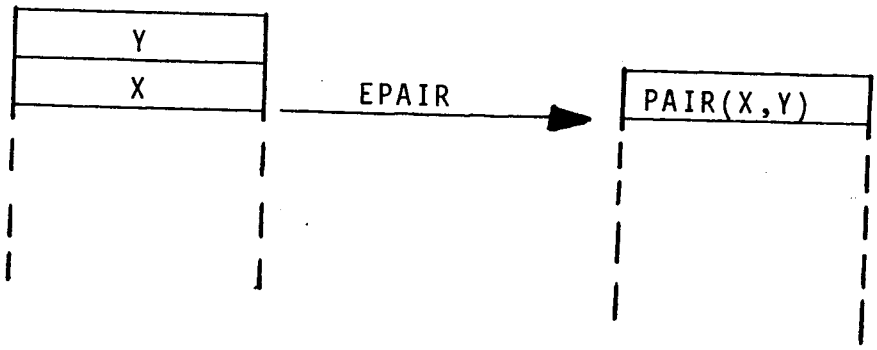
```
EVAL    (ARGS E A)
        (PUSHV E)
        (ATOM)
        (BRNIL SYMB12)
        (PUSHV E)
        (PUSHV A)
        (LINK ASSOC)
        (CDR)
        (GO SYMB11)
SYMB12  (PUSHV E)
        (CAR)
        (ATOM)
        (BRNIL SYMB13)
        (PUSHV E)
        (CAR)
        (PUSHQ QUOTE)
        (EQ)
        (BRNIL SYMB15)
        (PUSHV E)
        (CDR)
        (CAR)
        (GO SYMB14)
SYMB15  (PUSHV E)
        (CAR)
        (PUSHQ COND)
        (EQ)
        (BRNIL SYMB16)
        (PUSHV E)
        (CDR)
        (PUSHV A)
        (LINK EVCON)
        (GO SYMB14)
SYMB16  (PUSHV E)
        (CAR)
        (PUSHV E)
        (CDR)
        (PUSHV A)
        (LINK EVLIS)
        (PUSHV A)
        (LINK APPLY)
SYMB14  (GO SYMB11)
SYMB13  (PUSHV E)
        (CAR)
        (PUSHV E)
        (CDR)
        (PUSHV A)
        (LINK EVLIS)
        (PUSHV A)
        (LINK APPLY)
SYMB11  (FREE 2)
        (RETURN)
```

```
EVLIS   (ARGS M A)
        (PUSHV M)
        (LINK NULL)
        (BRNIL SYMB18)
        (PUSHV NIL)
        (GO SYMB17)
SYMB18  (PUSHV M)
        (CAR)
        (PUSHV A)
        (LINK EVAL)
        (PUSHV M)
        (CDR)
        (PUSHV A)
        (LINK EVLIS)
        (CONS)
SYMB17  (FREE 2)
        (RETURN)

EVCON   (ARGS C A)
        (PUSHV C)
        (CAR)
        (CAR)
        (PUSHV A)
        (LINK EVAL)
        (BRNIL SYMB20)
        (PUSHV C)
        (CAR)
        (CDR)
        (CAR)
        (PUSHV A)
        (LINK EVAL)
        (GO SYMB19)
SYMB20  (PUSHV C)
        (CDR)
        (PUSHV A)
        (LINK EVCON)
SYMB19  (FREE 2)
        (RETURN)
```

2.3.5. Validation d'un programme en langage machine

Il est facile de vérifier sur un exemple que le programme donné pour PAIR fournit le bon résultat; mais vu la façon dont les instructions ont été définies on peut démontrer que ce programme réalise effectivement la fonction PAIR, c'est-à-dire :



La démonstration se fait par induction récursive et exécution formelle.

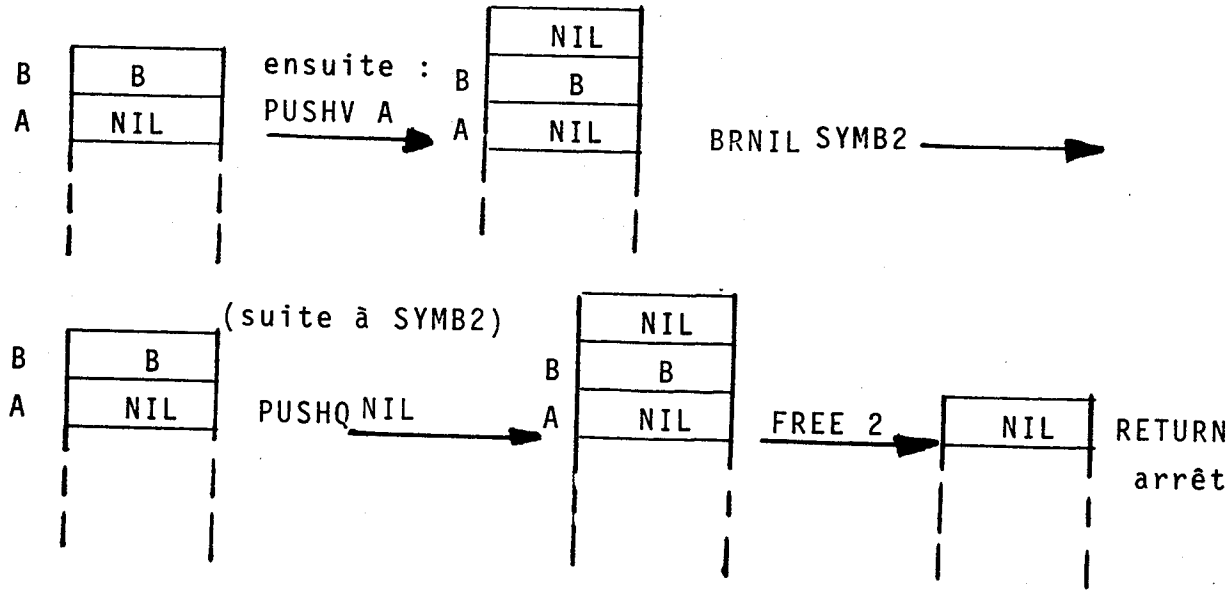
Premier pas de l'induction : Le programme donne le bon résultat si

A = NIL

Nous avons défini chaque instruction par son effet "formel" sur la pile. Par exécution formelle nous entendons une sorte d'exécution du programme avec des valeurs indéfinies formelles pour certains arguments. Les appels de fonction ne sont pas remplacés par des résultats (ils restent formels). D'autre part nous faisons des hypothèses sur des variables pour décider le sens des branchements conditionnels.

Cette notion d'exécution formelle est bien connue en preuve de programmes (c'est aussi la base de la mise au point manuelle).

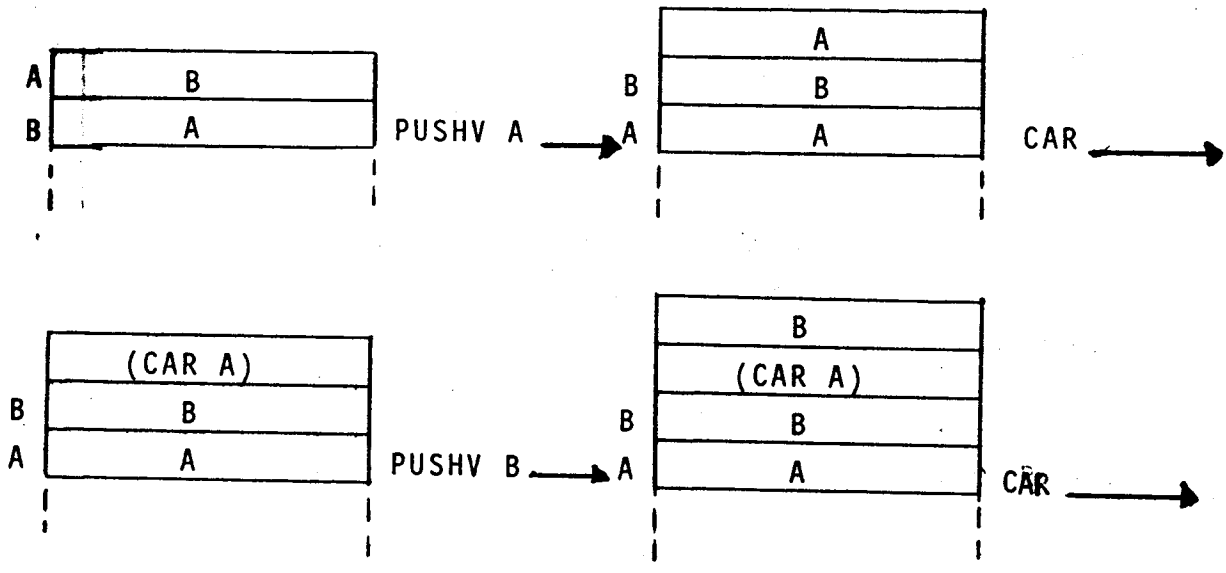
Voici ce que donne l'exécution formelle du programme PAIR avec l'hypothèse A = NIL. Au début PE doit avoir la forme :

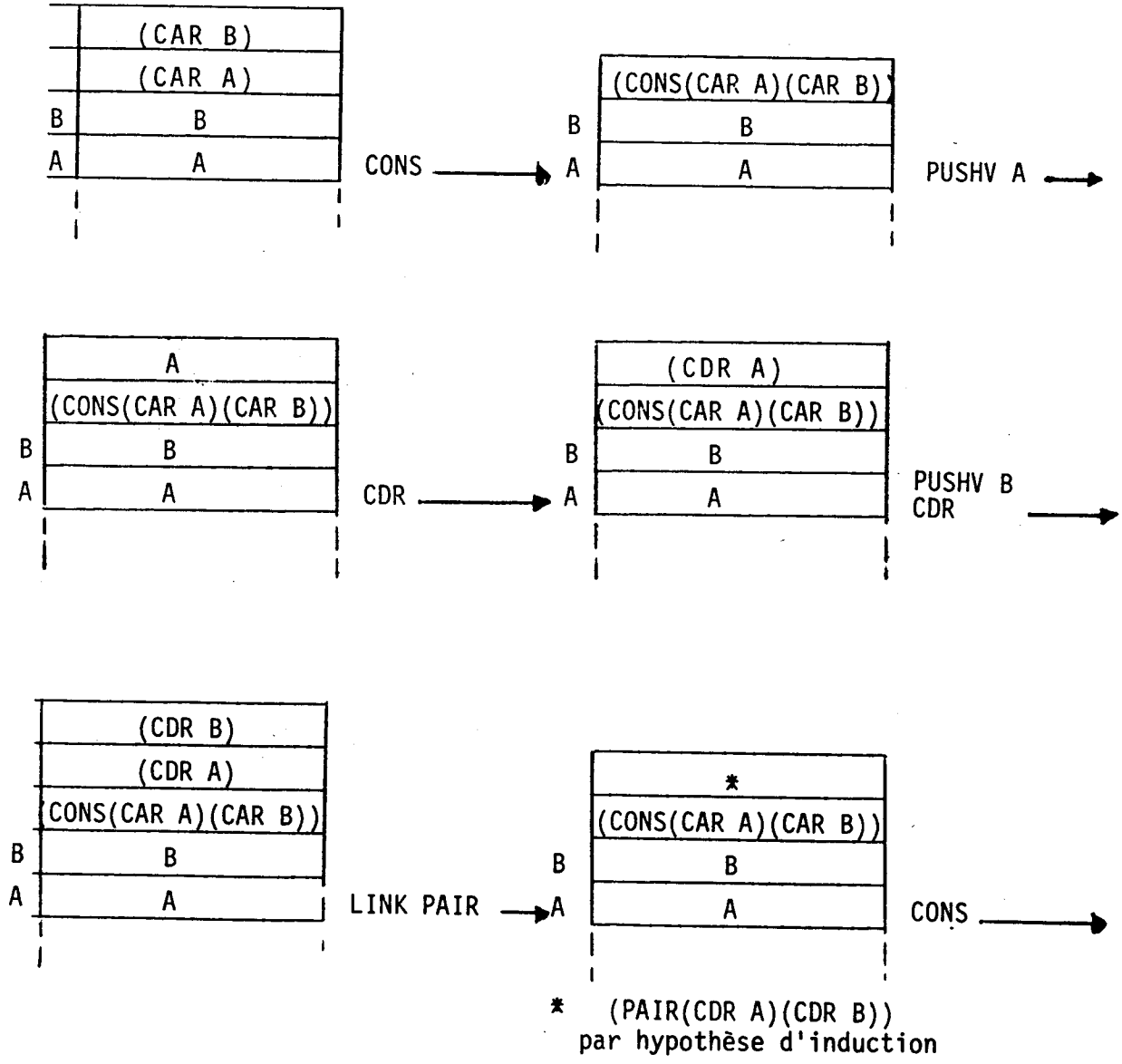


Deuxième pas de l'induction : Sous l'hypothèse que le programme est valide si longueur(A) $\leq$ n, il est également correct pour longueur(A) = n+1.

L'hypothèse intervient à l'appel récursif, en notant que : longueur((CDR a)) < longueur(a).

Le début de l'exécution formelle est identique à la précédente; mais cette fois-ci on ne va pas à SYMB2 et continue:




(CAR B)
(CAR A)
B
B
A
A
CONS
(CONS(CAR A)(CAR B))
B
B
A
A
PUSHV A
A
(CONS(CAR A)(CAR B))
B
B
A
A
CDR
(CDR A)
(CONS(CAR A)(CAR B))
B
B
A
A
PUSHV B
CDR
(CDR B)
(CDR A)
CONS(CAR A)(CAR B))
B
B
A
A
LINK PAIR
*
(CONS(CAR A)(CAR B))
B
B
A
A
CONS
* (PAIR(CDR A)(CDR B))
par hypothèse d'induction
(CONS(...
B
B
A
A
FREE 2
(CONS(CONS(CAR A)
(CAR B))
(PAIR (CDR A)(CDR B)))

Remarques : *1) Pour être plus rigoureux nous devrions dessiner dans les éléments de la pile des S-graphes à la place des S-expressions, mais cela serait trop encombrant.*

2) Ceci est le seul chapitre dans lequel nous avons affaire à des isomorphismes de S-graphes, alors qu'en général on s'intéresse plutôt à des homomorphismes (fonction EQUAL).

3) Dans ce paragraphe, l'exécution formelle apparaît comme une technique de traduction d'un programme assembleur en une fonction LISP. Les possibilités d'une telle traduction sont pourtant limitées : la simplicité ici vient du fait que les programmes considérés traduisent des λ-expressions et sont de ce fait très bien structurés.

2.4. Un compilateur

Nous avons motivé la définition de la machine MIL par la possibilité de lui faire exécuter toute fonction LISP. MIL étant définie, nous pouvons maintenant préciser comment on obtient un programme en langage machine qui réalise une fonction LISP donnée. Cela est fait par un compilateur.

Un compilateur est une fonction qui prend comme argument une définition de fonction sous forme de λ-expression, soit fn, et qui a comme résultat une liste d'instructions pour la machine MIL. L'assemblage de cette liste d'instructions produit un programme $\mathscr{C}$(fn).

Un compilateur est correct si $\mathscr{C}$(fn) réalise fn pour toute fonction fn et si il refuse toute fonction non-conforme aux spécifications du langage.

En essayant d'écrire un compilateur pour LISP pur, nous avons constaté qu'on pouvait le "calquer" sur l'interpré-

teur. Le programme page 21 donne ce compilateur, CAPPLY, qui a la même structure que APPLY dans le sens suivant :

a) Les schémas de programmes [20] des ensembles des quatre fonctions de l'interpréteur page 20 et des quatre fonctions du compilateur sont isomorphes.

b) Dans les deux schémas, les tests effectués sont les mêmes.

c) A l'intérieur des deux schémas il y a bijection entre :
APPLY et CAPPLY
EVAL et CEVAL
EVLIS et CEVLIS
EVCON et CEVCON.

Si un appel de APPLY figure dans l'interpréteur, un appel de CAPPLY figure dans la même position du compilateur.

Cet isomorphisme n'est ni du à une coïncidence ni une simple conséquence du fait que le compilateur et la machine MIL ont été développés ensembles, mais il reflète une relation plus générale entre interpréteurs et compilateurs. Cette relation pourrait se formaliser par l'écriture d'un métacompilateur, dont une application intéressante serait la compilation des appels de FEXPRs.

Ces questions trouveraient une réponse dans une théorie de la compilation et nous ne les avons pas poursuivies dans ce travail. Nous ne cherchons donc pas à approfondir l'étude de la notion de validité du compilateur - la définition donnée ci-dessus est très imparfaite : d'une part elle accepterait un "compilateur" qui génère simplement un appel de l'interpréteur, d'autre part des fonctions contenant des variables libres ne peuvent être traduites par le compilateur de la page 21.

Le problème des variables libres est un cas particulier d'intervention du contexte d'exécution qui complique la compilation. En LISP1.5 ce cas est résolu, mais les possibilités

d'influence de l'environnement à l'exécution sont telles qu'il est impossible de compiler des fonctions sans faire des hypothèses sur la discipline du programmeur. En plus, ces hypothèses ne peuvent être vérifiées - il est par exemple impossible de savoir en LISP1.5 si la définition d'une fonction subit des modifications dynamiques.

Nous concluons de cette réflexion que la compilation, qui est nécessaire pour des raisons d'efficacité, ne rend pas superflue l'interprétation. Dans un système LISP, les deux sont absolument indispensable pour faire du système un outil efficace et puissant.

Remarque : Le compilateur que nous présentons ne comporte pas toutes les parties d'un compilateur classique : il n'y a ni analyse syntactique ni traitement de types, et l'assembleur est considéré comme un programme distinct. En plus on fait l'hypothèse que l'argument est une fonction bien-formée. Cette hypothèse est en général justifiée dans un système LISP où la mise au point se fait à l'aide de l'interpréteur avant la compilation.

3. EXTENSIONS DU FORMALISME - LISP1.5

Le formalisme présenté dans les chapitres précédents s'applique à LISP pur. LISP pur est un modèle théorique d'un langage de programmation qui sert dans un certain nombre de recherches fondamentales.

Maintenant nous allons augmenter la complexité du modèle en introduisant un certain nombre d'extensions pour arriver à la description d'un système réel qui est LISP1.5(1). Ces extensions n'altèrent pas les fondations théoriques du langage : LISP pur reste toujours inclu dans LISP1.5. La syntaxe n'est guère changée. Les extensions sont introduites à l'aide de nouvelles fonctions primitives qui seront également des instructions de la machine MIL.

La différence essentielle entre LISP pur et LISP1.5 réside dans le contexte d'exécution. Par définition, un contexte (ou environnement) est constitué par l'ensemble des données, en dehors de la fonction elle-même, qui fournissent des informations à l'exécution d'une fonction.

En LISP pur le contexte est formé par la ALIST seulement (§ 2.2.) et le contexte initial est donné par les arguments de la fonction à interpréter. Le programmeur ne peut pas explicitement modifier le contexte. En LISP1.5 le contexte initial est donné par une mémoire permanente, en général très complexe, et les arguments. En plus la ALIST construite pendant l'exécution est accessible au programmeur, et peut être manipulée comme toute donnée. Cela permet la construction d'une arborescence de contextes, qui n'est pas équivalente à une pile, comme l'a montre MOSES [34] (2).

---

(1) Il existe un grand nombre de versions plus récentes de LISP, par exemple [18, 27,32,38,43 , 44] différentes de LISP1.5 par des points assez techniques facilement expliqués au niveau de la machine abstraite.

(2) C'est une finesse théorique : la plupart des implantations de LISP utilisent une pile à la place de la ALIST. L'idée d'une arborescence de contextes se retrouve dans les langages de "résolution de problèmes"[4] .

C'est ce contexte infiniment plus complexe qu'en LISP pur qui rend pratiquement impossible l'analyse formelle de fonctions en LISP1.5.

La première extension concerne les entrées/sorties que tout système réel doit effectuer. Ceci sera aussi l'occasion de définir formellement le lien entre les S-graphes sur lesquels nous basons notre formalisme et les S-expressions qui sont habituellement utilisées dans la présentation de LISP.

La deuxième extension marque la différence entre LISP pur et les langages de programmations qui en dérivent : l'introduction d'une mémoire permanente grâce à laquelle le système peut s'enrichir et accumuler de grandes quantités d'informations.

La définition des blocs donne de nouvelles possibilités de programmer des algorithmes, introduisant en LISP des constructions classiques dans d'autres langages de programmations.

Nous avons omis la description d'extensions à caractère plus technique, notamment le traitement d'erreurs.

A première vue, une autre extension, l'introduction de nombres et des fonctions d'arithmétique, revêt un caractère purement technique parce qu'elle est imposée par des impératifs pratiques. Si ce n'était pour des raisons d'efficacité, on pourrait représenter un entier n par une liste de longueur n. Néanmoins, les difficultés d'une implémentation très performante de l'arithmétique sont dûes à un problème de fond : l'introduction dans le langage de données d'un type différent de "S-graphe".

En LISP1.5 les nombres entiers et en virgule flottante sont considérés comme des atomes particuliers (1) qui peuvent servir d'arguments aux fonctions arithmétiques et logiques.

---

(1) L'égalité numérique est testée par EQUAL, non pas par EQ.

Les prédicats NUMBERP(X) et FLOATP(X) permettent de tester si un S-graphe est un nombre ou un nombre en virgule flottante resp. Ces prédicats ainsi que l'ensemble des fonctions arithmétiques et logiques (fonctions booléennes) sont des nouvelles fonctions primitives.

La représentation des nombres par un codage dans des S-graphes, choisie en LISP/360 (voir II.2.1.1), implique deux appels de CONS à la fin de chaque opération arithmétique pour construire l'atome-résultat. De ce fait, le coût d'une seule opération est d'au moins 50 instructions. Cette inefficacité des calculs numériques a limité les applications de LISP; cependant, il existe une solution meilleure réalisée dans MACLISP(1), où on peut effectuer des calculs numériques à la même vitesse qu'en FORTRAN, dans certains cas même plus vite (sur PDP)(2).

---

(1) Le dialecte de LISP utilisé dans le projet MAC du M.I.T.

(2) Project MAC Progress Report X.

3.1. Entrées-sorties

Les S-graphes constituent le format interne des informations traitées par la machine MIL. Dans les entrées/sorties - la communication avec l'extérieur, en particulier avec l'homme - elle utilise un langage défini par une grammaire hors-contexte. Ce langage des S-expressions a deux avantages par rapport à l'utilisation directe des S-graphes :

- un langage "linéaire" se prête mieux à la transmission sur les supports habituels,
- l'écriture sous forme de listes est plus compacte, alors que les S-graphes occupent vite des pages et des pages.

D'autre part, cette écriture est limitée aux S-graphes sans circuits - on limite donc l'ensemble des structures considérées.

3.1.1. Préliminaires

Un exposé rigoureux du point de vue mathématique serait assez fastidieux et demanderait une notation lourde. Mais comme il n'y a pas de difficulté particulière ni de résultat profond à démonter, nous nous contentons d'esquisser la situation et de donner ensuite les définitions et des explications intuitives.

La situation est compliquée parce que nous sommes en présence d'un grand nombre de structures algébriques, homomorphes les unes des autres.

Soient :
la structure des S-graphes :

$$\Sigma = (S, CAR, CDR, CONS, ATOM, EQ) \qquad (S \text{ l'ensemble des S-graphes})$$

et les sous-structures :

$\Sigma_X = (S_X, CAR_X, CDR_X, CONS_X, ATOM_X, EQ_X)$ la sous-structure de Σ définie sur

S_X, l'ensemble des S-graphes sans circuit et

$\Sigma_L = (S_L, CAR_L, CDR_L, CONS_L, ATOM_L, EQ_L)$ la sous-structure de Σ_L

(et de Σ) S_L est défini récursivement par

$$S_L = \{s \mid s \in S_X \wedge (\ ATOM(CDR(X)) \vee NULL(CDR(X)) \vee CDR(X) \in S_L)\}$$

(tous les CDR d'un élément de S_L sont non-atomiques ou l'atome NIL qui représente la liste vide).

Dans ce chapitre nous définirons trois autres structures algébriques :

— les S-expressions pointées $\Pi = (\pi, CAR_\Pi, CDR_\Pi, CONS_\Pi, ATOM_\Pi, EQ_\Pi)$

- les listes $\Lambda = (\lambda, CAR_\lambda, CDR_\lambda, CONS_\lambda, ATOM_\lambda, EQ_\lambda)$

- les S-expressions $\Xi = (\xi, CAR_\xi, CDR_\xi, CONS_\xi, ATOM_\xi, EQ_\xi)$

Les ensembles Π, λ et ξ sont définis par des grammaires hors-contexte avec le vocabulaire :

$$V = \{(\ ) \ .\} \cup \{atomes\}$$

Nous donnons ensuite des homomorphismes(1) :

$$PRINP : S_X \to \Pi$$
$$PRINL : S_L \to \lambda$$
$$PRINT : S_X \to \xi$$

Ces homomorphismes définissent une relation d'équivalence dans S_X et dans S_L, qui est donnée par la fonction EQUAL. Chaque classe contient un et un seul élément arborescent. Les applications :

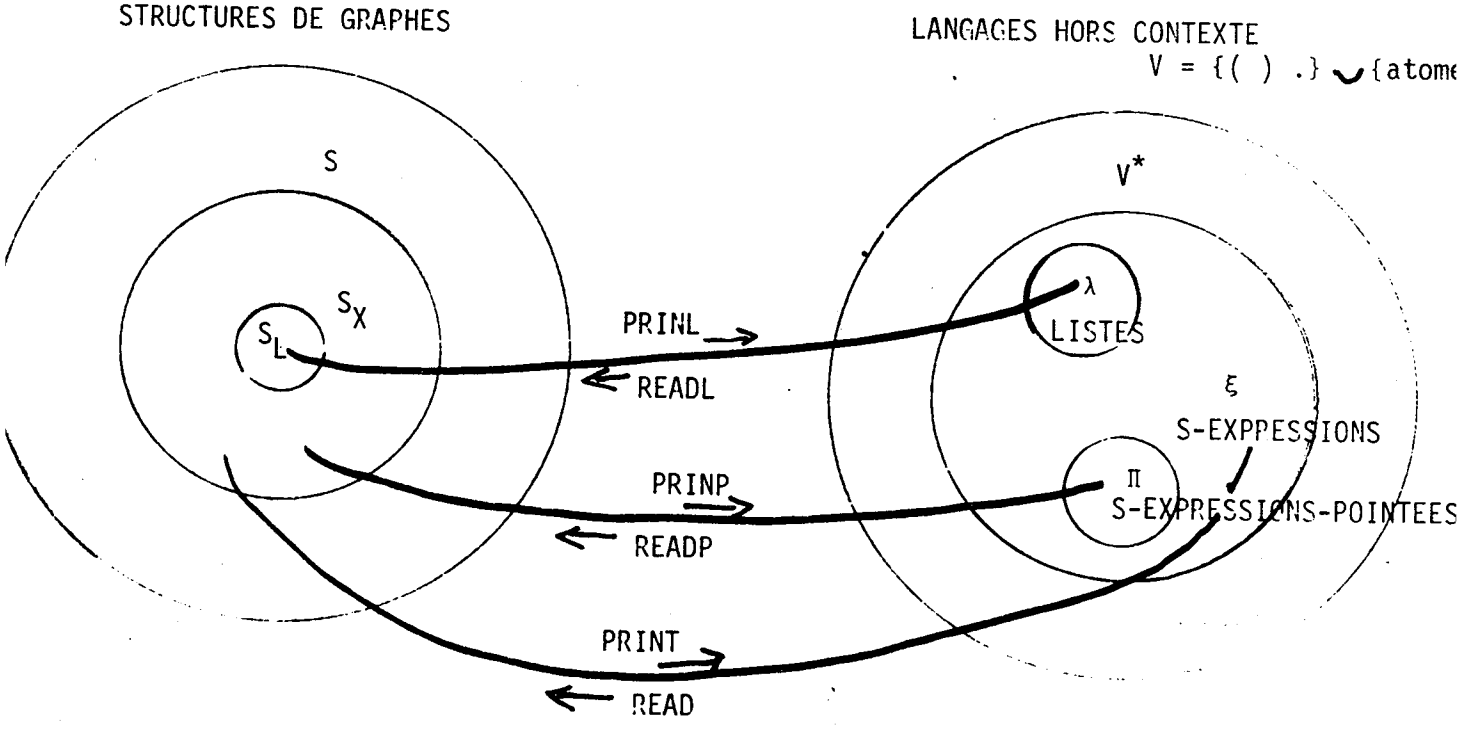
$$READP : \Pi \to S_X$$
$$READL : \lambda \to S_L$$
$$READ : \xi \to S_X$$

---

(1) Pour qu'on puisse effectivement parler d'homomorphisme il aurait fallu donner à EQ (p. 12) la définition donnée au 3.1.2., mais cela serait gênant dans la pratique.

fournissent le représentant arborescent associé à leur argument. (On peut également les considérer comme des applications dans l'ensemble quotient π/PRINP etc.).

La situation est décrite dans la figure ci-dessous. Les fonctions PRINT et READ sont les fonctions standard d'entrée-sortie utilisées par la machine MIL.



3.1.2. Définitions

3.1.2.1. S-expressions pointées

Soit le vocabulaire V = {( ) . } ∪ {atomes}
Les S-expressions pointées sont données par la grammaire :

<SXP> ::= <atome> | (<CAR> . <CDR>)
<CAR> ::= <SXP>
<CDR> ::= <SXP>

Cette grammaire définit également les applications CAR et CDR au niveau de ces expressions(1).

CONS est aussi facile à définir :

CONS [x1, x2] = (x1.x2)

La définition de ATOM est évidente, pour EQ voir le paragraphe suivant.

3.1.2.2. Listes

Les listes sont données par la grammaire :

<L> ::= <atome> | () | (<INT>)
<INT> ::= <L> | <L> <separateur> <INT>

Le séparateur habituel est le blanc.
L'atome NIL est synonyme avec la liste vide () (ce n'est plus le cas en RLISP, § 2.1.3.).

Fonctions de base pour les listes

Le CAR d'une liste non-atomique est son premier élément :

CAR [$(e_1 \ldots e_n)$] = e_1

Le CDR est la liste privée de son premier élément.

CDR [$(e_1 \ldots e_n)$] = $(e_2 \ldots e_n)$

---

(1) CAR_π, CDR_π dans la notation du préliminaire

La symétrie qui existe dans les S-graphes et dans les S-expressions pointées entre les opérations CAR et CDR disparaît donc au niveau des listes.

CONS introduit un nouvel élément au début de son deuxième argument qui doit être une liste non-atomique :

$$\text{CONS } [e, (e_1 \ldots e_n)] = (e\ e_1 \ldots e_n)$$

EQ et EQUAL

La fonction EQ est la seule primitive dont la définition au niveau des listes (et S-expressions) pose des problèmes. Dans Σ elle désigne l'identité entre S-graphes, l'égalité d'adresses au niveau de la machine, mais dans les entrées-sorties cette identité est conservée pour les atomes seulement (qui sont caractérisés par un emplacement unique en mémoire). C'est pourquoi EQ est une fonction partielle pour les listes et les S-expressions, définie seulement si un des deux arguments est un atome(1) :

$$EQ(x,y) \begin{cases} = \text{T si x et y sont le même atome} \\ = \text{NIL si x et y ne sont pas le même atome ou si un argument seulement est un atome} \\ = \text{indéfini dans les autres cas.} \end{cases}$$

L'égalité de deux listes non-atomiques en tant que chaînes de caractères correspond à une égalité de <u>structure</u> entre S-graphe qui est testée par la fonction EQUAL comme suit :

```
EQUAL(X,Y) := IF ATOM X THEN EQ(X,Y)
              ELSE IF ATOM Y THEN NIL
              ELSE IF EQUAL(CAR X, CAR Y) THEN EQUAL(CDR X, CDR Y)
              ELSE NIL;
```

La fonction EQUAL est un exemple simple d'un parcours récursif d'arborescences (EQUAL ne termine pas pour des S-graphes n'appartenant pas à S_X).

---

(1) Pour la même raison les fonctions RPLACA et RPLACD ne peuvent être définies pour des S-expressions.

3.1.3. Primitives de sortie

Pour la communication avec l'utilisateur, la machine M I L possède une voie d'entrée et une voie de sortie que nous imaginons connectées à une console (1). L'écriture de S-expressions par l'intermédiaire d'une zone de sortie se fait à l'aide de deux nouvelles fonctions primitives (qui sont des nouvelles instructions pour la machine).

PRIN1 a — Place (les caractères de) l'atome a dans la zone de sortie. Si la zone est pleine, elle est d'abord imprimée et remise à blanc.
PRIN1 est indéfini pour des arguments non-atomiques.

TERPRI — Fonction sans argument, TERPRI imprime la zone de sortie sur la console (plus généralement dans la voie de sortie).
Ensuite la zone est remise à blanc.

La conversion d'un S-graphe sans circuit en une S-expression pointée à l'aide de ces primitives est immédiate :

```
PRINP(X) := BEGIN PRINPO X; TERPRI(); RETURN X END ;  (2)
PRINPO(X) := IF ATOM X THEN PRIN1 X
             ELSE BEGIN
                    PRIN1 LPAR;
                    PRINPO CAR X;
                    PRIN1 DOT;
                    PRINPO CDR X;
                    PRIN1 RPAR
                  END;
```

LPAR, DOT, RPAR désignent les caractères (. et) resp.
Cette fonction est typique pour les fonctions qui vont suivre. La fonction PRINP même est triviale : elle appelle la fonction d'impression proprement dite, PRINPO, effectue le retour de chariot final et retourne comme résultat son argument. Ainsi, PRINP est une fonction "transparente" qu'on peut placer n'importe où dans une expression sans changer la valeur.

---

(1) L'utilisation d'autres voies de sortie est une question purement technique qui a été bien résolue en LISP1.6 et en MACLISP

(2) Le bloc "BEGIN END" est défini au paragraphe 3.3.3.

La fonction PRINPO effectue un parcours récursif de son argument.

On voit que l'écriture des S-expressions pointées est encore très proche des S-graphes : à chaque noeud non-atomique correspond un "cadre"

(.)

Cela rend très lourde l'écriture d'expressions compliquées; c'est pourquoi les S-expressions pointées ne sont pratiquement jamais utilisées. Elles ont un intérêt purement historique puisque à l'origine le langage LISP a été défini sur des S-expressions pointées et non pas sur des graphes. (De ce point de vue les S-graphes représentent une implantation possible de LISP pur. A ce jour, toutes les implantation de ce langage l'utilisent, et les fonctions RPLACA, RPLACD exploitent les possibilités spécifiques aux S-graphes). Mais l'écriture sous forme de listes est nettement plus avantageuse. Par exemple, à la liste :

(A (B C) D)

correspond l'expression pointée :

(A . ((B .(C . NIL)) . (D . NIL)))

La fonction suivante imprime la liste correspondant à un S-graphe:

```
PRINL(X) := BEGIN PRINLO X; TERPRI(); RETURN X END;
PRINLO(X) := IF ATOM X THEN PRIN1 X
                ELSE BEGIN
                       PRIN1 LPAR;
                       PRINLO CAR X;
                       WHILE CDR X DO
                           BEGIN PRIN1 BLANK; X := CDR X;
                                 PRINLO CAR X END;
                       PRIN1 RPAR
                     END;
```

3.1.4. S-expressions

Les listes représentent une structure légèrement moins générale que les S-expressions pointées (voir la définition de S_L dans le préliminaire), mais heureusement, il n'y a aucun inconvénient à mélanger les deux écritures. Cela amène à la définition des S-expressions, structure réellement utilisée dans les entrées/sorties de la machine MIL. Les S-expressions sont générées par la grammaire :

<S> ::= <atome> | (<INTER>)
<INTER> ::= <vide> | <S><QUEUE>
QUEUE ::= <vide> |. <S>|<séparateur> <S> <QUEUE>

On constate que les S-expressions pointées et les listes sont des sous-ensembles des S-expressions.

Les définitions des fonctions élémentaires données pour les expressions et les listes se généralisent facilement aux S-expressions.

Un défaut de cette écriture est qu'une S-expression a en général plusieurs écritures, c'est-à-dire :

EQUAL(X,Y)

peut être vrai même si X et Y ne sont pas des chaînes de caractères identiques. Par exemple :

(A (B . C) D) ≈ (A . ((B . C) D)) ≈ (A (B . C) . (D)) ≈
(A (B . C) D . NIL) etc...

Mais cela n'a aucune importance pratique. La fonction PRINT imprime toujours la S-expression unique qui utilise le moins de points possibles (la première dans l'exemple ci-dessus).

Voici la fonction PRINT :

```
PRINT(X) := BEGIN PRINTO X; TERPRI(); RETURN X END;
PRINTO(X) := IF ATOM X THEN PRIN1 X
     ELSE BEGIN
        PRIN1 LPAR; PRINTO CAR X; X := CDR X;
        WHILE NOT ATOM X DO
            BEGIN PRIN1 BLANK; PRINTO CAR X; X := CDR X END ;
        IF NULL X THEN RETURN PRIN1 RPAR;
        PRIN1 BLANK; PRIN1 DOT; PRIN1 BLANK;
        PRIN1 X; PRIN1 RPAR
     END;
```

3.1.5. Primitives d'entrée et l'organisation des atomes

La fonction READ construit un S-graphe arborescent à partir d'une S-expression; elle est donc l'inverse de PRINT.

Le seul point délicat de la lecture provient de la règle d'unicité des atomes (voir définition des S-graphes). Chaque atome est caractérisé par un emplacement unique dans l'espace listes à partir duquel on peut accéder au nom et à la liste de propriété de l'atome.

A la lecture d'un atome il faut donc d'abord chercher s'il existe déjà dans le système et le cas échéant le créer. Afin de faciliter cette recherche tous les atomes connus au système à un moment donné sont chaînés dans la liste d'objets, OBLIST, qui est également utilisée par le ramasse-miettes.

La fonction qui effectue la lecture d'atomes est une nouvelle primitive :

RDOBJ (sans argument) a comme valeur l' "objet" suivant dans le flot d'entrée. Cet objet est un atome et peut être de trois types :

- opérateur : les parenthèses et le point
- identificateur
- nombre.

A l'aide de cette primitive la fonction READ s'écrit comme suit :

```
READ() := (LAMBDA(X); IF X NEQ LPAR THEN X ELSE READ().READ1())
              RDOBJ();
```

Elle utilise la fonction READ1 qui construit le CDR d'une liste :

```
READ1() := (LAMBDA (OBJET);
     IF OBJET EQ DOT THEN (LAMBDA (X,Y);X) (READ(),RDOBJ())
        ELSE IF OBJET EQ RPAR THEN NIL
        ELSE IF OBJET EQ LPAR THEN (READ().READ1()).READ1()
        ELSE OBJET.READ1())
     RDOBJ();
```

Remarque : *Cette fonction n'est pas réaliste dans la mesure où elle ne tient pas compte de la possibilité d'erreur. Une bonne fonction de lecture doit accepter tout mot de V^* et signaler une erreur si ce mot n'est pas une S-expression.*

3.2 - Fonctionnement global

La boucle principale exécutée par un système LISP1.5 est donnée par l'expression suivante :

```
WHILE T DO PRINT APPLY(READ(), READ(), NIL) ;
```

ou en langage assembleur :

```
A     LINK    READ      lire fonction
      LINK    READ      lire liste d'arguments
      PUSHQ   NIL
      LINK    APPLY     interpréter
      LINK    PRINT     imprimer la valeur
      PULL              et la dépiler (PULL est défini au §.3.5)
      GO      A
```

Les systèmes LISP plus récents utilisent en général la fonction EVAL à la place de APPLY :

```
WHILE T DO PRINT EVAL(READ(), NIL) ;
```

3.3 - Structure interne des atomes - Listes de propriétés

Chaque atome possède une mémoire associative appelée "liste de propriétés" (parce que, en général, elle est implémentée par une liste), dont le comportement est défini par les trois primitives d'accès suivantes :

PUTPROP(a,v,i) a et i sont des atomes. Création d'une propriété : PUTPROP met le S-graphe v avec l'indicateur i dans la mémoire associative de l'atome a. On dira que v est la propriété i de a. (v est également la valeur de PUTPROP).

GET(a,i) donne comme valeur la propriété d'indicateur i dans la mémoire associative de l'atome a. Si l'indicateur n'y existe pas, la valeur de GET est NIL . (il est donc impossible de distinguer l'absence d'un indicateur de la présence avec une valeur NIL).

REMPROP(a,i) supprime la propriété d'indicateur i de la mémoire associative de l'atome a.

Les mémoires associatives des atomes ont un caractère permanent : leur durée de vie est celle du système. Ceci change complètement l'utilisation pratique d'un système LISP : alors qu'en LISP pur il faut à chaque appel de la fonction universelle définir toutes les informations qu'on veut utiliser (arrivé à l'étiquette A de la boucle principale (p. 54) l'espace listes est vide),on peut maintenant conserver toute sorte d'information dans les listes de propriétés. Elles constituent une banque de données qu'on manipule lors d'une "séance" de LISP. En général, un appel de l'interpréteur met des résultats partiels dans les listes de propriétés, qui serviront à l'appel suivant.

3.3.1 - Utilisation des listes de propriétés par l'interpréteur

L'interpréteur LISP1.5 utilise deux types d'information globale tout en gardant la liste d'association de LISP pur, dans laquelle se trouvent les données dynamiques d'une exécution : les variables peuvent être liées par une propriété d'indicateur APVAL, les fonctions par une propriété EXPR.

En règle générale, les informations globales (dans les listes de propriétés) prennent la priorité sur les informations temporaires (dans la ALIST), c'est-à-dire l'interpréteur fait d'abord une recherche dans les listes de propriétés et ensuite (si la propriété recherchée est absente) dans la ALIST. Par exemple, l'évaluation d'une variable X se fait de la façon suivante (V est une variable intermédiaire) :

```
IF V:=GET(X,'APVAL) THEN CAR V ELSE CDR ASSOC(X,ALIST)
```

Le fait de distinguer les liaisons de variables des liaisons de fonctions (par des propriétés différentes) implique la possibilité qu'un identificateur ait des valeurs différentes suivant qu'il est considéré comme nom de variable ou nom de fonction, ce qui n'est pas possible en LISP pur. Comme d'autre part, une variable peut avoir comme valeur une fonction il y a des cas ambigus, par exemple :

```
LAMBDA (FOO,Y) ; FOO Y ;
```

où on saura seulement à l'exécution si la deuxième occurence de FOO :

* est une fonction définie par une propriété EXPR
* ou une variable fonctionnelle définie sur la ALIST, ou par une propriété APVAL

Cela complique évidemment la compilation (le compilateur LISP/360 traite ce cas incorrectement).

3.3.2 - Définition de fonction

Il suit des paragraphes précédents qu'en LISP une définition de fonction se fait par un appel de la fonction PUTPROP, par exemple

```
PUTPROP(NOT (LAMBDA (X) (EQ X NIL)) EXPR)
```

Il est plus commode d'utiliser la fonction DEF :

```
DEF(NOM,FN) := PROG2(PUTPROP(NOM,FN,'EXPR),NOM) ;
```

et d'écrire

```
DEF(NOT (LAMBDA (X) (EQ X NIL)))
```

En RLISP une définition de fonction peut se faire de deux façons équivalentes :
En écrivant :

```
nom (arg1, ..., argn) := définition ;
```

ou

```
SYMBOLIC PROCEDURE nom (arg1 ,..., argn) ; définition ;
```

Le choix entre les deux formes est purement une question de goût.

3.3.3 - La propriété SUBR

Une fonction peut non seulement être définie par une λ-expression, mais également par une suite d'instructions dans MI. Elle possède alors un indicateur SUBR dont la valeur permet de trouver le point d'entrée de la fonction (et le nombre de paramètres).

Remarque : Ceci implique la nécessité de représenter dans M des pointeurs vers MI ; cette question est reprise au II.2.1.1.

3.3.4 - L'instruction LINK

Comme nous venons de voir il faut tenir compte de plusieurs possibilité si on cherche la définition d'une fonction ; APPLY les teste dans l'ordre suivant :

1) propriété EXPR (λ-expression)
2) propriété FEXPR (voir 3.4)
3) propriété SUBR (langage machine)
4) propriété FSUBR (voir 3.4)
5) ALIST (λ-expression)

L'instruction LINK doit procéder de la même façon ; cela permet de programmer en langage machine des appels à des fonctions dont on ne connait pas encore la définition.

Cette recherche d'une définition de fonction est évidemment coûteuse ; c'est pourquoi en LISP/360 (et dans d'autres implantations) l'appel d'une fonction en langage machine via LINK est modifié lors de la première exécution et remplacé par un branchement plus direct ("fast link"). Ceci peut diviser le temps d'exécution par un facteur de 4.

Remarque : Si la fonction appelée par LINK est définie par une λ-expression, LINK appelle l'interpréteur (APPLY).

3.3.5 - Traitement des variables libres

Nous avons vu au 2.3.1 que les fonctions de la machine abstraite ne peuvent pas utiliser de variables libres : les variables d'une fonction sont rangées dans un segment de PE propre à cette fonction et inaccessible aux autres. En LISP1.5 cette restriction est levée par l'introduction des variables de type SPECIAL. Une telle variable n'est pas liée dans la pile, mais sous l'indicateur SPECIAL de la liste de propriétés de son nom. La propriété SPECIAL contient une pile de valeurs, dont l'adresse du sommet (qui contient la valeur actuelle de la variable) est connue à la compilation de toute fonction.

3.4 - Passage de paramètres

En LISP pur le passage de paramètres se fait par valeur. Cela ne ressort pas de la définition de LISP en LISP (annexe I) et est explicitement indiquée dans la définition de l'instruction PUSHV. Ce type de passage de paramètres n'est donc pas dicté par la théorie, mais vient de considérations extérieures au langage : le passage par valeur est plus facile à manier dans l'analyse de programmes -par exemple pour la vérification ou des transformations- et il est très simple et efficace à implémenter.

D'autres types de passage de paramètres sont possibles sans changer profondément le formalisme. Par exemple, il est facile de modifier l'interpréteur en sorte que le langage interprété soit un "LISP avec passage de paramètre normal" (un argument est évalué quand sa valeur est effectivement nécessaire, Vuillemin [45]), alors que l'interpréteur lui-même utilise toujours un passage par valeur. Il suffit de compliquer légèrement la structure de la ALIST en associant à chaque nom de variable

a) un indicateur "évalué" ou "non-évalué"
b) la valeur

et modifier PAIRLIS et ASSOC pour en tenir compte. Cependant la compilation efficace de ce type de passage de paramètres nécessiterait une nouvelle instruction au niveau de la machine MIL.

En LISP1.5 il existe un deuxième type de passage de paramètres qui est le plus général possible : les arguments d'une fonction définie par une propriété FEXPR (FSUBR si elle est compilée) sont passés sans évaluation et le nombre d'arguments est arbitraire. EVAL traite un appel de FEXPR en donnant à la fonction deux arguments effectifs :

* la liste non-évaluée des arguments fournis par le programmeur
* le contexte d'évaluation courant (ALIST).

On s'aperçoit que, très généralement, une FEXPR définit une extension de l'interpréteur : l'interpréteur ne pénètre pas dans les arguments, c'est la FEXPR elle-même qui définit leur traitement et qui peut, par

exemple, effectuer des évaluations multiples.

Ce type de passage de paramètres est sensé seulement dans un langage contenant un interpréteur qui permet à une FEXPR d'évaluer facilement un argument.

Exemple : La fonction WHILE, § 3.5

3.5 - Le PROG (bloc)

En LISP pur une fonction est exprimée à l'aide d'appels récursifs de fonction et d'expressions conditionnelles. Ces moyens sont universels en ce sens qu'ils permettent d'exprimer toute fonction calculable, mais ils sont inadéquats pour exprimer naturellement la structure de tout algorithme. L'introduction des blocs met à la disposition du programmeur un arsenal plus riche (et redondant) de primitives, comportant

* la possibilité de regrouper plusieurs formes à des endroits où syntaxiquement il n'en faut qu'une (analogue aux instructions composées) Ce cas arrive fréquemment, par exemple dans les fonctions d'impressions du § 3.3.1 et dans le compilateur.

* La primitive GO permettant notamment la construction de primitives d'itération.
* La définition de variables locales
* l'affectation de variables.

La syntaxe du bloc est donnée par la grammaire donnée au 2.1. La sémantique est la suivante :

1) A L'entrée d'un bloc les variables locales, initialisées à la valeur NIL, sont mises au début de la ALIST.

2) Les instructions sont exécutées en ignorant les valeurs. Une instruction est une forme non-atomique (les atomes étant interprétés comme étiquettes).

3) L'ordre d'évaluation implicite est séquentiel, mais il peut être défini explicitement par la fonction GO.

4) L'exécution du bloc se termine
   - soit à la rencontre d'un appel de RETURN. La valeur du bloc est celle de l'argument du RETURN.
   - soit à l'arrivée à la fin du bloc. Dans ce cas la valeur du bloc est NIL.

5) L'utilisation du GO et du RETURN est soumise à certaines restrictions. Par exemple, il ne faut pas les imbriquer dans des appels de fonction. La nature exacte des restrictions dépend de l'implantation en général elles correspondent à des bonnes normes de programmation et assurent une compilation efficace.

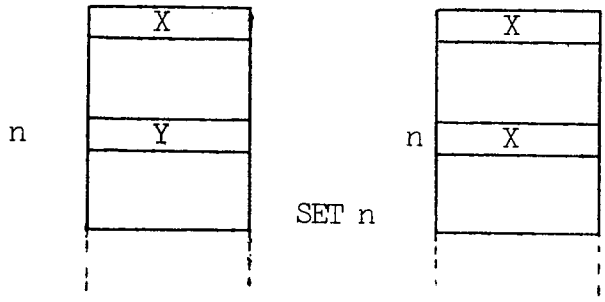
Implantation PROG

En LISP1.5 les blocs sont implantés à l'aide de la FSUBR PROG, donc à l'aide du mécanisme d'extension expliqué au paragraphe précédent. La programmation de cette FSUBR ainsi que la compilation de blocs nécessitent la définition de deux nouvelles instructions de la machine MIL.

PULL enlève le sommet de PE

PULL est utilisé pour enlever de PE les résultats des instructions d'un PROG, qu'il faut ignorer.

SET n recopie le sommet de PE dans l'élément de PE portant le nom n. n est de la même nature que l'argument de l'instruction PUSHV. Le sommet de PE n'est pas modifié.



L'instruction SET ...

Exemple : définition d'un WHILE

La structure d'un bloc étant très générale, elle permet la définition des primitives d'itération classiques. Voici la définition d'un WHILE avec la syntaxe suivante :

<WH> ::= (WHILE <FORME><FORME>)

Au niveau RLISP une syntaxe qui semble naturelle (et qui se définit par une extension simple) serait :

<WH> ::= WHILE <FORME> DO <FORME>

La sémantique est donnée par la FEXPR suivante :

```
FEXPR WHILE(L,A) :=
    BEGIN SCALAR COND, ACTION, RESULT ;
        COND := CAR L ;
        ACTION := CADR L ;
    A : IF EVAL(COND,A) THEN RESULT := EVAL(ACTION,A) ELSE RETURN RESULT
        GO A
    END ;
```

3.6. Récursion et itération

L'introduction des blocs est non seulement motivée par des critères esthétiques ou par le souci de donner au programmeur un maximum de "confort" - mais aussi par le fait que la programmation en LISP pur mène souvent à des exécutions plus complexes que la programmation à l'aide de blocs. Nous étudions ce phénomène sur l'exemple d'un parcours d'arborescence.

La fonction COPY(X) construit une arborescence homomorphe à son argument par un parcours postfixé. Les pages 64,65 montrent deux façons d'écrire cette fonction, qui sont équivalentes dans le sens qu'elles donnent le même résultat.

Soit l(x) la longueur du chemin le plus long entre la racine et une feuille de x, et p(x) le nombre maximal de CAR dans un chemin entre la racine et une feuille, c'est-à-dire :
L(X) := IF ATOM X THEN 0 ELSE 1 + MAX(L(CAR X),L(CDR X)) ;
P(X) := IF ATOM X THEN 0 ELSE MAX(1 + P(CAR X),P(CDR X));

Intuitivement l(x) représente la hauteur d'une arborescence et p(x) la profondeur d'imbrication. On a effectivement :

$$0 \leq p(x) \leq l(x) \qquad \forall\, x$$

Par récurrence on montre facilement que la taille maximale des piles PE et STK lors d'une exécution de COPY(X)
a) est proportionnelle à l(x) pour le programme récursif,
b) est proportionnelle à p(x) pour le programme itératif.

Dans la pratique la profondeur d'imbrication est une mesure de la complexité d'une expression qui s'avère très limitée. On vérifie que :

$p(x) < 100$ dans la quasi-totalité des applications de LISP.

Par contre, on ne peut pas donner de borne supérieure pour l(x).

De ces considérations nous tirons la conclusion suivante : On peut définir une taille raisonnable des piles PE et STK qui sera suffisante pour la fonction COPY itérative dans la majorité des applications. Cela n'est pas possible pour le programme récursif.

Cette conclusion est valable pour d'autres comparaisons entre fonctions récursives et itératives. Etant donné le coût d'une gestion dynamique des piles - si elles n'ont pas une taille fixe comme c'est le cas en LISP/CMS - la programmation itérative est nettement préférable à la programmation récursive en ce qui concerne l'utilisation des piles.

Le bilan est plus difficile à établir pour les temps d'exécution, mais là aussi il est clairement favorable à l'itération, d'autant plus que dans la plupart des implémentations l'appel de fonction (LINK) est relativement coûteux. DARLINGTON et BURSTALL [10] font même état d'un gain de temps d'un facteur de 100 (pour la fonction de Fibonacci). Ces gains sont en général payés par des programmes légèrement plus longs.

Remarque : Nous ne voulons pas affirmer qu'un programme itératif est toujours plus efficace qu'un programme récursif, ni non plus que tout programme récursif peut être transformé en itératif. Cela est vrai pour certaines formes de récursion seulement.

Par contre, il ressort aussi clairement de la comparaison des deux versions de COPY que le programme récursif est plus clair et mieux structuré ; il est certainement plus facile à écrire, à comprendre et à valider. Dans ce cas le programme récursif est donc préférable pour le programmeur, le programme itératif pour la machine.

Procédure récursive

en RLISP:

```
SYMBOLIC PROCEDURE COPY1(X) ;
IF ATOM X THEN X ELSE COPY1(CAR X).COPY1(CDR X) ;
```

en LISP:

```
(COPY1
   (LAMBDA (X)
      (COND
         ((ATOM X) X)
         (T (CONS (COPY1 (CAR X)) (COPY1 (CDR X))) ))))
```

en mach. MIL:

```
COPY1
         (ARGS X)
         (PUSHV X)
         (ATOM)
         (BRNIL SYMB2)
         (PUSHV X)
         (GO SYMB1)
SYMB2    (PUSHV X)
         (CAR)
         (LINK COPY1)
         (PUSHV X)
         (CDR)
         (LINK COPY1)
         (CONS)
SYMB1    (FREE 1)
         (RETURN)
```

Procédure itérative

en RLISP:

```
SYMBOLIC PROCEDURE COPY2(X) ;
IF ATOM X THEN X
  ELSE
    BEGIN SCALAR RESULT,QUEUE ;
       RESULT := QUEUE := COPY2(CAR X) . CDR X ;
       WHILE NOT ATOM CDR X DO
         BEGIN
            X := CDR X ;
            QUEUE := CDR(RPLACD(QUEUE,COPY2(CAR X) . CDR X))
         END ;
       RETURN RESULT
    END ;
```

en LISP:

```
(COPY2
   (LAMBDA (X)
      (COND
         ((ATOM X) X)
         (T (PROG (RESULT QUEUE)
                  (SETQ
                     RESULT
                     (SETQ QUEUE (CONS (COPY2 (CAR X)) (CDR X))) )
                  ( WHILE
                     (NOT (ATOM (CDR X)))
                     (PROG NIL
                           (SETQ X (CDR X))
                           (SETQ
                              QUEUE
                              (CDR (RPLACD QUEUE (CONS(COPY2(CAR
(CDR X))) ))))
                  (RETURN RESULT) )))))
```

en MIL :

```
COPY2   (ARGS X)
        (PUSHV X)
        (ATOM)
        (BRNIL E1)
        (PUSHV X)
        GO E2
E1      (PUSHQ NIL)
        (PUSHQ NIL)
        (ARGS RESULT,QUEUE)
        (PUSHV X)
        (CAR)
        (LINK COPY2)
        (PUSHV X)
        (CDR)
        (CONS)
        (SET QUEUE)
        (SET RESULT)
        (PULL)
E3      (PUSHV X)
        (CDR)
        (ATOM)
        (PUSHQ NIL)
        (EQ)
        (BRNIL E4)
        (PUSHV X)
        (CDR)
        (SET X)
        (PULL)
        (PUSHV QUEUE)
        (PUSHV X)
        (CAR)
        (LINK COPY2)
        (PUSHV X)
        (CDR)
        (CONS)
        (RPLACD)
        (CDR)
        (SET QUEUE)
        (PULL)
        (GO E3)
E4      (PUSHV RESULT)
        (FREE 2)
E2      (FREE 1)
        (RETURN)
```

Remarque : On voit sur cet exemple que les instructions de la machine MIL amènent à des programmes assez lourds, du moins pour des blocs.

On voit ainsi l'intérêt des recherches récentes sur les transformations de programmes [10,39] qui ont pour but le passage automatique entre des différentes formes de programmes, dont le passage récursif-itératif est un cas important. Les résultats obtenus par BURSTALL et DARLINGTON permettent déjà la traduction automatique de la fonction COPY récursive en la version itérative.

A cet égard il est remarquable que le premier compilateur de LISP, écrit vers 1963 par l'équipe de McCarthy (et aussi l'ancêtre du compilateur LISP/360) effectue une telle transformation dans un cas très simple mais fréquent. Voici un exemple illustrant le cas le plus général de cette transformation au niveau de RLISP.

Fonction à transformer :

```
SYMBOLIC PROCEDURE EX(X,Y);
IF X THEN EX(CAR X,CDR Y)
   ELSE IF EX(Y,X) THEN EX(CAR Y,CDR X)
   ELSE H(EX(CAR X,Y));
```

Après transformation deux appels récursifs sont gérés par itération:

```
SYMBOLIC PROCEDURE EX(X,Y);
  BEGIN SCALAR V2,V3;
E5:  IF X THEN GO E6
       ELSE IF EX(Y,X) THEN GO E7
       ELSE RETURN(H(EX(CAR X,Y))) ;
E7:  V2 := CAR Y;
     V3 := CDR X;
     GO E4;
E6:  V2 := CAR X;
     V3 := CDR Y;
     GO E4;
E4:  X := V2;
     Y := V3;
     GO E5
  END;
```

Remarque :

Cette transformation a été réalisée en trois étapes :

1) traduction RLISP -> LISP

2) transformation au niveau LISP

3) traduction LISP -> RLISP avec mise en page.

Seule la deuxième étape est contenue dans le compilateur. La troisième étape représente une application intéressante du traducteur LISP-RLISP mentionné au paragraphe 2.1.

4. UTILISATION DE MIL POUR L'INPLANTATION D'UN SYSTEME LISP

4.1. Principe

Dans les implantations actuelles de LISP, l'interpréteur et un grand nombre (≃ 150-250) de fonctions d'intérêt général sont codés en langage de bas niveau; par exemple, LISP/CMS représente un ensemble de 6000 cartes en langage d'assemblage.

La définition de la machine MIL permet une autre technique d'implantation, procédant en deux étapes :

1) Implantation de MIL
2) Définition du système proprement dit en MIL.

Dans les applications réelles, la première étape doit être programmée dans un langage d'écriture de systèmes ou en assembleur, puisqu'il s'agit notamment de programmer la gestion de mémoire et les primitives d'entrées/sorties. Dans LISP/CMS cette partie représente environ 15% de l'ensemble des programmes.

Par contre, toutes les fonctions définies à la deuxième étape peuvent être écrites en (R)LISP pour ensuite être compilées dans un système LISP existant. Ce principe -de programmer le plus de fonctions possibles dans un langage de plus haut niveau possible - a de nombreux avantages :

*le coût de l'implantation est réduit,
*la fiabilité et aussi la modifiabilité sont augmentées,
*toute la seconde étape est indépendante d'une machine réelle, l'implantation est donc transportable.

En général le prix payé pour ce genre d'avantages est une perte d'efficacité, mais en LISP cela n'est pas forcément le cas : l'élément critique de l'implantation est un bon compilateur; étant donné qu'il est indépendant d'une machine réelle, il est justifié d'investir un certain effort dans la réalisation d'un compilateur optimisé et souple. L'interpréteur ainsi compilé sera moins efficace qu'un interpréteur codé à la main (peut-être de l'ordre de 25 %), mais l'ensemble des fonctions définies par l'utilisateur sera mieux compilé qu'avec les compilateurs courants, et dans les applications importantes de LISP la performance globale du système sera bonne.

Dans la conception d'un système LISP, la priorité doit donc être accordée à l'efficacité des fonctions compilées; bien entendu, cela n'empêche pas de faire quelques retouches manuelles de l'interpréteur (à ce sujet, voir aussi la conclusion).

4.2. Optimisation du modèle

Un deuxième point critique de l'implantation est l'adaptation de la machine MIL à la machine réelle. A cet égard, notre modèle qui a été conçu avec le but de dégager des notions théoriques et de les présenter sous leur forme la plus naturelle, a deux inconvénients majeurs :

*il ne fait aucune utilisation de registres,
*en réduisant toute action à un petit nombre de primitives très simples, les fonctions compilées sont très longues et on ne profite pas de toute la richesse du répertoire d'instructions des machines actuelles (en exemple, voir le code lourd de la fonction COPY2 du § 3.3.4).

Pour remédier à ces inconvénients le compilateur doit générer un code assez riche permettant une expression dense des actions fréquentes en LISP. L'expansion de ce code dépend de la machine réelle et peut donc utiliser les registres. L'utilisation des registres reste ainsi locale; mais en raison du grand nombre d'appels récursifs de fonction en LISP, il serait difficile d'utiliser les registres comme support permanent de variables.

Cette idée est déjà partiellement réalisée dans une nouvelle version du compilateur de LISP/360 par l'équipe de A.C. HEARN à l'Université de Utah.

La sémantique des instructions générées par le compilateur doit être définies par une expansion en instructions de la machine MIL.

Exemples informels et purement hypothétiques

macro-instruction	expansion
CADR	CDR CAR
branchement si faux	
BF eq,x,y,eti	PUSHV X PUSHV Y EQ BRNIL ETI
NEQ x,y	PUSHV X PUSHV Y EQ PUSHQ NIL EQ

Bien entendu, la définition exacte d'un tel code doit se faire en liaison avec l'écriture du compilateur.

DEUXIEME PARTIE

"LISP/CMS"

1. LISP/360 ET LISP/CMS

Le système LISP/360 [24] est directement dérivé des premiers systèmes LISP développés par l'équipe de McCARTHY pour l'ordinateur IBM 7090. Cela explique, d'une part, pourquoi la structure interne de LISP/360 n'est nullement adaptée au système 360 et comporte des passages très maladroits, et d'autre part, pourquoi elle reflète une conception désuète, puisqu' elle n'a pas évoluée depuis 10 ans. Cela est tout particulièrement le cas pour l'utilisation du compilateur (§ 3.3.).

Il est regrettable qu'en raison de la grande diffusion de LISP/360 ce système soit souvent identifié avec le langage LISP. Les nombreux défauts de ce système sont à l'origine d'un grand nombre de préjugés contre LISP qui ne sont pas justifiés dans le cas général.

Le défaut le plus grave vient d'une mauvaise utilisation des techniques d'adressage du 360 : comme avec un registre de base on peut adresser un segment de 4K octets seulement. Les réalisateurs de LISP/360 ont d'abord utilisé quatre registres de base différents pour l'ensemble de l'interpréteur. Au fur et à mesure des modifications du système (dont on reconnait encore, facilement les traces), ces segments se sont complètement remplis, et l'on a à utiliser d'autres registres de base, avec des conventions de sauvegarde très curieuses.

Avec les années, LISP/360 avait finalement acquis une structure "baroque" qui rendait difficile toute nouvelle transformation.

* Le programme n'étant pas modulaire, toute modification nécessiterait l'assemblage coûteux de l'ensemble des 6000 cartes.
* En général tout rajout faisait déborder un segment adressable par les registres de base du système.

Le développement de LISP/CMS à partir de LISP/360 s'est déroulé en plusieurs étapes :

1) Reprise de la structure globale de l'interpréteur. L'ensemble de LISP/360 a été découpé en une vingtaine de modules indépendants, chacun avec son propre registre de base, et qui communiquent par des sections communes (DSECT). Cela s'est fait avec des nouvelles conventions de liaison de fonction et d'adressabilité. Ces conventions, plus simples que les anciennes, ont comme effet secondaire une légère amélioration de la performance du système et une diminution du volume des fonctions compilées (gains d'environ 7%).

2) Après ce travail pratiquement invisible à l'utilisateur il était possible d'incorporer dans LISP les outils souhaitables dans l'environnement conversationnel CMS, en particulier une interface d'entrée/sortie et un traitement d'erreurs conversationnel.

A la fin de cette étape, LISP/CMS fut "mis en service

3) Depuis, l'utilisation de LISP a conduit à l'étude de problèmes plus complexes d'une part, l'implantation des facilités décrites dans les chapitres 2,3 et 4, et d'autre part, les développements plus théoriques donnés dans la première partie de cette thèse.

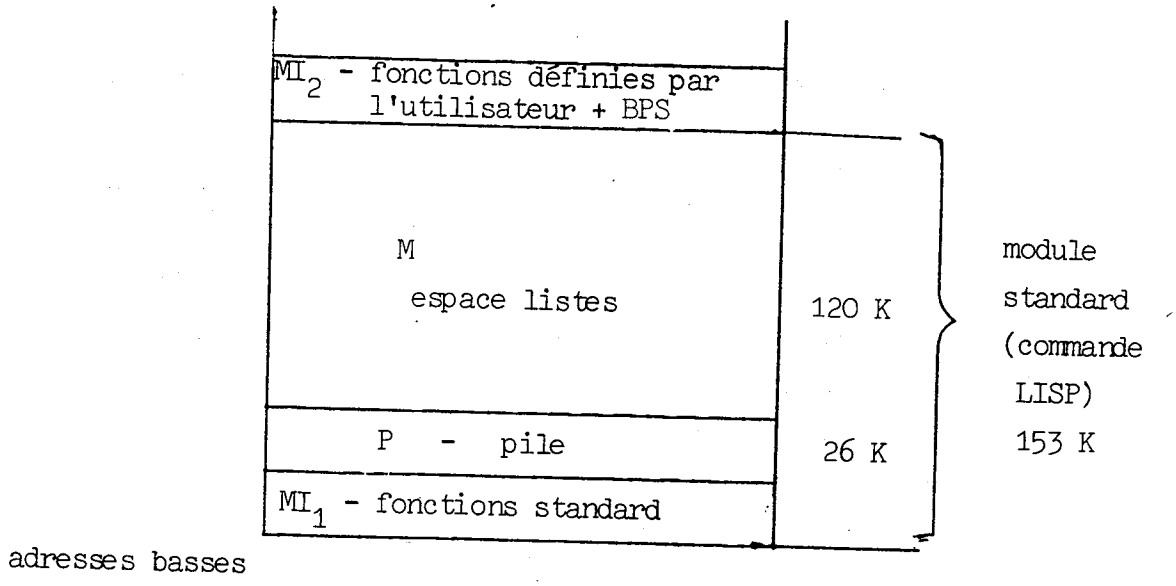
Grâce à la souplesse de LISP/CMS des nouvelles extensions sont possibles et, même si quelques caractéristiques héritées de LISP/360 limitent les possibilités du système, le développement n'est pas terminé : Victor RIVERO a réalisé la liaison avec des programmes FORTRAN et la connexion du terminal graphique 2250 [40], à Berlin LISP/CMS a été adapté à VM370, et nous sommes en contact avec l'équipe de A.C. HEARN à l'Université de Utah, qui étudie également l'amélioration de LISP/360, notamment du compilateur.

2 - Gestion de la mémoire

D'après la structure de la machine abstraite (chapitre 2) on peut distinguer trois espaces dans l'organisation d'un système LISP :

* M ou espace listes, contenant des S-graphes
* MI l'espace des fonctions
* P un espace pour les piles d'exécution.

En LISP/360 ces espaces occupent la mémoire dans l'ordre donné par le schéma suivant :



La figure indique également la configuration du module LISP "standard" tel qu'il existe actuellement sous CMS. Suivant ses besoins, chaque utilisateur peut modifier cette configuration dynamiquement, c'est-à-dire pendant l'exécution : à l'heure actuelle on peut définir des extensions de P, M, et de MI. Cependant, certains défauts de conception de LISP/360 imposent des restrictions. Ceci est représenté plus en détail au 2.2.

2.1 - L'espace listes

C'est l'espace listes qui occupe de loin la plus grande partie du système LISP. Cet espace contient

* les S-graphes sur lesquels travaillent les fonctions en cours (par l'intermédiaire de la pile)

* les atomes avec les listes de propriétés, en particulier les définitions d fonctions interprétées.

* la liste libre dans laquelle CONS puise les nouveaux éléments de listes.

2.1.1 - Types d'information et format en LISP/360

Les éléments de l'espace listes contiennent des informations de plusieurs types. A titre d'illustration nous indiquons le format utilisé en LISP/360.

Un élément est représenté par un double-mot, dont les moitiés sont appelée CAR et CDR. En général un élément contient deux adresses (absolues) occupant deu fois trois octets. Les deux octets restant sont utilisés pour coder le type de l'élément.

a) Le type de loin le plus fréquent est "noeud non-atomique de S-graphe" :

00	pointeur sur le CAR	00	pointeur sur le CDR

b) Les éléments représentant un atome sont caractérisés par le bit 0. Un tel élément donne accès

* au nom d'impression de l'atome
* à la liste de propriétés

80	pointeur nom d'impression	00	pointeur liste de propriétés

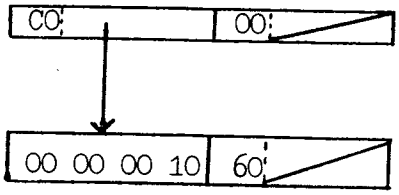
c) Les chaînes de caractères (noms d'impressions des atomes) sont rangées dans des parties CAR d'éléments de listes, en raison de 4 caractères par élément

A B C D		60	pointeur suite (ou NIL)

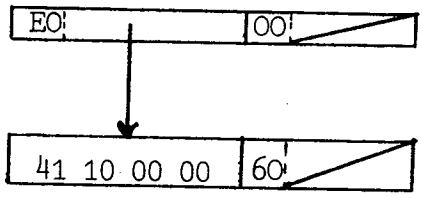
Le premier octet du CDR (valeur 60_{16}) indique que le CAR ne contient pas une adresse.

d) Les atomes numériques ont une structure analogue à celle des atomes ordinaires, mais à la place d'un nom d'impression on trouve la valeur :

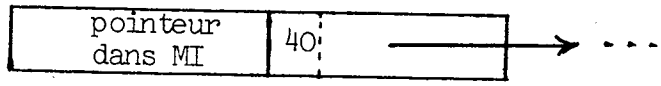
atome nombre entier 10_{16}



atome nombre flottant 1.0



e) Certaines propriétés d'atomes (indicateurs SUBR, FSUBR) contiennent des adresses dans l'espace MI, caractérisées par la valeur 40 du premier octet du CDR :



La distinction entre les types d'éléments est primordiale pour les algorithmes de ramassage de miettes et de tassement (2.1.)

Pour l'interpréteur, seule la distinction entre les noeuds ordinaires et les atomes est importante.

Bien entendu, les formats indiqués sont spécifiques à l'implantation de LISP/360. Ailleurs on trouve d'autres types d'éléments et d'autres façons d'indiquer le type, mais il est clair que le choix de ces formats est fondamental pour la programmation et la performance de l'ensemble du système. Le choix de LISP/360 a des avantages et des inconvénients bien connus :

* les opérations CAR et CDR sont rapides ; elles s'exécutent en une instructions, mais il n'y a pas de test du type de l'argument : on peut prendre le CAR et le CDR d'un atome, qui représentent le nom d'impression et la liste de propriétés resp . Cette ambiguité des fonctions de base complique la mise au point du programme.

* chaque élément de l'espace listes peut indifféremment être utilisé pour un type ou un autre. L'espace listes est parfaitement homogène.

* gaspillage de mémoire : les 64 bits d'un double-mot sont en général mal utilisés. 18 bits par pointeur seraient bien suffisants, mais ceci est incompatible avec la longueur du mot machine. La représentation d'un nombre de 32 bits occupent 128 bits -ceci aussi est difficile à changer.

* dans l'implantation actuelle on peut difficilement introduire de nouveaux types.

2.1.2 - Le ramasse-miettes

Le ramasse-miettes (r.m.) est activé chaque fois que la fonction CONS trouve la liste libre vide. Le r.m. procède alors en deux étapes :

1°) Marquage de tous les éléments actifs. Un élément de l'espace listes est actif s'il peut être référencé par une fonction, c'est-à-dire s'il fait partie d'un S-graphe,

* pointé par la pile d'exécution

* contenu dans la structure interne d'un atome. Pour simplifier le marquage des atomes, ceux-ci sont chaînés dans la liste d'objets, OBLIST. La OBLIST peut être considérée comme un graphe binaire avec circuits contenant tous les atomes et les listes de propriétés.

Ici se pose un problème important qui est très mal résolu en LISP/360 : les fonctions contenues dans MI peuvent contenir des pointeurs sur des S-graphes par l'intermédiaire de l'instruction PUSHQ (I.2.3) et ces S-graphes doivent évidemment être considérés comme actifs. Mais compte tenu de la réalisation en LISP/360 de l'instruction PUSHQ il est pratiquement impossible de localiser les pointeurs vers M contenus dans MI.

Le chargement dans le registre A d'un pointeur sur un S-graphe constant est programmé par

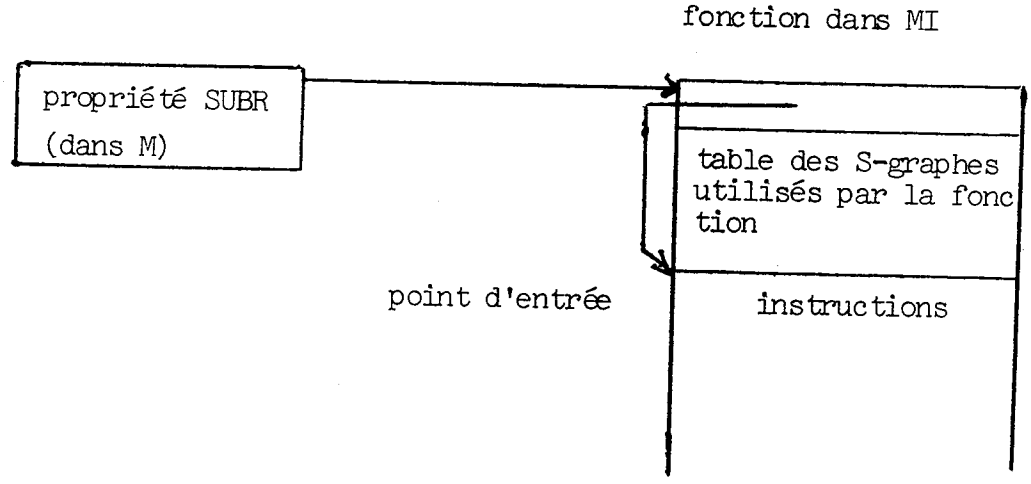
```
        L     A,AS       adresse relative
        AR    A,NILR     adresse absolue
        :
AS      DC    A (adresse du S-graphe modulo NIL)
```

Il n'est pas possible de localiser les pointeurs sur des S-graphes (comme AS dans l'exemple) à l'intérieur des fonctions.
Le même problème se pose pour les doublets SPECIALS.

Le problème est contourné plutôt que résolu : les S-graphes "quotés" par les fonctions dans MI sont chaînés dans une liste qui est la propriété SPQT de l'atome QUOTES. Ils sont donc accrochés à la OBLIST.

Ceci est suffisant pour le r.m., mais limite les possibilités de tassement (§ suivant)

Une bonne solution demanderait une modification importante de l'assembleur LAP. Il faudrait mettre tous les pointeurs sur des S-graphes dans une table dont le descripteur doit être accessible à partir de la propriété SUBR de la fonction.
Par exemple, une SUBR pourrait avoir la structure suivante :



Remarque : Ce problème ne concerne que des fonctions dans MI_2. Les fonctions standards dans MI_1 ont seulement besoin de quelques atomes qui doivent de toute façon faire partie de la OBLIST.

En résumé on peut dire que la propriété suivante est fondamentale pour la réalisation de la première phase du r.m. :

A chaque instant où la fonction CONS peut être appelée tous les éléments actifs de l'espace listes doivent être accessible au r.m.

2°) Construction de la nouvelle liste libre et démarquage.
C'est un parcours linéaire (d'un bout à l'autre) de l'espace listes. Les élémer non marqués (dont non-actifs) sont chaînés dans la nouvelle liste libre, les éléments actifs sont démarqués.

Si la nouvelle liste libre n'est pas vide, la fonction CONS qui avait activé le r.m. peut s'exécuter ; si elle est vide, le système reporte une erreur.
Cet algorithme de r.m. en deux phases est très classique. Par sa nature même, c'est un traitement coûteux : soient

nt le nombre total d'éléments de l'espace listes
na le nombre d'éléments actifs
t_{rm} le temps d'un r.m.

On a alors la relation

$$t_{rm} \sim nt + na \qquad (1)$$

(c'est-à-dire $t_{rm} = c_1 * nt + c_2 * na + c_3$ où les c_i dépendent de la formulation du programme et de la vitesse d'exécution des instructions),

puisque le temps de la première étape est à peu près proportionnel à na, le temps de la deuxième étape à nt.

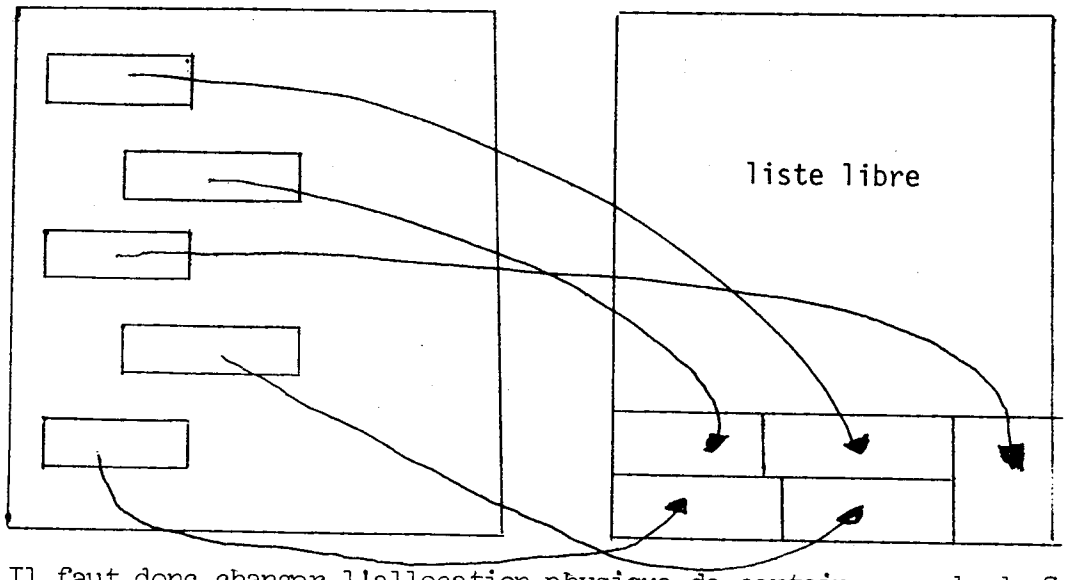
Dans les applications typiques nous avons constaté que 15 % à 30 % du temps total sont utilisés par la vingtaine d'instructions dans les deux boucles principales du r.m. Pour réduire ce temps on peut penser à agrandir l'espace listes.Cependant, vue la relation (1) ceci est sans effet si

$$nt >> na$$

Les appels du r.m. sont moins fréquents mais chaque appel est plus coûteux. (En plus, dans un environnement paginé le nombre de fautes de pages augmente avec la taille de l'espace listes).

2.1.3 - Tassement

En LISP/CMS un tassement de l'espace listes est effectué en vue de la sauvegarde de l'espace dans un fichier externe. Dans sa forme la plus simple, le tassement a l'effet suivant : tous les éléments actifs de l'espace listes deviennent contigus.



Il faut donc changer l'allocation physique de certains noeuds de S-graphes dans l'espace listes ; ensuite il faut mettre à jour tous les pointeurs sur ces noeuds. Pour que ceci soit possible la condition suivante doit être vérifiée :

Tous les pointeurs sur un élément de l'espace listes donné peuvent être trouvés à un prix raisonnable.

Ceci n'est pas le cas en LISP/360 comme nous l'avons vu au § précédent. Ceci impose des restrictions à l'utilisation de la fonction SAVE (3.2.1)

2.1.4 - L'incidence du r.m. sur la performance du système

L'analyse qualitative du coût du r.m. à deux phases, exprimée par la relation (1) est alarmante et pousse à la recherche d'autres techniques de r.m.

La méthode suivante, à l'origine dûe à Minsky |31|, a été réinventée

et discutée avec des variations dans plusieurs publications [6,12,13,17] Il s'agit d'un r.m. avec tassement (compacting garbage collector). Supposons que l'on dispose de deux espaces listes, M^{α} et M^{β} dont un seulement est utilisé, soit M^{α}. Quand la liste libre dans M^{α} est vide on recopie les éléments actifs de M^{α} dans des mémoires successives de M^{β}. Ensuite M^{β} est utilisé comme espace listes, et au r.m. suivant les rôles de M^{α} et M^{β} sont inversés. (La liste libre n'est pas organisée explicitement, mais elle est constituée des éléments au delà de la dernière mémoire utilisée dans la recopie.)

Dans une machine ayant une seule mémoire physique cette méthode a peu d'intérêt. La moitié de la mémoire resterait inutilisée la plupart du temps. Par contre, dans une mémoire paginée la méthode est très séduisante.

* Lors du r.m. on remplit M^{β} séquentiellement. Il suffit donc d'avoir en mémoire rapide une seule page de M^{β} à un moment donné.

* Après le r.m. on peut éliminer M^{α}, devenue inutile, en supprimant les pages correspondantes dans la table des pages.

* en plus du r.m. on a un tassement gratuit.

* Un bon algorithme de recopie permet d'éviter le problème de l'éparpillement de petites listes sur plusieurs pages, évoqué par Cohen [8].

* le coût de l'opération est proportionnel à na

$$t_{rm} \sim na \qquad (2)$$

Par conséquent, un aggrandissement de l'espace listes diminue toujours la part c_{rm} du r.m. dans le coût total du système

$$\Delta c_{rm} \sim \Delta(nt-na)$$

si n_{cons} est le nombre d'appel de CONS, on a la relation

$$c_{rm} \sim n_{cons} * na \qquad (3)$$

Une réflexion simple montre que l'on peut encore faire mieux. Entre deux appels du r.m. l'ensemble des listes actives ne change pas complètement et des grandes parties de M^{α} seront recopiées telles quelles dans M^{β}. Si on organise judicieusement l'espace listes et que l'on associe à chaque page une marque d'écriture, les pages contenant des données statiques n'interviennent plus dans le r.m. On a alors, ns étant le nombre d'éléments actifs appartenant aux données statiques :

$$t_{rm} \sim na - ns \qquad (4)$$

Cela est minimal, car il est clair que le coût de la gestion de données dynamiques est au moins proportionnel à leur nombre. Même s'il est impossible de séparer complètement données statiques et données dynamiques, on peut tirer la conclusion suivante :

On peut définir une implantation de LISP techniquement fiable, dans laquel le le coût de la gestion de l'espace listes est conforme au minimum théorique t_{rm} de la formule (4).
Ceci montre -au moins de façon qualitative- que le préjugé très répandu selon lequel la gestion de mémoire coûte toujours cher en LISP, n'est pas justifié dans le principe. Bien entendu, un tel jugement est valable au sujet d'une implantation concrète comme celle de LISP/360, comme le montre la relation (1).

On peut encore énoncer deux principes importants :

* une nouvelle implantation de LISP doit partir d'une analyse rigoureuse du coût de r.m. (c_{rm})
  (c'est sur ce point que Weizenbaum a sévèrement critiqué LISP2 [48])

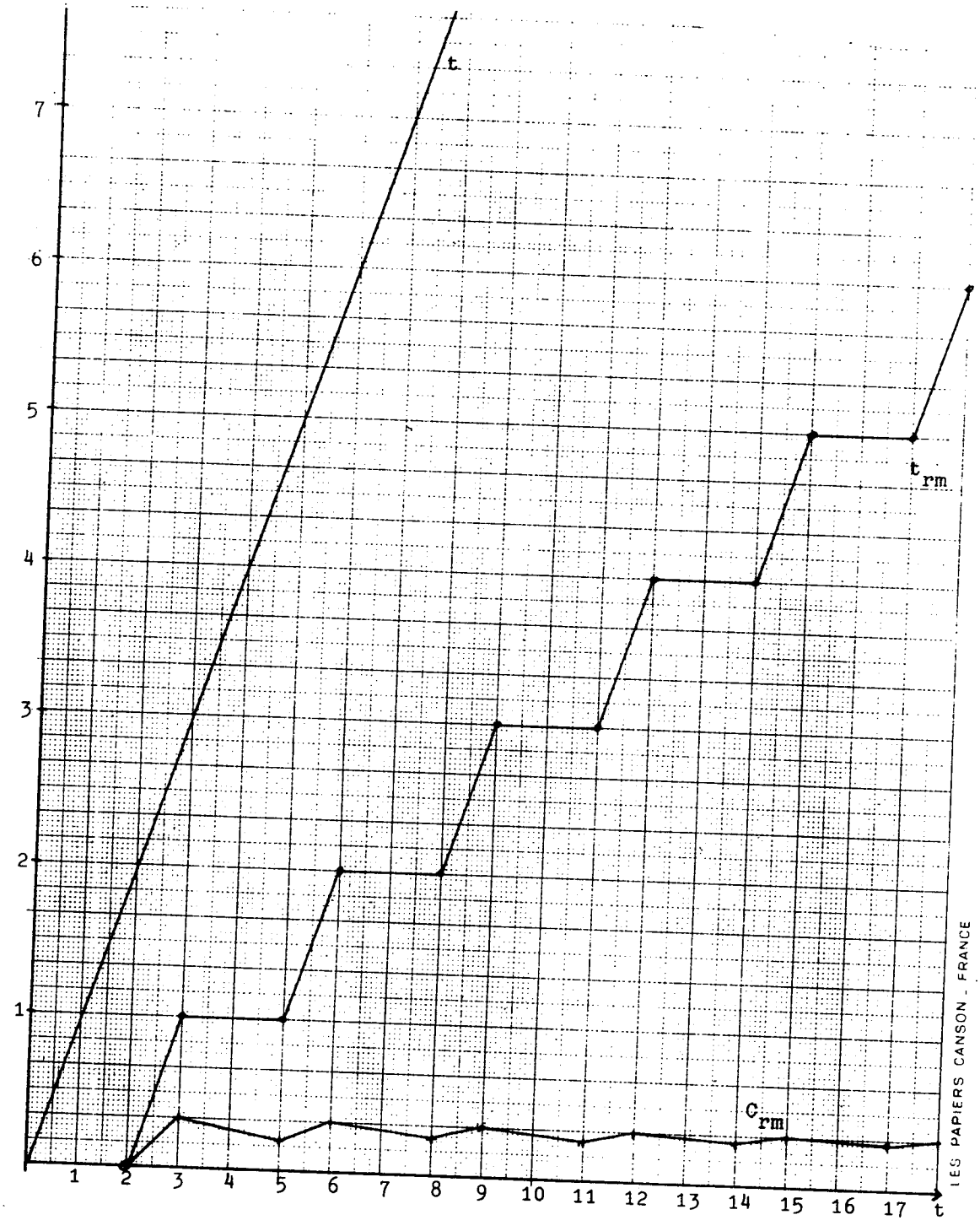
* dans un environnement paginé, le programme de r.m. doit directement communiquer avec le système de pagination.
  (ceci n'est pas le cas sous CP/CMS.)

Remarque : sur la nature de c_{rm}.
Du point de vue mathématique

$$c_{rm} = \frac{\text{temps de calcul total}}{\text{temps passé dans le rm}} = \frac{t}{t_{rm}}$$

est une fonction du temps total t, dont la courbe est en dents de scie de plus en plus fines. Elle prend une valeur significative seulement pour t assez grand, c'est-à-dire des exécutions longues. Une exécution courte peut très bien se terminer sans aucun appel du r.m.

Les rapports entre t, t_{rm}, et c_{rm} sont esquissés dans la figure suivante pour laquelle on fait l'hypothèse simplificatrice que le r.m. est déclenché après 2 sec. d'exécution normale et qu'il dure 1 sec.



La situation devient nettement plus compliquée si on inclut le coût de la pagination, puisque un r.m. avec tassement influe sur le nombre de fautes de pages provoquées par les calculs ultérieurs.

2.2 - La pile

La pile d'exécution de LISP/360 correspond aux deux piles de la machin MIL. Pour savoir à laquelle des deux piles appartient un élément donné, il suffit de le comparer aux limites de l'espace listes. L'implantation physique de la pile rend onéreuse toute modification de sa taille : il faudrait translater l'ensemble de l'espace listes (et les fonctions dans MI_2) en mettant à jour tous les pointeurs.

Etant donné que jusqu'à ce jour la taille standard de la pile (6000 mots) a été suffisante, une extension dynamique n'a pas été implantée. Par contre, la création d'un module LISP avec une pile non-standard est simple et rapide. Les fonctions SAVE et OBLIST (voir 2.2.2) tiennent compte de cette possibilité.

2.3 - MI - l'espace des fonctions

En LISP/360 l'espace MI est coupé en deux parties : MI_1 et MI_2 (schéma p. 74) MI_1 contient l'ensemble des fonctions standard -l'interpréteur proprement dit, les fonctions arithmétiques, d'entrées-sorties et quelques dizaines de fonctions de traitement de listes. MI_1 occupe les mémoires basses du système. Bien entendu, cette partie est figée à la génération du module.

Si l'utilisateur veut définir lui-même des fonctions codées en langage machine il peut dynamiquement demander des extensions de MI dans les mémoires hautes du système (MI_2 dans le schéma p. 74) Ceci est possible de deux façons :

* A l'aide du chargeur CMS on peut introduire dans le système des fonctions assemblées sour CMS. Ceci est expliqué au 3.

* En utilisant le compilateur. Celui-ci, lui-même situé dans MI_2 (par chargement), dispose d'un espace appelé BPS (Binary Program Space) qui est une partie libre de MI_2. La compilation d'une λ-expression produit des instructions dans BPS, alors que la place occupée par la λ-expression dans l'espace listes est libérée.

3. OUTILS DE CONSTRUCTION DE SYSTEMES.

La plupart des applications de LISP constituent des programmes de taille importante (par exemple REDUCE occupe ~ 140K octets) pour lesquels il est indispensable d'avoir des nombreuses facilités :

* modification de la taille des espaces du système, en particulier la taille de l'espace listes M et celle de MI.
* introduction rapide de grandes masses de fonctions et de données.
* définition de "modules" facilement accessibles à d'autres utilisateurs.

L'ancienne version de LISP/360 ne répond à aucune de ces contraintes.

Dans ce chapitre, nous exposons les solutions offertes dans LISP/CMS ; leur réalisation s'est heurtée à des problèmes liés à la conception de base du système ; c'est pourquoi les réalisations sont moins générales que, par exemple, les possibilités de MACLISP. Dans la pratique, ces solutions se sont montrées très utiles, et suffisantes pour les applications actuelles.

3.1 Chargement dynamique - la fonction LOAD.

Toute extension du système LISP/CMS est réalisée au moyen d'un appel du chargeur CMS, accessible au niveau LISP par la fonction LOAD à deux paramètres

LOAD fi $\begin{bmatrix} L \\ BPS \\ NIL \end{bmatrix}$

L'appel de LOAD provoque d'abord le chargement du fichier fi TEXT à l'emplacement suivant immédiatement la limite actuelle du système LISP. Cette limite est mise à jour.

L'appel du chargeur permet de réaliser deux choses différentes :

D'une part l'acquisition de place mémoire, utilisée pour aggrandir l'espace listes ou le BPS. Dans ce cas, l'appel du chargeur est fictif et équivalent (en CMS) à un appel de GETMAIN.

D'autre part, le fichier fi TEXT peut effectivement contenir des instructions machine sous forme binaire translatable.

Le type d'une extension introduite par l'appel du chargeur est indiqué par le deuxième paramètre de la fonction LOAD :

a) L - il s'agit d'une extension de l'espace listes. La constante BOTTOM qui contient l'adresse du dernier élément de l'espace listes est mise à jour et réfère le dernier double-mot chargé.

L'extension n'est pas physiquement intégrée dans la liste libre. Cela sera fait lors du prochain appel du r.m.

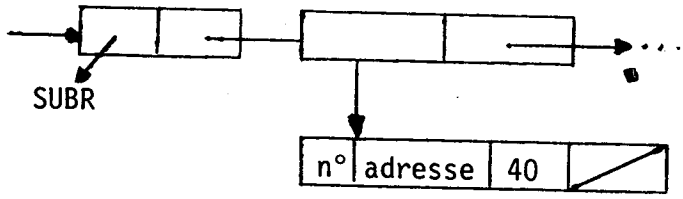
On voit que, s'il existait déjà un espace MI_2, celui-ci sera considéré comme extension de l'espace listes par le r.m. et automatiquement transformé en espace listes.
L'ordre dans lequel on définit les extensions du système est donc très important.

b) BPS - il s'agit d'un "Binary Program Space", un espace amorphe qui sert actuellement au compilateur et à l'éditeur (zone de travail). C'est une partie libre de MI_2.

c) NIL - il s'agit de fonctions codées en langage machine, donc d'une extension de MI_2. Il faut alors créer un certain nombre de liens entre ces fonctions et l'espace listes.

Il existe trois types de liens :

1) Une fonction dans MI est accessible au système par l'intermédiaire d'une propriété SUBR ou FSUBR (dans la liste de propriétés de l'atome qui est le nom de la fonction), indiquant l'adresse de la fonction et le nombre d'arguments :



voir aussi 2.1.

2) Une fonction dans MI doit connaître l'adresse des S-graphes "quotés", des arguments d'une instruction PUSHQ. Eventuellement ces graphes doivent être crées, et il faut les protéger contre le r.m. (voir p.76).

3) Pour partager une variable, les fonctions dans MI utilisent un "doublet SPECIAL" qui se trouve sur la liste de propriétés du nom de la variable. Les adresses de tels éléments (qu'il faut éventuellement créer) doivent être insérées dans le code des fonctions.

Cette édition de liens ne peut être assurée par le chargeur CMS, c'est donc la fonction LOAD qui l'effectue, suivant les indications fournies par une zone à la fin du fichier chargé.

La fonction LOAD joue un rôle central dans les mécanismes d'extension de LISP/CMS. En particulier, sans cette fonction il n'aurait guère été possible d'intégrer dans le système les outils de mise au point décrit au chap. 4. Au 3.2.4 nous montrons sur un exemple le gain de temps réalisé grâce au LOAD.

3.2 Sauvegarde :

A 'laide d'une série d'appels de la fonction LOAD et en lisant d'autres données sous forme de S-expression, l'utilisateur peut définir tout environnement voulu.

Il peut ensuite le conserver dans un fichier et le restituer à nouveau en une seule commande, sans refaire toute la construction et en un temps bien plus rapide.

Cette possibilité, offerte par les fonctions SAVE et OBLIST, permet en particulier

* de conserver des configurations complexes - appelées sous-systèmes-sous une forme facilement accessible et avec un minimum de place sur disque.
* d'interrompre un travail commencé - à la fin d'une session CMS, par exemple - et de le reprendre un autre jour.

La fonction SAVE (fi) crée un fichier fi OBLIST contenant

- les adresses et la taille des espaces P,M et MI_2
- le contenu de M et de MI_2.

Ces informations définissent complètement l'état du système et permettent de le restituer à partir d'une configuration quelconque.

En général M, l'espace listes, est très grand mais contient peu d'informations effectives, la plus grande partie étant occupée par des éléments non-actifs. C'est pourquoi l'espace listes n'est pas simplement recopié dans le fichier. On effectue d'abord un tassement (2.1.3) pour copier ensuite seuls les éléments actifs.

Nous avons déjà indiqué que la présence de S-graphes "QUOTES" dans les fonctions de MI_2 limite les possibilités de tassement. Ceci impose la restriction suivante à la fonction SAVE :

Si des fonctions dans MI_2 utilisent de tels S-graphes, la création de la configuration et la sauvegarde doivent se dérouler en trois étapes :

1) Introduction dans le système de tous les S-graphes "quotés" et premier tassement. Leur liste est fournie par l'assembleur ACMS (3.3.3)

2) Chargement des fonctions dans MI_2.

3) Sauvegarde dans un fichier. L'algorithme de tassement ne touche pas aux données qui sont déjà tassées, les adresses critiques ne sont pas modifiées.

Cette solution est relativement facile à mettre en oeuvre, mais elle n'est pas élégante et demande une certaine attention de la part de l'utilisateur qui en est largement récompensé par un résultat très agréable à utiliser.

3.3 Utilisation du compilateur :

L'histoire du compilateur utilisé en LISP/360 est parallèle à celle de l'interpréteur, et les deux programmes ont beaucoup de défauts en commun.

Encore plus que l'interpréteur, le compilateur porte des traces d'un grand nombre de modifications effectuées depuis la première écriture au début des années 60. En particulier, une phase d'optimisation a été rajoutée.

Le compilateur prend comme argument une fonction définie par une λ-expression qui est la propriété EXPR du nom de la fonction, et il la traduit en instructions en langage machine en passant par plusieurs étapes. Nous en distinguons deux :

1) La compilation proprement dite, au sens du 1-2.4 : la fonction est traduite en une liste d'instructions assembleur, chaque instruction étant représentée par une sous-liste. Cette étape comporte en particulier une transformation du programme en un LISP standardisé.

2) A partir de la liste des instructions, l'assembleur LAP (LISP Assembly Program) génère des instructions en langage machine qui sont placées dans BPS (la partie libre de MI_2).
Après l'assemblage la propriété EXPR de la fonction est supprimée et remplacée par une propriété SUBR indiquant l'adresse de la fonction (la place occupée par la λ-expression sera reprise par le prochain r.m.).

Cette utilisation classique du compilateur présente des inconvénients bien connus :

* les résultats se trouvent en mémoire centrale et sont perdus à la fin de l'exécution : il faut donc recompiler à chaque utilisation.

* le compilateur a besoin d'une configuration de LISP relativement grande qu'on est obligé de garder par la suite, même si l'on n'en a pas besoin. En plus, le compilateur lui-même occupe de la place qui n'est plus utilisée après la compilation.

Il est clair que le compilateur est un programme désuet qu'il faudrait entièrement refaire. Nous nous sommes limités dans une première étape à l'implantation de la génération de code binaire translatable avec le moins d'effort possible. Cette solution imparfaite rend pourtant bien service.

Sur les bases de ce travail, il nous a été possible de définir les principes d'un nouveau compilateur dans un cadre plus général (I-4) .

Nous avons donc écrit une fonction ACMS qui peut remplacer LAP dans la deuxième étape de la compilation ACMS génère deux fichiers :

1) Un fichier contenant des instructions assembleur dans le format requis par l'assembleur standard de CMS, suivies d'une zone d'édition de liens pour la fonction LOAD (3.1)

Une fois assemblées les fonctions contenues dans le fichier peuvent être amenées dans toute configuration de LISP grâce à la fonction LOAD en un temps très court.

2) Les données nécessaires pour la sauvegarde des fonctions (3.2).

Il est évident que l'inconvénient de cette méthode réside dans le temps considérable consommé par l'assembleur CMS (dont la lenteur est un fait bien connu).

3.4 Comparaison des méthodes d'entrée de données :

Il ressort des paragraphes précédents qu'en LISP/CMS il existe plusieurs méthodes pour introduire des données dans le système.

⁂ les données en forme de S-graphe peuvent être entrées
1) par lecture de S-expressions
2) par restitution d'une OBLIST.

⁂ des fonctions dans MI_2 peuvent être obtenues par
3) compilation à partir de λ-expressions
4) assemblage à partir d'un fichier LAP
5) chargement à partir d'un fichier TEXT (binaire translatable)
6) restitution d'une OBLIST.

Seule la méthode 4 n'est pas intéressante. Dans un cas concret, le choix parmi les autres méthodes se fait en fonction de

- la taille des données
- le caractère définitif ou expérimental
- l'application envisagée.

Il est à noter que les méthodes les plus efficaces pour des grandes masses de données - 2, 5, 6 - n'existent pas dans LISP/360, ni dans d'autres version de LISP, à l'exception de MACLISP. Leur utilité est bien illustrée par l'exemple de la génération du compilateur qu'il faut considérer comme un programme assez grand à caractère définitif.

En LISP/360, le compilateur est normalement généré par la méthode 4 et ensuite conservé sous forme d'un module CMS. La génération demande ∿ 550 sec. CPU, le module occupe ∿ 5000 cartes.

En LISP/CMS, le compilateur est généré par chargement,(l'ensemble des fonctions a donc été assemblé une fois sous CMS ce qui a demandé ∿ 200 sec), et ensuite conservé à l'aide de la fonction SAVE. Le chargement proprement dit dure ∿ 1 sec., l'opération totale 6 sec. (la sauvegarde nécessite deux appels du tassement, comme il est expliqué au 3.2). Le fichier généré occupe ∿ 900 cartes et la restitution dure ∿ 0.3 sec.

Les gains de temps et de place sont donc considérables (d'autant plus que la génération du compilateur est une opération assez fréquente, nécessaire après chaque mise à jour du système).

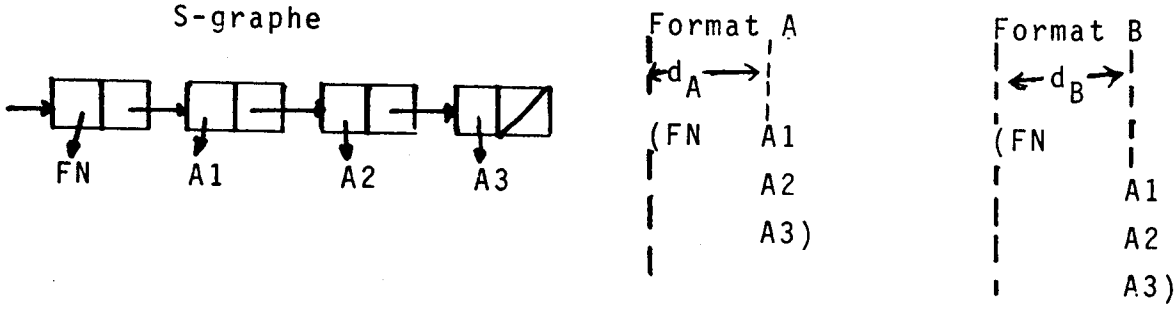
4. AIDE A LA MISE AU POINT DE PROGRAMMES

4.1. Jolie impression

La fonction PRINT décrite au § I.3.1. imprime un S-graphe sous forme d'une expression linéaire sans aucune mise en page. Ce format est quasiment incompréhensible dès que la longueur de l'expression dépasse une ligne. La fonction PRINT est donc peu utile pour l'écriture de fonctions, par exemple, dans des messages d'erreur ou dans une liste de sortie.

De ce problème vient une idée importante : l'impression de S-graphes dans un format structuré, appellé "jolie-impression" (pretty printing). La jolie-impression est fondée sur trois principes :

1) Mettre en évidence la structure arborescente des graphes sur la feuille d'impression. Le schéma suivant montre deux solutions possibles pour la forme (FN A1 A2 A3).



(D'après la grammaire du paragraphe I.2.1, cette liste représente un appel de la fonction FN avec les arguments A1, A2, A3).

Dans les deux cas, les arguments de la fonction se trouvent dans les lignes successives, alignés, et décalés par rapport à la fonction.

Le format A utilise une ligne de moins, mais le décalage d_A entre fonction et arguments est variable suivant la longueur du nom de la fonction.

Le format B peut utiliser un décalage d_B constant.
Bien entendu, ces règles d'écriture s'appliquent récursivement, chaque argument Ai pouvant à son tour être une forme non atomique. Comme la longueur de la ligne d'impression est finie, la profondeur des arbres qu'on peut imprimer ainsi est limitée. En général, l'écriture A arrive plus vite à la fin de la ligne que l'écriture B.

2) L'application récursive de la première idée conduit à une écriture très étalée d'une expression : avec la règle B, un seul atome est écrit sur chaque ligne, avec la règle A il y en a en général un ou deux (sauf s'il y a des fonctions à un argument imbriquées : (F1 (F2 (F3 ... x) ...))).
Ceci nuit à la compréhension des expressions - il faut réaliser un compromis entre l'écriture purement linéaire et l'écriture purement arborescente : quand une expression est simple, utiliser l'impression linéaire ; quand elle est compliquée, l'étaler sur plusieurs lignes. La mesure de complexité choisie dans LISP/CMS est le nombre de paires de parenthèses dans l'écriture de l'expression. Dans d'autres systèmes une expression est considérée simple si elle tient dans la partie restante de la ligne (ce qui est légèrement plus coûteux en temps machine).

3) Les principes 1 et 2 donnent des bons résultats, sauf pour les formes spéciales (COND, PROG, QUOTE, ...). Il faut donc prévoir des formats spéciaux (ceci se programme facilement à l'aide d'une propriété). Dans le cas du PROG, par exemple, on fait la distinction entre les instructions, écrites l'une en dessous de l'autre, et les étiquettes placées devant les instructions.

Ces principes ne définissent pas un algorithme unique de jolie-impression. Bien au contraire - le choix est grand et fait intervenir des critères esthétiques, techniques et algorithmiques. Ira GOLDSTEIN a analysé quelques aspects théoriques de cette question dans une publication décrivant le "pretty printer" de MACLISP [14].

Voici l'algorithme utilisé dans LISP/CMS :

Jolie impression d'une expression x

I Si x est simple, l'imprimer linéairement (PRINT).
X est simple, si son impression nécessite moins de LIM paires de parenthèses, LIM étant une variable globale (valeur par défaut : 5).

II X n'étant pas simple, il est de la forme (fn a1 ... an) ou (fn a1 ... a_{n-1}.a_n).
S'il y a un format spécial associé à la racine de x (fn), utiliser ce format spécial.

III Sinon, étaler x

* suivant le format A s'il y a moins de deux arguments
* suivant le format B s'il y a deux arguments ou plus, avec un décalage de trois positions entre la fonction et ses arguments.

Sauf dans les formats spéciaux, la position du premier caractère d'une ligne (divisée par 3) indique à quel niveau de l'arborescence on se trouve, la racine ayant le niveau 0.

Exemple : toutes les fonctions LISP données dans cette thèse sont écrites à l'aide de cet algorithme.

Cet algorithme extrêmement simple donne des bons résultats. L'utilisateur qui ne serait pas de cet avis peut facilement définir lui-même des nouveaux formats spéciaux.

Il s'est avéré que la profondeur des expressions à imprimer dépasse très rarement la valeur de 10 ; la limitation théorique imposée par la longueur de la ligne ne joue donc pas dans la pratique. Cela est probablement dû au fait que la conception d'expressions plus complexes est difficile, et que le programmeur préfère les décomposer. Cette limite correspond aux niveaux d'écriture d'une programmation structurée.
Par contre, des programmes écrits dans un langage de niveau plus élevé et ensuite traduits en LISP ont tendance à être plus complexes.

Utilisation :

La jolie impression facilite la compréhension des programmes et est un outil important dans la mise au point : grâce au décalage suivant les niveaux des arborescences, on peut voir les erreurs de parenthésage, puisqu'elles se traduisent par une mauvaise structure de l'arbre. Il n'est plus besoin de compter les parenthèses, ni de chercher les parenthèses gauches correspondantes aux parenthèses droites. Cela peut sembler naïf, mais l'expérience a montré que le programmeur est ainsi libéré d'un travail long et fastidieux.

L'utilisation pour la mise au point impose une propriété qui complique la programmation de la jolie-impression :

Toute expression, même syntaxiquement fausse, doit être correctement imprimée.

Du point de vue pratique, la jolie-impression est naturellement utilisée dans l'éditeur intégré dans LISP, décrit dans le chapitre suivant.

4.2 L'éditeur :

4.2.1 Motivation :

La mise au point d'un programme LISP sous CMS se fait selon le cycle suivant :

I Appel de l'éditeur CMS
II Mise à jour ou création du fichier source
III Retour à CMS
IV Appel de LISP
V Lecture du fichier source - en cas d'erreur on passe à VII
VI Tests - création de résultats - détection d'erreurs
VII Retour à CMS (perte des résultats partiels).
VIII Reprise au I.

Ce cycle de mise au point n'est pas satisfaisant pour plusieurs raisons, surtout s'il s'agit de gros programmes :

a) La lecture du fichier source (étape V) peut être coûteuse en temps machine (10 sec. n'est pas une valeur exceptionnelle), mais toute correction, même si elle est purement locale, nécessite une nouvelle lecture

de l'ensemble du fichier. Ceci est particulièrement ennuyeux pour les erreurs de parenthésage, si fréquentes dans la première phase de mise au point.

b) Si on détecte une erreur au milieu d'une exécution longue, on perd tous les résultats partiels, parce que l'on est obligé de quitter LISP pour la corriger. Ceci fait perdre du temps au programmeur (c'est coûteux en temps machine également).

En partie ces difficultés sont dues aux problèmes discutés au 3 (outils de création de système), mais la raison essentielle réside dans les changements d'environnement, les passages fréquents entre l'éditeur CMS, CMS, et LISP.

Afin d'éviter ces changements, il faut intégrer un éditeur dans le système LISP ; ceci conduit à un nouveau schéma du cycle de mise au point :

I Appel de LISP
II Lecture du fichier source, s'il en existe un.
III Tests - création de résultats - détection d'une erreur.
IV Appel de l'éditeur intégré - correction (ou création) d'une fonction.
V Reprise au III, éventuellement après avoir sauvegardé la fonction corrigée dans le fichier source, ou retour à CMS.

On constate des différences importantes par rapport au schéma précédent :

a) La lecture du fichier source ne se fait qu'une fois, en dehors de la boucle III-IV-V.

b) Les résultats partiels (obtenu au III) ne sont pas perdus lors de l'édition[1].

c) On édite une seule fonction à la fois - le temps machine nécessaire pour une correction est donc proportionnel à la taille de la fonction (qui n'est jamais très importante), alors que dans le premier schéma ce temps est proportionnel à la taille du programme entier qui, lui, peut devenir très grand.

1 Par résultats partiels, nous entendons ici les résultats faisant partie des listes de propriétés. Jusqu'à ce jour, nous n'avons pas éprouvé le besoin d'un système de "debugging" permettant des modifications en cours d'un appel de la fonction universelle.

Il en résulte des gains appréciables de temps de mise au point, de la part du programmeur aussi bien que de la part de la machine, et ces gains augmentent très vite avec la taille des programmes.

A titre de comparaison, voici deux chiffres typiques de temps correction d'une erreur :

Schéma 1 : étapes II, IV, V : 2 sec pour un petit programme,
5 sec pour un programme moyen.
Schéma 2 : étapes IV, V : 0.3 sec par fonction (entre 0.1 et 0.9).

4.2.2 *Réalisation* :

A l'édition, on considère un S-graphe comme une chaine de caractères et non pas comme une arborescence. C'est pourquoi l'éditeur procède en trois étapes : conversion en images de cartes, édition des cartes, construction du nouveau S-graphe.

1) Conversion en chaine de caractères. L'appel de l'éditeur - EDIT(s) - provoque l'écriture du S-graphe s sous forme d'images de cartes dans une zone d'édition prise dans la partie libre de MI, BPS.

L'écriture se fait bien entendu à l'aide de la jolie-impression mettant en évidence la structure arborescente de l'expression.

Si la zone d'édition contient déjà l'expression à éditer (parce que celle-ci vient d'être éditée), cette étape est sautée.

2) L'édition proprement dite se fait à l'aide d'un éditeur conversationnel semblable à l'éditeur de CMS ; la plupart des utilisateurs ne trouveront pas de différence puisque toutes les requêtes importantes sont disponibles :

BOTTOM
CHANGE
DELETE
FIND
INSERT
INPUT
LOCATE
NEXT
PRINT

REPLACE
TOP
UP

(Il manque donc notamment les requêtes concernant la tabulation, inutile dans ce contexte, et la possibilité de macro-requêtes).

En fait, cet éditeur est complètement indépendant de LISP et peut être utilisé dans un contexte tout-à-fait différent.

Remarque : Il aurait été souhaitable d'utiliser directement l'éditeur de CMS, mais il s'est avéré plus facile d'écrire en nouvel éditeur simple que de trouver des spécifications précises concernant les modalités d'appel de l'éditeur CMS (qui vraisemblablement est un programme beaucoup plus grand).

Nous pensons que l'éditeur LISP a une structure claire et modulaire et qu'il peut facilement être intégré dans un autre système.

3) Le retour à LISP peut se faire par trois requêtes différentes :

* FILE - La fonction READ (I-3.1.5) est appelée pour convertir la chaine de caractères dans la zone d'édition en un S-graphe. Celui-ci est le résultat de la fonction *EDIT. La zone d'édition n'est pas modifiée et peut être réutilisée si on veut de nouveau éditer le même S-graphe.

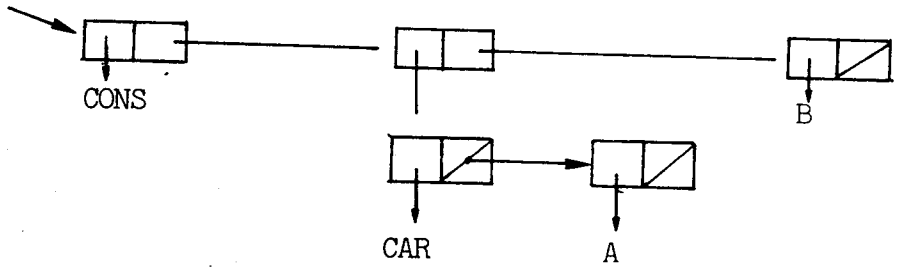
* Q - retour à LISP avec le résultat NIL. La zone d'édition n'est pas modifiée. Ceci permet de regarder (à l'aide de la requête PRINT) des parties d'une fonction longue à très peu de frais.

* QUIT - Retour à LISP avec le résultat NIL, et destruction de la zone d'édition. Ceci est utile après une fausse manoeuvre pour forcer l'écriture de l'expression à éditer au prochain appel (première étape).

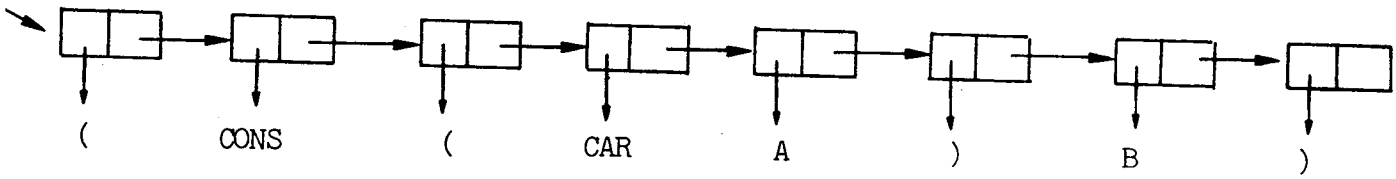
Bien entendu, l'éditeur ne sert pas seulement à l'édition de fonctions, mais est aussi à la disposition de l'utilisateur pour l'édition de toute sorte de données (fonction EDP).

4.2.3 *Autre solution* :

Un éditeur d'images de cartes est une solution très commode et performante, mais il nécessite l'utilisation d'un langage différent de LISP (en occurence l'assembleur). Il est également possible d'écrire un éditeur entièrement en LISP. Un tel éditeur a été bien documenté dans [38]. Au lieu de travailler avec des images de cartes, il utilise une liste linéaire d'objets primitifs ("token"). L'expression (CONS (CAR A) B), c'est-à-dire le S-graphe.



serait traité sous la forme



Cette solution élégante a l'avantage d'être parfaitement transportable.

5. REDUCE

Parmi les systèmes de calcul formel (voir [37] pour une présentation des systèmes actuels de calcul formel) REDUCE dont l'auteur est A.C. HEARN de l'Université de Utah, est probablement le plus répandu. C'est aussi le système le plus ouvert à l'utilisateur, en ce sens qu'il est bien documenté, écrit dans un langage facilement compréhensible, et parfaitement extensible. C'est donc un excellent outil de recherche dans le domaine du calcul formel dont l'évolution est rapide. En comparaison, MACSYMA, le système le plus sophistiqué du point de vue algorithmique n' est accessible qu'au spécialiste.

Pour LISP/CMS l'implantation de REDUCE a été une véritable épreuve : REDUCE exploite toutes les possibilités de LISP, y compris le compilateur et quelques fonctions "obscures" jamais testées auparavant. A cette occasion il s'est avéré que les simplifications dans les conventions de liaison avec lesquelles avait commencé le développement de LISP/CMS améliorent légèrement la performance du système : les fonctions compilées occupent 7 % moins de place, et le temps d'exécution est réduit de 13% (gains réalisés à la génération de REDUCE).

En plus, il fallait opérer quelques modifications de l'interpréteur même, qui ont confirmé la souplesse d'emploi de LISP/CMS. La modification la plus importante fut le remplacement de l'ensemble des fonctions arithmétiques par des fonctions à précision arbitraire (illimitée). Dans ce cas, un nombre est représenté par une liste de "chiffres" à base $2^{31}-1$ (taille d'un mot machine). Bien entendu, cela s'intègre parfaitement dans la gestion de l'espace listes de LISP. Les programmes d'arithmétique distribués avec REDUCE sont dérivés du système SAC-I de COLLINS [9].

Les gros systèmes de calcul formel écrit en LISP- MATHLAB, SCRATCHPAD, REDUCE et MACSYMA - ont montré les avantages de LISP au niveau algorithmique aussi bien que l'insuffisance pratique de la plupart des implantations. Ainsi REDUCE souffre en particulier de la mauvaise qualité du compilateur de LISP/360 et de l'inefficacité des calculs arithmétiques (une addition coute environ 100 instructions par opérande !). Déjà une révision rapide du compilateur par l'équipe de A.C. HEARN a amélioré les résultats de 20 %. Et ce n'est pas par hasard si l'implantation la plus efficace de LISP, MACLISP, a été développé parallèlement avec MACSYMA.

C'est finalement au contact du calcul formel que LISP est sorti des laboratoires de recherche pour devenir un langage d'application.

C O N C L U S I O N

Le travail présenté dans cette thèse fait nettement apparaître deux étapes : la réalisation de LISP/CMS et une phase de réflexion sur la structure d'un système LISP.

Une première conclusion sur LISP/CMS a été donnée à la fin du paragraphe II.1. Le développement du système est en principe terminé, et il est utilisé à Grenoble et dans plusieurs autres universités. Un travail simple de programmation permettrait de le transformer assez rapidement en LISP1.6 qui est un dialecte de LISP très utilisé.

Par contre, la représentation en mémoire des S-graphes limite les possibilités d'amélioration; pour surmonter ces limitations il faudrait écrire un système entièrement nouveau.

L'étude de meilleures solutions des problèmes fondamentaux posés par une implantation de LISP fait partie de la deuxième étape, laquelle, étant au départ un travail plutôt théorique, se retourne maintenant vers les applications. Cependant, notre définition de MIL, énoncée avec l'intention de montrer des principes, est incomplète et imparfaite en tant que modèle d'implantation :

Elle est incomplète parce qu'elle ne traite pas certains aspects importants d'un système, comme le traitement d' erreurs et le traitement d'interruption; le traitement de nombres n'a pas été assez étudié en détail. Il serait aussi intéressant de tenir compte, au niveau de la machine MIL, de structures de contrôle plus complexes que l'appel hiérarchique de fonctions, tel que le "spaghettit stack" de BOBROW [3].

Elle est imparfaite, comme nous l'avons vu au paragraphe I.4, parce que le langage MIL est trop pauvre pour être directement utilisé dans une implantation. Le volume des fonctions compilées serait prohibitif et les temps d'exécution mauvais.

L'élaboration plus détaillée de ce modèle doit se faire en liaison avec une réalisation car ceci constitue la seule justification réelle d'un tel travail et permet d'en éprouver la validité. Déjà un projet très limité, actuellement en cours, devrait permettre d'approfondir le modèle et donner des résultats dans les mois à venir.

La simplicité de la machine MIL n'est pas obligatoirement un défaut pour son implantation : elle suggère aussi une solution microprogrammée (ou cablée). La vitesse d'exécution d'un tel système serait essentiellement fonction du nombre d'accès à la mémoire de S-graphes M. En effet, quelques publications récentes sur la microprogrammation de LISP prévoient des gains de place et de temps importants, et proposent de nombreuses idées intéressantes. Déjà deux tels projets sont en cours de réalisation aux Etats-Unis.

Quelle que soit la technique d'implantation de la machine MIL, le compilateur LISP sera un facteur dominant pour l'efficacité de l'ensemble du système. L'étude d'un compilateur performant est un sujet très intéressant, lié à des problèmes théoriques tel que les transformations de programmes (I.3.6) et la métacompilation (I.2.4). Par contre, nous avons bien vu que les principes de base d'un tel compilateur peuvent être très simples. On pourrait aussi utiliser des techniques de compilation développées pour d'autres langages. Cette possibilité est encore confirmée par la réflexion du paragraphe suivant.

Dans cet exposé nous avons utilisé un certain nombre de langages : LISP, RLISP, la machine MIL, l'assembleur 360.

Nous allons maintenant établir le lien entre ces langages.

La première idée qui vient à l'esprit est de penser à une relation hiérarchique, allant des langages de "bas niveau" vers ceux de "haut niveau" :

langage machine → MIL → LISP → RLISP → REDUCE

mais cet ordre linéaire n'est pas satisfaisant :

1) En ce qui concerne la sémantique LISP et RLISP sont entièrement équivalents et devraient donc se trouver au même niveau.

2) Le traitement en machine d'un de ces langages fait passer d'un niveau au niveau immédiatement à gauche : RLISP est traduit en LISP, LISP est traduit en un macro-code, etc... Seul le traitement de REDUCE fait une exception, puisqu'il est traduit en LISP, en sautant l'étape intermédiaire de RLISP.

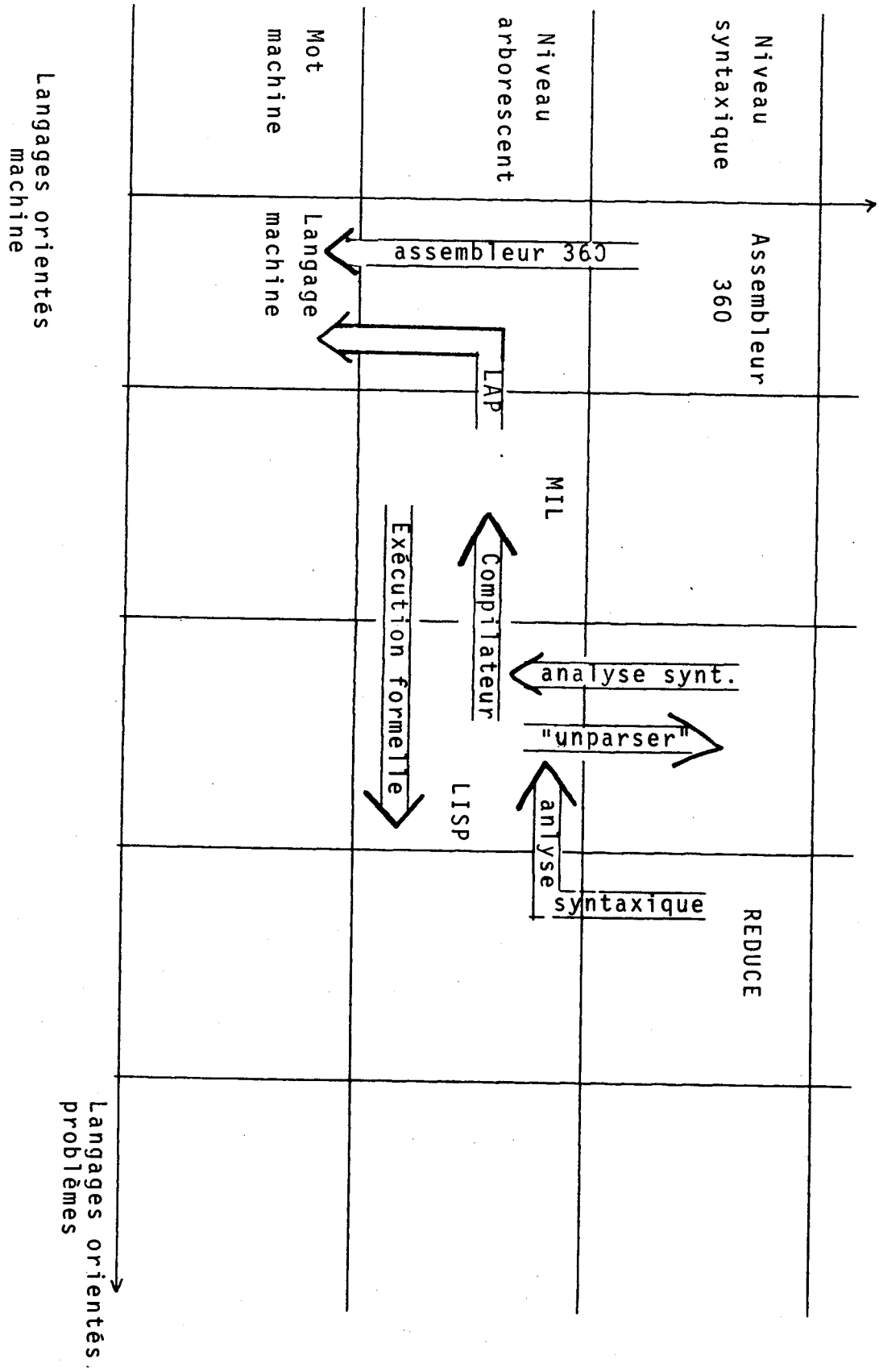
Il est plus naturel d'utiliser un schéma à deux dimensions comme il est fait dans la figure page 105.

L'axe horizontal correspond à l'ordre donné ci-dessus et va des langages primitifs ou "orientés machine" vers les langages évolués ou "orientés problème".

La deuxième dimension correspond à la représentation d'un programme. Il nous semble utile de distinguer trois niveaux de représentation :

1) sous forme de mots en machine,
2) sous forme arborescente,
3) par une phrase conforme à une grammaire (niveau syntaxique).

On peut exprimer un programme écrit dans un langage L donné à l'aide de chacune de ces représentations quelle que soit la position de L le long de l'axe horizontal.


Langages orientés machine
Mot machine
Niveau arborescent
Niveau syntaxique
Langage machine
Assembleur 360
assembleur 360
LAP
MIL
Compilateur
Exécution formelle
analyse synt.
"unparser"
LISP
analyse syntaxique
REDUCE
Langages orientés problèmes

Alors que la position horizontale définit la "puissance" d'un langage, le niveau vertical détermine les traitements que peut subir un programme, et en particulier les possibilités de passage à un autre niveau. Pratiquement tous les traitements décrits dans cette thèse réalisent la traduction d'un programme d'une case de la figure vers une case adjacente (sinon ils se décomposent en deux phases).

Le niveau syntaxique est celui de la communication homme-machine(1).Il est relativement bien adapté à la compréhension et à la conception de l'homme (c'est le "sucre syntaxique") et le passage automatique au niveau arborescent est un des sujets les mieux maîtrisés de l'informatique : il s'agit de l'analyse syntaxique. Dans la mesure où ce passage suit un trajet purement vertical (dans la figure), aucune information n'est perdue et une traduction dans l'autre sens doit en principe être possible.

Les traitements locaux au niveau syntaxique d'un programme incluent l'édition avec un éditeur de textes et une macro-expansion.

Le niveau arborescent est fastidieux à manipuler pour l'homme - la programmation classique en LISP en est un bon exemple - mais il est bien adapté au traitement automatique.

En restant à ce niveau, on peut effectueur des transformations de programmes au sens le plus large, par exemple en vue d'une optimisation.

---

1) du moins à l'heure actuelle - les langues naturelles que l'on peut situer à un niveau élevé de l'axe des y sont utilisables seulement dans la mesure où on les limite à une syntaxe formelle.

Nous avons vu qu'il est possible de passer au niveau syntaxique, ce qui permet de donner l'image de ces transformations à ce niveau (exemple du I.3.6.)

La compilation (au sens limité du I.2.4) opère un passage horizontal vers un langage de plus bas niveau dont MIL est un exemple.

Pour les langages orientés machine, le passage du niveau arborescent au langage machine est réalisé par un (macro-) assembleur (LAP dans le cas de LISP).

Dans la mesure où LISP permet une manipulation très aisée d'arborescences, ce langage occupe une place privilégiée parmi les langages de programmation, qui le prédestine à ce type de traitement, c'est-à-dire la manipulation de fonctions en tant qu'objets. Les recherches en programmation automatique décrits dans [15] en sont un bon exemple.

Le niveau machine est celui de l'exécution; toute autre opération à ce niveau est extrêmement fastidieuse.

Une comparaison avec un compilateur classique montre une analogie frappante : un tel compilateur, qui effectue un passage "en diagonale" d'une case dans la position de RLISP vers le langage machine, n'accomplit pas son travail directement - il opère en plusieurs phases qui s'apparentent naturellement aux passages entre deux cases de la figure (sauf dans le cas du traitement de types, inexistant en LISP).

On peut donc dire que l'ensemble des programmes de traitement d'un système LISP met à la disposition de l'utilisateur les mêmes outils qu'un compilateur classique; la différence réside dans le fait qu'en LISP ces morceaux sont complètement séparés, dans une grande mesure accessibles à l'utilisateur et qu'ils sont complétés par un interpréteur.

Si l'on perfectionnait les outils actuellement existant dans ce sens, il serait possible de construire un système interactif dans lequel l'utilisateur contrôlerait les étapes successives de traitement, en intervenant quand il le désire avec des directives globales ou en indiquant des transformations précises. Un tel système pourrait ne pas être limité au seul langage LISP, car l'analogie avec un compilateur classique évoquée ci-dessus permettrait de rapprocher les travaux associés à LISP des développements d'autres langages. Par exemple, on peut se demander dans quelle mesure l'analyseur syntaxique et le traducteur LISP→RLISP accepteraient des programmes écrits dans d'autres langages. Les facilités de traitement de listes amènent à envisager un système de traduction entre langages de haut niveau. Les progrès que connaît la programmation automatique montrent qu'en imposant quelques restrictions raisonnables, on devrait arriver à des résultats intéressants.

A long terme, un tel développement permettrait de réduire l'incompatibilité qui existe actuellement entre les langages de programmation.

ANNEXES

ANNEXE I : L'interpréteur LISP pur en LISP

```
(APPLY (LAMBDA (FN X A)
       (COND
          ((ATOM FN)
             (COND
                ((EQ FN (QUOTE CAR)) (CAAR X))
                ((EQ FN (QUOTE CDR)) (CDAR X))
                ((EQ FN (QUOTE CONS)) (CONS (CAR X) (CADR X)))
                ((EQ FN (QUOTE ATOM)) (ATOM (CAR X)))
                ((EQ FN (QUOTE EQ)) (EQ (CAR X) (CADR X)))
                (T (APPLY (EVAL FN A) X A)) ))
          ((EQ (CAR FN) (QUOTE LAMBDA))
             (EVAL (CADDR FN) (PAIRLIS (CADR FN) X A)) )
          ((EQ (CAR FN) (QUOTE LABEL))
             (APPLY (CADDR FN) X (CONS (CONS (CADR FN) (CADDR FN)) A)) ))
))

(EVAL (LAMBDA (E A)
       (COND
          ((ATOM E) (CDR (ASSOC E A)))
          ((ATOM (CAR E))
             (COND
                ((EQ (CAR E) (QUOTE QUOTE)) (CADR E))
                ((EQ (CAR E) (QUOTE COND)) (EVCON (CDR E) A))
                (T (APPLY (CAR E) (EVLIS (CDR E) A) A)) ))
          (T (APPLY (CAR E) (EVLIS (CDR E) A) A)) )))

(EVLIS (LAMBDA (M A)
       (COND
          ((NULL M) NIL)
          (T (CONS (EVAL (CAR M) A) (EVLIS (CDR M) A))) )))

(EVCON (LAMBDA (C A)
       (COND
          ((NULL C) NIL)
          ((EVAL (CAAR C) A) (EVAL (CADAR C) A))
          (T (EVCON (CDR C) A)) )))
```

ANNEXE II : Le compilateur du § 2.4 en LISP

```
(CAPPLY
   (LAMBDA (FN)
      (COND
         ((ATOM FN)
            (COND
               ((EQ FN (QUOTE CAR)) (ATTACH (QUOTE (CAR))))
               ((EQ FN (QUOTE CDR)) (ATTACH (QUOTE (CDR))))
               ((EQ FN (QUOTE CONS)) (ATTACH (QUOTE (CONS))))
               ((EQ FN (QUOTE ATOM)) (ATTACH (QUOTE (ATOM))))
               ((EQ FN (QUOTE EQ)) (ATTACH (QUOTE (EQ))))
               (T (ATTACH (LIST (QUOTE LINK) FN))) ))
         ((EQ (CAR FN) (QUOTE LAMBDA))
            (PROG NIL
                  (COND ((CADR FN) (ATTACH (CONS (QUOTE ARGS) (CADR FN)
))) )
                  (CEVAL (CADDR FN))
                  (COND
                     ((CADR FN)
                        (ATTACH (LIST (QUOTE FREE) (LENGTH (CADR FN))))
 )
                     (T NIL) )))
         ((EQ (CAR FN) (QUOTE LABEL))
            (PROG NIL (ATTACH (CADR FN)) (CAPPLY (CADDR FN))) ))))
(CEVAL
   (LAMBDA (E)
      (COND
         ((ATOM E) (ATTACH (LIST (QUOTE PUSHV) E)))
         ((ATOM (CAR E))
            (COND
               ((EQ (CAR E) (QUOTE QUOTE))
                  (ATTACH (LIST (QUOTE PUSHC) (CADR E))) )
               ((EQ (CAR E) (QUOTE COND))
                  ((LAMBDA (ET1) (PROG NIL (CEVCON (CDR E) ET1) (ATTACH
ET1)))
                     (GENSYM) ))
               (T (PROG NIL (CEVLIS (CDR E)) (CAPPLY (CAR E))) ))
         (T (PROG NIL (CEVLIS (CDR E)) (CAPPLY (CAR E))) )))
(CEVLIS
   (LAMBDA (M)
      (COND
         ((NULL M) NIL)
         (T (PROG NIL (CEVAL (CAR M)) (CEVLIS (CDR M))) ))))
```

```
(CEVCON
   (LAMBDA (C ETI)
      (COND
         ((NULL C) NIL)
         ((EQ (CAAR C) (QUOTE T)) (CEVAL (CADAR C)))
         (T ((LAMBDA (X)  (PROG NIL
                     (CEVAL (CAAR C))
                     (ATTACH (LIST (QUOTE BRNIL) X))
                     (CEVAL (CADAR C))
                     (ATTACH (LIST (QUOTE GO) ETI))
                     (ATTACH X)
                     (CEVCON (CDR C) ETI) ))
               (GENSYM) )))))



(ATTACH
   (LAMBDA (I) (SETQ LISTING (APPEND1 LISTING I))) )
(COMPILE
   (LAMBDA (N)  (PROG (LISTING)
          (SETQ LISTING (CONS N NIL))
          (CAPPLY (GET N (QUOTE EXPR)))
          (ATTACH (QUOTE (RETURN)))
          (LPRIN LISTING)
          (RETURN LISTING) )))
```

B I B L I O G R A P H I E

[1] ABRAHAMS Paul W., et al.
"The LISP2 programming language and system"
Proc. FJCC 1966, p. 661-676

[2] BOBROW Daniel G., RAPHAEL Bertram
"A Comparison of list-processing computer languages"
CACM vol. 7,4 (avril 1964), p. 231-240

[3] BOBROW Daniel G.
"A model and stack implementation of multiple environnements"
CACM vol. 16,10 (oct. 1973), p. 591-603

[4] BOBROW Daniel G., RAPHAEL Bertram
"New programming Languages for Artificial Intelligence"
Computing Surveys, vol. 6,3 (sept. 1974), p. 155-174

[5] BERKELEY Edmund C., BOBROW Daniel G. (éditeurs)
"The programming language LISP : its operation and applications"
Information International Inc., Cambridge, Mass., mars 1964

[6] CHENEY C.J.
"A nonrecursive list compacting algorithm"
CACM vol. 13,11 (nov. 1970), p. 677-678

[7] CHURCH Alonzo
"The calculi of lambda-conversion"
Princeton University Press, Princeton, N.J., 1941

[8] COHEN Jacques
"A use of fast and slow memories in list-processing languages"
CACM vol. 10,2 (février 1967), p.82-86

[9] COLLINS G.E.

"The SAC-1 integer arithmetic system-version III"

Technical Report 156
Computer Science Dept., University of Wisconsin, mars 1973.

[10] DARLINGTON J., BURSTALL R.M.

"A system which automatically improves programs"

3rd IJCAI, Stanford, août 1973, p. 479-485

[11] DEUTSCH Peter L.

"A LISP machine with very compact programs"

Proc. 3rd IJCAI, Stanford 1973, p. 697-703

[12] FENICHEL Robert R., YOCHELSON Jerome C.

"A LISP garbage collector for virtual-memory computer system"

CACM vol. 12, 11 (nov. 1969), p. 611-612

[13] FENICHEL Robert R.

"Comment on Cheney's list-compaction algorithm"

CACM vol. 14,9 (sept. 1971), p. 603-604

[14] GOLDSTEIN Ira

"PRETTY-PRINTING - Converting List to Linear Structure

MIT, AI-Laboratory, Memo 279, février 1973

[15] GREEN C. Cordell, et al.

"Progress report on program-understanding systems"

Computer Science Dept, Report Stan-CS-74-444, Stanford, août 1974

[16] GREENBLATT Richard

"The LISP Machine"

MIT Artificial Intelligence Lab., Working Paper 79 (draft), novembre 1974.

[17] HANSEN Wilfred J.

"Compact list representation : definition, garbage collection, and system implementation"

CACM vol. 12, 9 (sept. 1969), p. 499-507

[18] HEARN Anthony C.

"Standard LISP"

Stanford Artificial Intelligence Project,
Memo AI-90, Stanford, mai 1969

[19] HEARN Anthony C.

"REDUCE-2 User's manual"

University of Utah, Salt Lake City, mars 1973

[20] HEWITT Carl

"More Comparative Schematology"

MIT, Project MAC, AI-memo n° 207, août 1970

[21] JENKS Richard D.

"META/PLUS The Syntax extension facility for SCRATCHPAI

IBM T.J. Watson Res. Center
Yorktown Heights, février 1971

[22] JENKS Richard D.

"META/LISP An interactive translator writing system"

IBM T.J. Watson Res. Center,
Yorktown Heights, juillet 1970

[23] KNIGHT Tom

"CONS"

MIT Artificial Intelligence Laboratory,
Working Paper 80 (draft), novembre 1974

[24] LISP/360 Reference Manual

4° ed., mars 1972
Stanford Computer Center, Stanford University
Document number SCC024

[25] LONDON Ralph L.

"Correctress of a compiler for a LISP subset"

SIGPLAN notices, janvier 1972, p. 121-127

[26] LÜGGER J., MELENK H.

"Darstellung und Bearbeitung umfangreicher LISP-
programme"

Angewandte Informatik 6/73, (juin 1973), p. 257-26

[27] Manuel LISP pour UNIVAC 1108

Politechnico di Milano, mai 1973

Le système LISP pour UNIVAC 1108 a été développé à l'Université de Wisconsin.

[28] McCARTHY John

"Recursive functions of symbolic expressions and their computation by machine"

CACM vol.3 (avril 1960), P. 184-195

[29] McCARTHY John

"A basis for a mathematical theory of computation"

dans : Computer programming and formal systems
ed. P. Braffort et D. Hirschberg
North Holland Publishing Comp., Amsterdam
1963.

[30] McCARTHY John et al.

"LISP1.5 Programmer's Manual"

MIT Press, Cambridge, Mass. 1965

[31] MINSKY Marvin L.

"A LISP garbage collector algorithm using serial secondary storage"

Project MAC, MIT, Memo MAC-M-129,
Cambridge, Mass., décembre 1963

[32] MOON David A.

"MACLISP Reference Manual"

Project MAC-MIT, Cambridge, Mass., avril 1974

[33] MOORE J. Strother

"Computational Logic : Structure sharing and proof of program properties"

DCL Memo n° 67,68 - Université d'Edimbourgh,
Department of Computational Logic.

[34] MOSES Joël

"The function of function in LISP"

SIGSAM Bulletin, juillet 1970, p. 13-27

[35] NEWELL Allen et al.
"The kernel approach to building software systems"
Computer Science Research Review 1970-1971
CMU, p. 39-52

[36] NORDSTROM Mats
"A parsing technique"
Utah Computational Physics Group, operating note 12.
University of Utah, novembre 1973

[37] Proceedings of EUROSAM 74
Royal Inst. of Technology Stockholm, août 1974
dans : SIGSAM Bulletin n° 31, vol. 8, 3

[38] QUAM L.H. DIFFIE W.
"Stanford LISP1.6 Manual"
AI Operating note 28-7, Stanford University 1972

[39] RISCH Tore
"REMREC - a program for recursion removal"
Université de Uppsala, novembre 1973

[40] RIVERO Victor
"INTERPOL"
Publication interne, Laboratoire d'informatique de L'université Scientifique et Médicale de Grenoble, février 1975

[41] SMITH D.C. "MLISP"
Artificial Intelligence Project Memo AIM-135
Stanford University, 1970

[42] SMITH D.C., ENEA H.J.
"MLISP2"
Artificial Intelligence Project Memo AIM-195,
Stanford University, 1973

[43] TEITELMAN W.
"Interlisp Reference Manual"
Xerox Palo Alto Research Center 1974

TABLE DES MATIERES

[44] TEITELMAN Warren

"CLISP - conversational LISP"

Proc. 3rd IJCAI, Stanford, août 1973, p. 686-690

[45] VUILLEMIN Jean

"Proof techniques for recursive programs"

IRIA, juin 1973

[46] WEGBREIT Ben

"The treatment of data types in EL1"

CACM vol. 17,5 (mai 1974), p. 251-264

[47] WEISSMAN Clark

"LISP1.5 primer"

Dickenson Publishing Company Inc., Belmont, Cal., 1967

[48] WEIZENBAUM Joseph

"Review R67-22 (of the LISP 2 programming language and system).

IEEE Trans. Elec. Comp. Vol. EC-16,2 Avril 1967, p. 236-238

critique de [1]