Institut National Polytechnique de Lorraine

Centre de Recherche en Informatique de Nancy

Etude et implantation d'un langage de traitement d'images Optimisation pour des architectures spécialisées

THESE DE 3ème CYCLE

Spécialité INFORMATIQUE

PRESENTEE PAR

Zizette BOUFRICHE

SOUTENUE LE 12 septembre 1987

DEVANT LA COMMISSION D'EXAMEN

PRESIDENT J.P Haton

EXAMINATEURS A. Belaid
J.P Finance
R. Mohr
B. Zavidovique

Institut National Polytechnique de Lorraine

Centre de Recherche en Informatique de Nancy

Etude et implantation d'un langage de traitement d'images Optimisation pour des architectures spécialisées

THESE DE 3ème CYCLE

Spécialité INFORMATIQUE

PRESENTEE PAR

Zizette BOUFRICHE

SOUTENUE LE 12 septembre 1987

DEVANT LA COMMISSION D'EXAMEN

PRESIDENT J.P Haton

EXAMINATEURS A. Belaid
J.P Finance
R. Mohr
B. Zavidovique

REMERCIEMENTS

Je tiens à remercier ici :

Monsieur Jean-Paul HATON pour l'honneur qu'il me fait en présidant ce jury. Je n'oublie pas qu'à mon arrivée, il m'a ouvert les portes de son équipe de *Reconnaissance des Formes et Intelligence Artificielle.*

Monsieur Abdelwahab BELAID qui avec SAPIN, a su créer l'environnement adéquat pour le langage LPSI. Ses critiques et ses conseils amicaux m'ont incitée à faire mieux.

Monsieur Jean-Pierre FINANCE qui, malgré ses nombreuses responsabilités, s'est intéressé à ce travail et a accepté de le juger.

Monsieur Roger MOHR pour avoir dirigé cette recherche et pour me compter parmi les membres de son équipe *VISION*. Je lui dois énormément pour ses compétences et son expérience. Sans ses multiples interventions et sa bienveillante exigence, ce travail ne serait pas ce qu'il est. Je saisis l'occasion pour lui dire toute ma gratitude.

Monsieur Bertrand ZAVIDOVIQUE pour l'intérêt qu'il porte à ce travail et pour sa participation à ce jury. La collaboration avec son équipe de recherche a été pour nous des plus bénéfiques.

Mes remerciements vont aussi à tous ceux qui m'ont aidé. Je voudrai citer :

K. TOMBRE pour avoir toujours répondu à mes innombrables sollicitations et pour son amitié.

A. BOYER, A. QUERE qui par leur lecture attentive et leurs judicieuses remarques ont contribué à l'amélioration de ce texte.

J. MZALI, Y. GONG, A. ROUSSANALY pour leur aide en matière de traitement de textes.

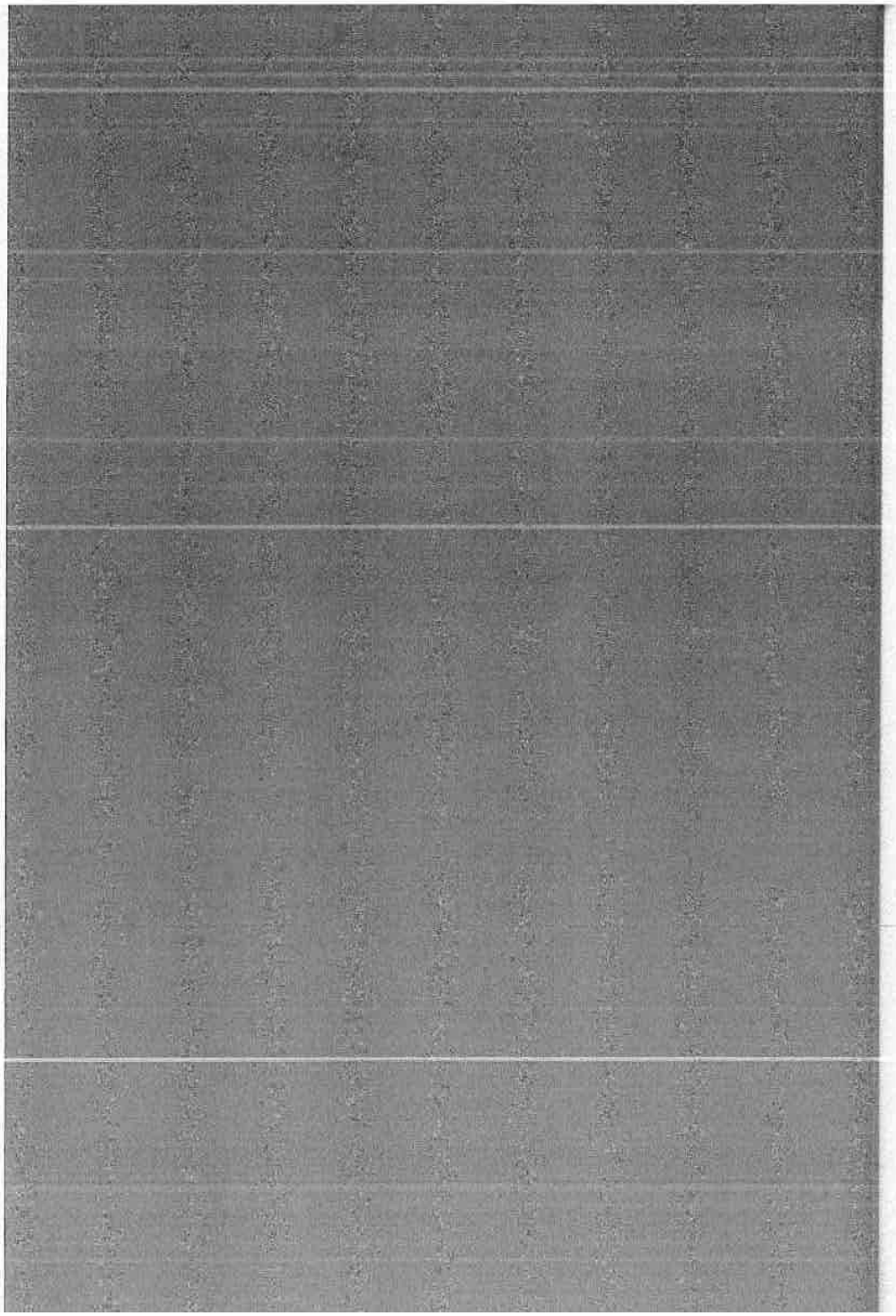
l'*ADI* et l'*ETCA* pour leur soutien financier qui a permis la réalisation du projet SAPIN.

Merci à tous mes amis du CRIN pour leur appui amical et pour leur présence rassurante, en particulier Josette et Ahlem.

Enfin que ma famille, qui a béni mon désir d'apprendre et qui m'a toujours encouragée, trouve ici le témoignage de ma profonde tendresse ainsi que *Lynda*, ma soeur spirituelle, à qui je dis toute mon affection.

Table des matières

INTRODUCTION



INTRODUCTION

L'image n'est pas seulement une distraction. Elle est aussi une source d'informations riches sur notre environnement. Son traitement par ordinateur connaît depuis quelques années, un développement sans cesse croissant lié à une véritable évolution matérielle et logicielle.

Sur le plan matériel, les progrès technologiques récents dans la conception des circuits intégrés (VLSI) et des architectures parallèles ont permis et permettent encore des gains importants de vitesse d'exécution.

Parallèlement, le logiciel s'est développé pour s'adapter à une évolution sans cesse croissante du matériel et pour répondre aux besoins des utilisateurs dans de nouveaux domaines d'application. Une forme d'aide à l'utilisateur a été la création de bibliothèques spécialisées utilisables à partir de langages existants. Mais le développement et l'usage des systèmes de traitement d'images ont été ralentis par l'inadéquation entre les langages classiques et les applications traitées dans ce domaine. On s'est alors penché sur la notion de langage pour le traitement d'images en mettant en évidence les objets image et les traitements possibles sur eux.

C'est dans ce sens que nous proposons **LPSI**, un langage de programmation structurée pour le traitement d'images. Notre objectif est de définir un langage de haut niveau capable d'exprimer de manière simple et efficace des traitements spécifiques. Il doit par ailleurs constituer un outil d'aide au développement de programmes spécialisés et un moyen de structuration des traitements.

Rappelons à ce propos que LPSI évolue dans un système de traitement d'images complet appelé **SAPIN** [ABD*82,Bel83,Bel87]. SAPIN est destiné à deux catégories d'utilisateurs. Il offre aux non spécialistes, un système interactif à menus (SAPIN NI) [Cas83], apportant à l'utilisateur une aide à tous les niveaux. Aux spécialistes, il propose un langage de haut niveau (LPSI) capable d'exprimer simplement le traitement d'images. L'essentiel de notre travail porte sur la conception et la réalisation de ce langage.

Dans une première étape, nous nous sommes intéressés à la définition des concepts image

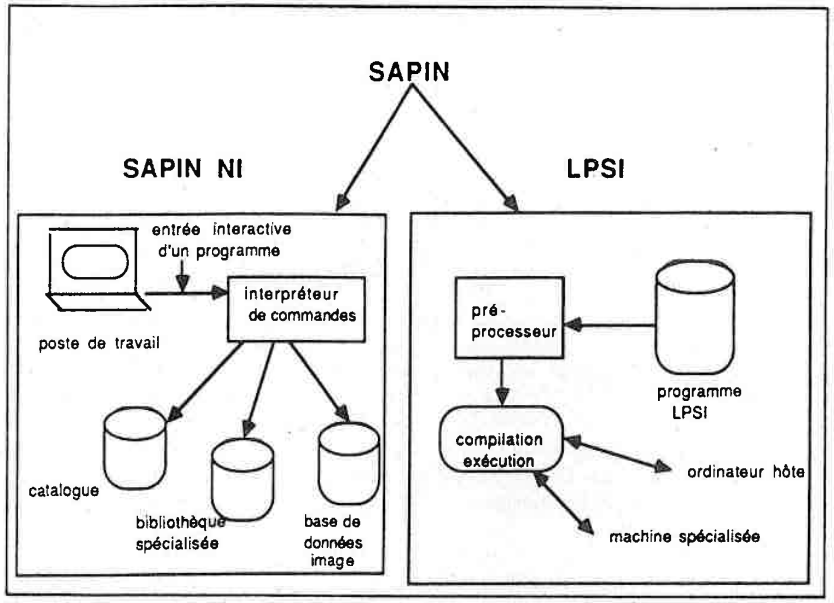


Figure 0.1: Organisation générale du système SAPIN

et aux formes d'expressions permises sur eux [Bou84]. Dans le dessein de mettre au point un outil général de programmation, nous avons opté pour un surlangage de **PASCAL**. Le choix de PASCAL a été essentiellement motivé par ses facilités de structuration des données et sa portabilité.

Dans la seconde étape, l'étude des modes d'implantation des différents objets de LPSI a permis d'aborder la réalisation de son pré-processeur.

Dans le travail que nous exposons, une présentation générale des différentes formes de langages et de machines spécialisés en traitement d'images est effectuée au chapitre 1. Notre propos n'est pas de faire une analyse détaillée des systèmes existants mais de situer notre travail dans son contexte.

Le chapitre 2 est consacré à la présentation de LPSI. C'est à ce niveau que nous décrivons les concepts du langage et les moyens de les combiner en termes de traitements d'images.

Dans le chapitre 3 nous présentons les différentes étapes de la construction du pré-processeur et discutons en particulier tous les problèmes d'implantation ainsi que les raisons qui ont guidé nos choix de solutions.

Les types de traitement à optimiser en fonction des possibilités de la machine cible et les schémas de traduction adéquats sont définis au chapitre 4. Nous nous sommes intéressés à deux types de machines : une machine multiprocesseurs (ICOTECH) [DZW86] et une machine pyramidale (SPHINX) [Mer83], dans lesquelles la stratégie de parallélisation est différente.

Dans la conclusion, nous dressons un bilan de ce qui a été réalisé et faisons part de nos suggestions pour l'amélioration de ce travail.

Enfin la partie annexe est consacrée à la présentation de quelques exemples de programmes LPSI commentés avec le code généré correspondant suivis de la syntaxe complète du langage.

CHAPITRE 1

Chapitre 1

LANGAGES ET MACHINES SPECIALISES EN TRAITEMENT D'IMAGES

1 INTRODUCTION

Le traitement d'images suscite un intérêt croissant. Cet intérêt est principalement motivé par la grande quantité de données qu'il faut manipuler et les temps d'exécution prohibitifs que cela entraîne sur une machine séquentielle. La littérature abonde dans ce sens : une image occupe 256 koctets. Ce chiffre est à multiplier par 3 pour une image couleur dont chacune des trois composantes représente une couleur fondamentale. Dans le cas de l'imagerie spatiale, une image comme celle fournie par le satellite LANDSAT peut être représentée par 2340 × 3240 pixels dans quatre bandes spectrales, ce qui donne au total 30 Mégaoctets, représentant une zone de 185 × 185 km [Cas85].

Si l'oeil et le cerveau humains analysent ces images avec une rapidité étonnante grâce au parallélisme du cerveau, ce n'est sûrement pas le cas des ordinateurs. Pour donner un ordre de grandeur : la recherche par un simple procédé de corrélation d'un motif de taille 100×100, dans une image 1024×1024, nécessite environ $10^4 \times 10^3 \times 10^3$ soit 10^{10} opérations !

Par ailleurs, si la plupart des programmes se sont concentrés sur les aspects les plus simples à traiter, les problèmes abordés maintenant sont d'une complexité bien supérieure et nécessitent le développement de programmes complexes comme par exemple l'identification et le chaînage de contours complexes et faiblement perceptibles.

Pour pallier ces problèmes de temps de calcul prohibitifs et de complexité de programmation, un certain nombre de logiciels et d'architectures spécialisés ont vu le jour. Plusieurs ouvrages font état de travaux menés dans ce domaine. Citons, parmi tant d'autres [HM78, PR80,DL81,Mer83,Cas85,Dan86,CBL86].

En ce qui concerne les logiciels, l'accent est mis sur leur souplesse d'utilisation et leur portabilité. Un grand nombre d'algorithmes classiques de traitement d'images sont regroupés autour de bibliothèques spécialisées constituant des ensembles logiciels portables [SB84]. A une plus grande échelle, des systèmes complets sont élaborés incluant un langage de haut niveau facilitant la pratique du traitement d'images.

Pour optimiser les temps de calcul, les recherches ont été orientées vers une voie permettant d'augmenter la vitesse d'exécution. Le principe est d'exécuter simultanément plusieurs opérations ce qui explique que de nombreuses architectures parallèles dédiées au traitement d'images ont été élaborées. De nouvelles notions se sont développées dans ce contexte (processeurs spécialisés, tableau de processeurs élémentaires, etc..) conduisant à des classes d'architectures parallèles différentes [Mon85].

Un autre aspect du parallélisme concerne les moyens de l'exprimer et de l'exploiter. Ceci reste la tâche essentielle du logiciel car quelle que soit sa puissance, une architecture n'est vraiment performante que si le logiciel qui la régit peut y accéder facilement et facilite à l'utilisateur la mise en oeuvre des programmes [Woo81].

2 LES LANGAGES SPECIALISES EN TRAITEMENT D'IMAGES

Afin de mieux comprendre la nécessité d'un langage spécialisé, essayons de cerner le contexte général d'évolution des langages dans le temps.

2.1 Généralités sur les langages

Les langages constituent un moyen de communication entre l'homme et la machine. Du langage machine au langage évolué en passant par l'assembleur, la différence est considérable. L'évolution des langages suit une logique qui tend à les rapprocher de l'utilisateur et à les rendre aussi indépendants que possible des machines qui les utilisent. Ils sont devenus plus normalisés et mieux structurés.

Les premiers balbutiements de la communication homme-machine étaient constitués d'échanges bruts sous forme de 0 et de 1. Réservé à des techniciens spécialisés, ce travail fastidieux accroît rapidement les possibilités d'erreurs et devient très vite impossible.
Avec le remplacement du code binaire par un ensemble de mnémoniques plus parlants, la seconde génération de langages vit l'apparition de l'assembleur et de ses dérivés. Moins contraignant que le langage machine, ce type de langages permet l'exécution rapide des instructions et est encore couramment utilisé. Il est par contre, tout autant réservé aux spécialistes et reste très spécifique à un ordinateur.

Une nouvelle génération de langages de programmation dénommés langages évolués est alors apparue. Leur objectif est d'être près de l'utilisateur et aussi indépendants que possible des machines qui les utilisent.

Si ces langages sont plus puissants et offrent une plus grande souplesse dans la programmation, ils présentent néanmoins quelques inconvénients. Malgré les progrès réalisés, aucun de ces langages n'est vraiment universel : FORTRAN a été conçu pour programmer facilement des calculs scientifiques. PASCAL intègre de par sa conception les règles de la programmation structurée. Même pour le langage ADA qui fut conçu indépendamment de toute considération technique, il n'existe pas de compilateur actuel capable d'exploiter toutes les possibilités du langage [Zub84].

D'autre part, ces langages ne permettent pas toujours de résoudre tous les problèmes dans les meilleures conditions (temps de mise au point, temps d'exécution), d'où l'apparition d'un grand nombre de langages spécialisés dans des applications très spécifiques telles que la simulation, l'analyse statistique, le traitement d'images et l'intelligence artificielle. Ces derniers résultent de la combinaison de trois facteurs [Zub84] :

- l'adaptation à la machine (ou au type de machine) sur laquelle le langage devra tourner,
- l'adaptation au type d'applications qu'il sera censé réaliser,
- la puissance et la rapidité auxquelles il devra se conformer.

2.2 Pourquoi un langage spécialisé en traitement d'images

Un langage spécialisé est censé résoudre dans les meilleures conditions les problèmes relatifs à une application donnée. Ceci est encore plus vrai en traitement d'images où les contraintes de taille et de diversification des données rendent plus délicats les problèmes à résoudre.

Devant l'apparition des machines spécialisées en traitement d'images, la nécessité d'un langage capable d'exprimer des traitements spécifiques sur les nouveaux processeurs a été ressentie. A cause de la nature spécialisée du traitement parallèle, on a préféré un nouveau langage pour la programmation des machines **SIMD** et **MIMD** (cf section 3.2).

Le développement d'un langage spécialisé en traitement d'images est également motivé par la spécificité des objets manipulés. L'introduction de types spécifiques contribue à une meilleure lisibilité des programmes ; la définition d'opérations propres facilite leur utilisation : le programmeur pourra alors manipuler simplement les données sans se soucier de leur gestion, les variables de type image étant utilisées comme des variables courantes du langage.

En vérifiant ces critères, le langage spécialisé pourra alors permettre au spécialiste la programmation du traitement d'images dans sa plus grande étendue.

2.3 Présentation de quelques langages de traitement d'images

Levialdi dans [Lev82] estime à 50 le nombre de langages de traitement d'images opérationnels, réalisés pendant ces deux dernières décennies. Les différentes orientations suivant

lesquelles ils ont été développés ont donné naissance à des langages de niveau de complexité varié qu'on peut schématiquement classer en trois niveaux :

1. les bibliothèques de sous-programmes spécialisées, utilisées à partir d'un langage de haut niveau,
2. les langages de commandes, le plus souvent adaptés à une architecture spécialisée,
3. les langages de haut niveau construits autour de deux variantes. Ils sont hybrides (issus de l'enrichissement de langages universels) ou autonomes.

Dans tous les cas, le but est d'apporter une solution efficace aux problèmes posés par le traitement d'images et d'en améliorer la puissance.

Les bibliothèques spécialisées en traitement d'images

Pour de nombreux chercheurs, un langage de traitement d'images est un ensemble de subroutines réalisées dans un langage universel et appelées à partir d'une bibliothèque. Un grand nombre d'opérations et d'algorithmes de traitement d'images y sont décrits, déchargeant ainsi l'utilisateur de la programmation de certains algorithmes classiques.

Dans la plupart des cas, ceci conduit à des ensembles logiciels facilement portables sur des systèmes incluant le langage hôte. Pour sa grande diffusion et ses facilités d'indexation des tableaux, FORTRAN est le plus souvent choisi. Il a été utilisé pour programmer *PAX* [Joh70], *SLIP* [PR80], *SPIDER* [TTS*82], *CHAP* [FG82]. Plus récemment (1986) est apparu *VISILOG*, un ensemble de sous-programmes de traitement d'images écrits dans le langage C.

Les fonctions disponibles dans ces bibliothèques couvrent à peu près l'ensemble des techniques de traitement d'images. PRESTON dans [PR80] les a regroupées en 7 classes résumées dans la table 1.1.

Cette approche est très utilisée dans le domaine. Elle reste jusqu'à présent le meilleur moyen pour assurer la portabilité du logiciel sur les machines universelles. Pour l'illustrer, nous donnons un exemple d'utilisation de la bibliothèque CHAP à partir d'un programme FORTRAN. Les noms de fonctions appartenant à la bibliothèque sont soulignés.

exemple

```
      integer chain1 (204)
      real area, chln, lnkscl, mperin, lperin
      data chain1/100,203*0/
C     Enter resolution of a region
      write(1,100)
  100 format('enter map scale ','(miles per inch):' )
      read(1,*)mperin
      write(1,101)
  101 format('enter digitization resolution','(links per inch):')
      read(1,*)lperin
      lnksl = mperin/lperin
C     Input chain data of a region (INPUT)
      call INPUT(5,chain1)
C     Compute perimeter of a region (LENGTH)
      call LENGTH(chain1,chln,1)
      write (6,102)chln
  102 format ('perimeter of the region is:',F10.4,'miles.')
C     Compute area of a region (ECREA)
      call ECREA(chain1,area)
      area = area *(lnksl**2)
      write(6,103)area
  103 format('area of the region is:',F10.4,'miles.')
```

Utilities Identifiers Executives Formaters I/O commands Test pattern generators Help files	Image Transforms Noise removal Fourier analysis and other spectral transforms Power spectrum Filtering Cellular logic	Geometric Manipulation Scaling/rotation Rectification Mosaicing/registration Map projection Gridding/masking
Image Display CRT Hard copy Interactive graphics	Image Measurement Histogramming Statistical Principal components	
Arithmetic Operators Point Line (Vector) Matrix Complex number Boolean	Decision Theoretic Feature select (training) Classify (unsupervised) Classify (supervised) Evaluate results	

table 1.1: Les fonctions sur les images recensées par PRESTON

Les langages de commandes

L'utilisation des bibliothèques spécialisées nécessite la connaissance d'un langage de programmation. Or dans certains cas, destinées à des domaines comme la télédétection et la médecine où le spécialiste n'est pas nécessairement informaticien, elles s'avèrent insuffisantes.

Afin de rendre leur usage plus général, on a pensé à leur associer des langages simples pour les manipuler rendant ainsi possible leur utilisation de manière transparente. Ces langages sont constitués d'une liste de commandes et sont, pour des raisons de rapidité, interprétés. L'accent est mis sur leur interactivité afin d'assurer une communication aisée entre l'utilisateur et le système de traitement d'images.

Le premier de ces langages est *GLOL* (GLopr Operating Language) réalisé par PRESTON [PR80,Pre81] en 1968. Ce langage est composé d'une série de commandes interprétées ligne par ligne qui opèrent sur des images de formats variables (32 x 32, 64 x 64 ou 128 x 128), manipulant jusqu'à trois images simultanées.

Codé en assembleur, GLOL est adapté à la programmation du processeur parallèle GLOPR (Golay LOgic PRocesseur) spécialisé dans le traitement rapide des transformations logiques. GLOL est reconnu comme un langage symbolique autorisant, en plus des commandes usuelles, des opérateurs standards sur des tableaux numériques à la manière d'un langage de haut niveau.

Longtemps commercialisé, GLOL existe dans de nombreux laboratoires d'hématologie. Il est utilisé pour l'analyse automatique d'images de globules blancs à la vitesse approximative de 4000 images à l'heure. Actuellement, il est exclusivement utilisé comme langage de traitement et d'exploitation d'images par Perkin-Elmer Corporation.

Comme exemple de programme GLOL, nous proposons la liste des commandes qui calculent le "ou excusif" de deux images. Dans le but de comparer GLOL avec les autres langages, cet exemple sera repris dans la suite. Tous ces programmes ont été soumis à un colloque sur les langages de traitement d'images, tenu a Windsor (Grande Bretagne) en 1979.

exemple

SET SYS 128	fixe à 128 la taille des images
DEF A,B,C	définit 3 images A, B, C
SEARCH A,N1	range l'image de nom N1 dans A
SEARCH B,N2	range l'image de nom N2 dans B
C = A * B	calcule le ou excusif de A, B et le range dans C

SUPRPIC est le second langage de commandes développé par PRESTON en 1978 pour des ordinateurs 32 bits de moyenne puissance. Comparé à GLOL, SUPRPIC est un langage hautement interactif, complètement paramétrique n'autorisant aucun symbole particulier.

Sa bibliothèque est une collection de vingt subroutines codées dans un mélange d'assembleur et de FORTRAN (cf table 1.2). Au cours d'une session interactive, elles sont présentées à l'utilisateur sous forme de menu où chaque commande est constituée d'un mnémonique suivi de la liste de paramètres requis.


```
COMPRESS (01,FILE,MODE,TH1,TH2,TH3,TH4)
MSKHIST (02,FILE, LU,BUF...)
ERASE (03,LU,BUF)
COPY (04,LU1,BUF1,LU2,BUF2)
INVCOPY (05,LU1,BUF1,LU2,BUF2)
AND (06,LU1,BUF1,LU2,BUF2,LU3,BUF3)
OR (07,LU1,BUF1,LU2,BUF2,LU3,BUF3)
EXOR (08,LU1,BUF1,LU2,BUF2,LU3,BUF3)
AUGRED (09,LU,BUF,CYC,FAC,CNUM,MODE,BORD)
MRGPLOT (10,LU)
MIXPLOT (11,LU)
GRAPLOT (12,LU)
EXPAND (13,LU,BUF,FILE)
DISPLAY (14,MODE,LU,BUF)
PICK (15,MODE,LU1,BUF1,LU2,TIME,COLOR)
STATUS (16)
ANALYZE (17)
COUNT (18,MODE,LU,BUF)
HISTORY (19)
QUIT (20)
```

table 1.2: Liste des commandes SUPRPIC

Afin de faciliter l'étape de saisie, chaque symbole du langage est codé. Ceci, malgré la concision obtenue, nuit à la lisibilité des programmes comme nous pouvons le constater pour la commande *0409010902* qui égale (04) deux images A (01) et B (02) associées à l'unité logique U1 (09). Cette mise au point nous paraît essentielle pour comprendre l'exemple donné à la fin de ce paragraphe.

exemple

RUN SUPRPIC	charge le système
0809010902	exécute le ou exclusif
20	arrêt

MORPHAL [Pre81,SB84] est un autre langage de commandes développé au centre de Morphologie Mathématique de Fontainebleau. Ses opérations, orientées vers l'analyse quantitative d'images, sont basées sur les concepts de morphologie mathématique [Ser82]. Elles sont de deux types :

1. les commandes d'analyse d'images dont la syntaxe est figée,
2. les structures de contrôle élémentaires comportant des ordres de branchement et d'entrée-sortie.

MORPHAL est adapté à une machine fortement parallèle, la machine AT4 en l'occurrence, autorisant le traitement simultané de plusieurs images. Le système est initialement utilisé pour des transformations logiques d'images sur des voisinages hexagonaux. A titre

d'exemple, l'union de plusieurs images et l'intersection du résultat avec l'union de plusieurs autres images nécessitent en moyenne 20 millisecondes. Des opérations plus compliquées telle une squelettisation demandent cependant 120 millisecondes.

La figure 1.1 schématise une session interactive d'un programme MORPHAL dans laquelle est sélectionnée une opération d'érosion. Le code MORPHAL pour "le ou exclusif" est donné ici :

exemple

```
SFLD(MA,MB)(128,128)(1.1)              dimensionne A et B
NOL((MA,MB,JN)-(MA,MB,IN)(MC))         exécute le ou exclusif
```

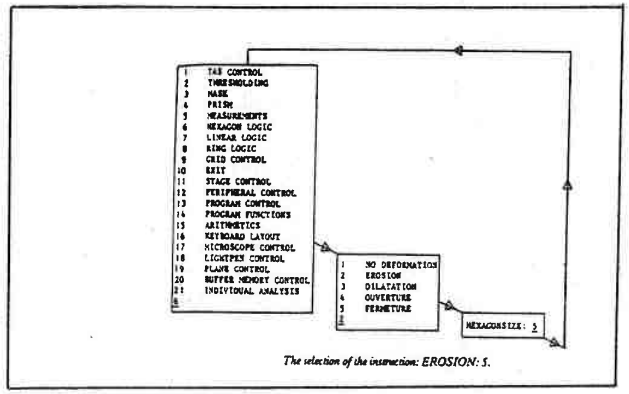


Figure 1.1: Session interactive d'un programme MORPHAL

SUSIE est un langage de commandes interactif codé en assembleur. Ces instructions sont de deux types : immédiates si elles sont directement exécutées ou, dans le cas contraire, différées.

Le premier type regroupe des commandes simples telle que la commande NE qui calcule le négatif d'une image, ou la commande TH32 qui effectue une opération de seuillage ou encore la commande 6xLP qui exécute 6 fois une moyenne locale sur des ensembles d'éléments fixés.

Les commandes différées sont des macro-instructions définies par l'intermédiaire de la primitive DM et rendant possible l'enchaînement séquentiel de commandes simples.
DM XY (3xLP;GR;TH32) crée par exemple une commande dénommée XY qui calcule successivement une moyenne, un gradient et un seuillage.

Toutes les commandes de SUSIE opèrent sur des images de taille fixe (128x128). Le système alloue un espace de 16 k-mots de 12 bits pour la manipulation simultanée de deux images. Une commande spéciale permet de sélectionner, quand cela est nécessaire, l'image à traiter.

PICASSO-SHOW est un ensemble de langages interactifs, de moyen niveau dont la première version opérationnelle fut réalisée en 1976. Le langage est en lui-même simple, pouvant servir d'outil pour le développement de programmes spécialisés de moyenne complexité.

Une bibliothèque modulaire codée en assembleur constitue son noyau de base, complété par des structures de données image et des opérations de contrôle. Toutes les instructions ont une syntaxe uniforme proche de celle de l'assembleur.

PICASSO-SHOW autorise trois modes d'exécution :

1. un mode interactif pour l'exécution immédiate de commandes élémentaires,
2. un mode interprétatif dans lequel sont exécutés des programmes plus complexes,
3. un mode programme ou interprétatif rapide dans lequel les programmes sont d'abord traduits dans un code intermédiaire puis exécutés. Dans ce mode, le "ou exclusif" de deux images s'écrit :

   exemple

```
S0, S1, S2 : 1,1,128,128,1,1    définit 3 images
SLOADN, S0, 1                   charge S0
SLOADN, S1, 1                   charge S1
ALTERNATE :                     exécute le "ou exclusif"
SADDM, S0, S1, S2
BOR, S0, S1, S2
BAND, S0, S1, S0
BDIFF, S2, S0, S2
```

La présentation de ces quelques langages est suffisante pour souligner leur simplicité mais aussi leur grande diversité. Certains langages sont extrêmement puissants et ressemblent à APL (GLOL en est un exemple). Ils intègrent des structures de contrôle qui rendent possible l'écriture de programmes plus ou moins structurés. D'autres, pour des raisons d'efficacité sont restés très près de l'assembleur dans leur structure. Quelques uns ont puisé dans FORTRAN comme c'est le cas pour MORPHAL alors que d'autres ont suivi des orientations nouvelles. Beaucoup ont favorisé l'aspect interactif les rendant ainsi accessibles à toutes les catégories d'utilisateurs.

Dans une extension plus large, beaucoup de ces langages sont fortement adaptés à une architecture spécialisée dans le but de bénéficier de ses grandes capacités de calcul. Si ce dernier point a l'avantage de réduire les temps de traitement, il crée malheureusement un problème de portabilité qui rend difficile tout échange de logiciels. La multiplicité des mnémoniques utilisés est également un obstacle à l'échange des logiciels et rend compte d'un manque total de standardisation, qualité reconnue essentielle pour un logiciel. Il devient alors urgent de remédier à ces problèmes afin que les efforts déjà réalisés ne soient pas perdus.

Les langages de haut niveau

La réalisation de langages de haut niveau en traitement d'images répond aux préoccupations des professionnels désireux de disposer d'outils spécialisés pour exprimer plus efficacement les traitements. Levialdi dans [Lev82], définit pour ces langages les deux qualités suivantes :

1. la facilité d'exprimer des traitements spécifiques,
2. la facilité de communiquer.

Le point (1) découle de l'inadaptation des langages universels au domaine du traitement d'images et de leur insuffisance pour programmer efficacement des traitements spécifiques. Le point (2) est primordial si l'on pense comme Preston [PR80] qu'un langage de programmation a trois buts essentiels : permettre à l'utilisateur de communiquer avec la machine, permettre à l'utilisateur de communiquer avec lui-même et permettre à l'utilisateur de communiquer avec les autres.

Dans ce contexte, deux voies de recherche différentes se sont dégagées. L'idée dans la première, est d'enrichir par des concepts image de haut niveau et par un ensemble de structures de contrôle adaptées à leur manipulation, un langage de programmation déjà existant. Ce choix de conception permet, d'une part, de bénéficier de l'expérience acquise pour les langages de programmation généraux, et d'autre part, d'expérimenter sur des machines séquentielles, des structures de données image et des constructions parallèles.

Pour sa disponibilité sur la plupart des ordinateurs, sa structuration et sa puissance, PASCAL s'est révélé être une bonne structure d'accueil pour les types et les structures de contrôle image. Il a servi de support à des langages expérimentaux comme Parallel PASCAL [Ree84], Parallel Extensions ou encore PASCAL PL [Uhr81].

La seconde direction est liée à l'expansion des ordinateurs spécialisés et au besoin de développer des langages parfaitement adaptés à leur architecture. Entièrement spécialisés, ces langages introduisent des structures de haut niveau facilitant la programmation des processeurs mis en place. Cette idée est supportée par Kruse et Douglass qui proposent, le premier un langage de haut niveau (PPL) pour sa machine PICAP, (la machine et le langage sont exclusivement désignés pour le traitement d'images) et le second, un langage orienté

machine (MAC) pour décrire et gérer des processus parallèles dans les algorithmes de traitement d'images

Dans les paragraphes qui suivent, nous résumons les principales idées supportées par quelques langages de traitement d'images de haut niveau dans les deux approches citées.

Le langage PAL

L'objectif principal des concepteurs de PAL est la définition d'un langage de haut niveau permettant :

- la représentation effective des structures de données fondamentales relatives aux images,
- la définition des traitements parallèles sur les images,
- une totale abstraction de la représentation par rapport à la machine cible.

Parmi les concepts image introduits dans PAL, l'objet **PICTURE** représentant une image, est le plus important. Il peut être de type binaire (BIN), entier (INT) ou réel (REAL) et peut être rangé :

- sous forme de plans binaires ; l'image occupe dans ce cas autant de plans que de bits par pixel,
- sous forme contigüe avec un plan unique par objet.

Les constructions du langage comprennent entre autres :

- une clause permettant de choisir un parmi plusieurs blocs de traitement sélectionnés par une valeur entière (case clause),
- des clauses itératives de deux sortes :
  1. l'itération classique (FOR statement),
  2. l'itération globale (FOR ALL) permettant d'exécuter une instruction donnée pour tous les points d'une image d'une manière parallèle, si c'est spécifié (PAR FOR ALL),
- des procédures de définition de nouveaux opérateurs rendant possible l'extension de la liste des opérations du langage.

Le langage L

Comme PAL, L est un langage de traitement d'images étendant ALGOL. Il définit six types spécifiques manipulés par un ensemble d'opérations et de fonctions. Dans L, les auteurs se sont particulièrement intéressés aux accès sélectifs dans les objets image et au moyen de leur associer un ensemble d'attributs.

Les types image reconnus par L sont :

- **IMAGE** pour désigner une image ; celle-ci est définie par ses dimensions et peut avoir une ou plusieurs bandes. Les bandes correspondent à différentes vues d'une même scène.

  exemple

  Image : A<15,15>, E[5]<20,20>;
  E est une image possédant 5 bandes de taille 20 x 20.

- **IMAGE GROUP** pour désigner un ensemble d'images, chaque image du groupe étant elle-même de type IMAGE.

  exemple

  IMAGE GROUP : A(3)<20,20>;

- **BOOLEAN IMAGE** pour définir des images binaires,

- **VIRTUAL IMAGE** et **VIRTUAL IMAGE GROUP** pour désigner des parties d'images réelles avec certaines contraintes de lecture et d'écriture.

  exemple

  IMAGE : A<100,100>;
  VIRTUAL IMAGE : [X in A<51:100, 51:100>];
  X est une image virtuelle de taille 50×50 définie à partir de A.

- **WINDOW** définit un outil de sélection de zones rectangulaires dans une image. La fenêtre est caractérisée par une taille et un point d'application modifiables.

L permet de définir, au début d'un programme, le nombre de bits devant être réservés par pixel ainsi que la taille qu'il faut attribuer par défaut aux images non dimensionnées.

exemple

DEFAULT IMAGE : <1024,520>;
BITS PER PIXEL : 8;

Une autre directive permet d'associer à un objet image une liste d'attributs telle que l'histogramme, la variance et la moyenne. Ces attributs accompagnent l'objet pendant toute sa durée de vie, peuvent être utilisés à tout moment mais ne sont mis à jour que si l'objet en question a été modifié ; ce qui a pour avantage d'éviter les calculs inutiles.

exemple

ASSOCIATE ATTRIBUTE :
A with HISTO,
B with (HISTO, VAR);
les attributs HISTO pour l'histogramme et VAR pour la variance sont prédéfinis.

Les modes d'accès dans les objets de L sont de type local et global. Ainsi :

- A<i,j> accède à un pixel de A,
- A(i) fait référence à la ième image du groupe,
- B[j] désigne la jème bande dans B,
- B:W accède dans B à un ensemble de points que sélectionne la fenêtre W initialisée par l'affectation :
  W := <h,l>
  et positionnée grâce à la fonction :
  POSITION(W,B,x,y).

Dans L, la plupart des opérations arithmétiques sont simplement exprimés par des opérateurs globaux prédéfinis. Des opérations plus complexes sont réalisées par des fonctions. Comme dans PAL, une itération globale (FOR ALL statement) permet de traiter simultanément tous les points d'une même image. Cette itération peut être convenablement prise en charge par des processeurs parallèles asynchrones.

exemple

FOR ALL <i,j> **in** B do
begin
POSITION(V,A,i*5,j*3);
B<i,j> := **MEAN**(A:V);
end;
POSITION positionne la fenêtre V sur l'image A, MEAN calcule la moyenne des valeurs de cette partie d'image.

Le langage PIXAL

PIXAL (extension d'Algol 60) est un langage de traitement d'images alliant des possibilités de programmation séquentielle à des possibilités de programmation parallèle. Il doit être en mesure de programmer une machine de type SIMD (la machine CLIP par exemple). Dans sa version expérimentale, le parallélisme est simulé.

Les constructions image définies dans PIXAL sont les suivantes :

- les types **BINARY** et **GREY** renseignent sur le type (binaire ou niveau de gris) de l'information élémentaire dans les structures bidimensionnelles. Il n'existe pas de type IMAGE proprement dit,

  exemple

  BINARY ARRAY A[1:256,1:256];

- le type **MASK** permet de définir des masques de convolution,
  exemple
  BINARY MASK M[1:3,1:3];
- le type **FRAME** localise un voisinage autour de son point d'application. Il rappelle le type window du langage L,
- la directive **EDGE** offre la possibilité d'initialiser globalement une image à un certain niveau de gris,
  exemple
  EDGE OF A IS 0;

Le parallélisme dans PIXAL s'exprime de deux manières dans lesquelles le traitement de tous les points s'effectuent simultanément.
Il s'agit, dans le premier cas, d'un parallélisme local exprimé par la structure "**PAR** instructions **PAREND**", rappelant d'un point de vue sémantique l'instruction FOR ALL définie dans L et PAL.
exemple

> **PAR IF** A<>0 **THEN** A:=1 **PAREND;**
> Dans cette instruction, la condition est évaluée localement. Le traitement est effectué sur tous les points qui la vérifient.

Il s'agit, dans le second cas, d'un parallélisme global implicite.
exemple

> **IF** A <>R 0 **THEN** A := 1;
> La condition est évaluée globalement : le traitement ne sera effectué que si tous les points de A vérifient la condition posée.

Le langage PASCALPL

Comme son nom l'indique, PascalPl est une extension de PASCAL. Il est développé sur une machine séquentielle à laquelle sont connectés des processeurs spécialisés. Ces derniers prennent en charge le traitement parallèle des tableaux de pixels alors que le reste du code est normalement exécuté sur la machine hôte.

Une notation syntaxique (les symboles || en l'occurrence) permet de localiser les traitements parallèles à effectuer. Celle-ci intervient à plusieurs niveaux : si une procédure est, par exemple, susceptible de présenter des traitements parallèles, sa déclaration doit apparaître avec le mot clé ||PROCEDURE. Les déclarations des tableaux qui suivent doivent comporter la directive ||DIM s'ils sont traités parallèlement et les instructions qui les utilisent doivent être désignées de la même façon :

> ||SET pour l'affectation,
> ||CASE et ||IF ||THEN ||ELSE pour les tests,

> ||READ et ||WRITE pour la lecture et l'écriture.

exemple

```
||SET A := B+C;
||IF A < 0 ||THEN A := 0;
```

Le langage PPL

PPL (Picture Programming Language) [Gud81] est né de la proposition de Kruse pour le développement d'un langage de haut niveau adapté à la machine PICAP. Le langage est développé sur un matériel spécifique supportant un environnement logiciel complet (éditeur de textes, traducteur, gestionnaire de fichiers, etc...).

Ce langage est particulièrement puissant dans les manipulations conditionnelles et les branchements. Certaines de ses instructions sont fortement dépendantes de la structure de la machine, en particulier il permet de définir des opérateurs pour l'exécution des transformations logiques et convolutions d'entiers supportées par la machine. Il permet aussi d'accéder aux registres spéciaux qui contiennent l'information rangée par le processeur PICAP.

Un programme PPL est constitué d'un ensemble de procédures pouvant être éditées et compilées séparément. Les principaux éléments du langage sont :

- les types
  les types autorisés par PPL sont les suivants :
  entier, réel, booléen, tableau, image binaire (1 bit/pixel), image (8 bits/pixel),
- les opérateurs de traitement d'images
  de nature arithmétique (MATR) ou logique (TEMP), ils opèrent parallèlement sur des voisinages 3×3 ; ces opérateurs ne sont pas figés.
  exemple

  MATR LAPLAC
  0 1 0
  1 -4 0 0
  0 1 0
  définit un opérateur de convolution de type MATR nommé LAPLAC.
- les structures de contrôle
  PPL comporte un jeu complet de structures de contrôles :
  DO instructions **OD;**
  WHILE expression **DO;**
  IF .. **THEN** ... **FI;**
  CASE.. **OF**...;

Conclusion

De cette présentation des langages de traitement d'images se dégagent deux grandes tendances :

- La première repose sur l'utilisation des langages classiques pour la réalisation de bibliothèques spécialisées. Elle constitue une aide appréciable pour le développement de programmes d'application variés présentant un caractère de portabilité optimal.
- La seconde tendance est celle de langages construits autour de structures particulières qui leur confèrent de grandes possibilités dans le domaine de l'image. Les langages présentés sont une preuve des efforts déjà fournis bien qu'on soit encore loin du langage de traitement d'images formellement défini, simple, indépendant de la machine, facilement extensible, parallèle et pourquoi pas interactif, qualités que Levialdi [Lev82] juge indispensables dans un langage de traitement d'images dit *universel*.

3 LES MACHINES SPECIALISEES EN TRAITEMENT D'IMAGES

De nombreux logiciels de traitement d'images sont actuellement opérationnels s'exécutant sur différents types de machines mais on leur reproche d'être encore trop lents.

Pour la plupart, la lenteur provient du très grand nombre d'opérations arithmétiques élémentaires et des tests à effectuer. Par ailleurs, la vitesse à laquelle sont restituées les informations est contraignante : les machines habituelles SISD (Single Instruction Single Data-stream) sont insuffisantes aux performances attendues.

Le recours à des machines spécialisées est la solution envisagée depuis quelques années. Ces machines fondées sur une architecture parallèle offrent un moyen efficace pour augmenter la rapidité des traitements.

3.1 Possibilités des machines séquentielles

Les machines séquentielles sont les machines classiques mono-instruction, mono-flot de données. Malgré les progrès technologiques qui ont permis l'augmentation de leur vitesse d'exécution (celle-ci est passée d'un millier d'instructions par seconde à quelques MIPS c'est à dire quelques millions d'opérations par seconde), ces machines restent très peu adaptées au traitement d'images.

De nombreux processeurs spécialisés ont alors vu le jour et leurs performances ont rendu possible le développement de machines spécialisées en traitement d'images, organisées autour d'ordinateurs conventionnels. Le parallélisme est obtenu à partir des processeurs eux-mêmes ou par des moyens efficaces de connexion. On parle alors d'array-processeurs, de réseaux systoliques ou de machines pyramidales. Un ordinateur hôte permet alors la gestion des ressources matérielles et la prise en charge de toutes les tâches qui ne concernent pas directement l'image.

3.2 Les machines parallèles

Aspects classiques du parallélisme

D'une manière générale, le parallélisme permet l'exécution simultanée de différentes tâches. Plusieurs classifications de machines parallèles ont été proposées. Celle de FLYNN est la plus simple et la plus connue bien que moins complète. Elle repose sur le parallélisme réalisé au niveau des flots d'instructions et des flots de données. Elle propose quatre classes de machines :

- **SISD** (Single Instruction stream Single Data stream) correspond à la machine séquentielle de Von Neumann.
- **SIMD** (Single Instruction stream Multiple Data stream) correspond aux machines où une même instruction est exécutée sur plusieurs données indépendantes.
- **MISD** (Multiple Instruction stream Single Data stream) désigne les machines où plusieurs instructions travaillent sur un même chemin de données. Cette technique est généralisée sous le nom de pipe-line lorsque les processeurs sont répartis sur le chemin de données. Les machines systoliques constituent une classe particulière dans laquelle le flux de données circule de façon rythmique.
- **MIMD** (Multiple Instruction stream Multiple Data stream) désigne les machines où des instructions différentes travaillent sur des chemins de données différents.

Parallélisme en traitement d'images

Castan dans [Cas85] classe les différents types de parallélisme permettant de traiter une image de la manière suivante :

- **parallélisme des images** correspondant au traitement ponctuel dans lequel le calcul de chaque point s'effectue indépendamment de ses voisins,
- **parallélisme des voisinages** qui correspond au traitement local utilisant l'information d'un pixel et celle des pixels appartenant à un certain voisinage,
- **parallélisme des opérateurs** dans lequel plusieurs opérations sont effectuées en parallèle sur différentes images,
- **parallélisme des bits par pixel** dans lequel chaque processeur est à même de manipuler plusieurs bits à la fois.

Machines utilisant le parallélisme des images

Le parallélisme des images est réalisé par des machines SIMD de la première classification. La plupart de ces machines reposent sur une organisation bidimensionnelle du réseau de processeurs dont la structure est semblable à celle de l'image. Ainsi un certain nombre de processeurs élémentaires disposés suivant un arrangement matriciel effectuent la même instruction d'une manière synchrone. Chaque processeur possède une mémoire locale et peut

communiquer avec ses voisins (4 ou 8 dans une maille carrée, 6 dans une maille hexagonale). Sur ce principe fonctionnent les machines CLIP et MPP.

La machine CLIP4

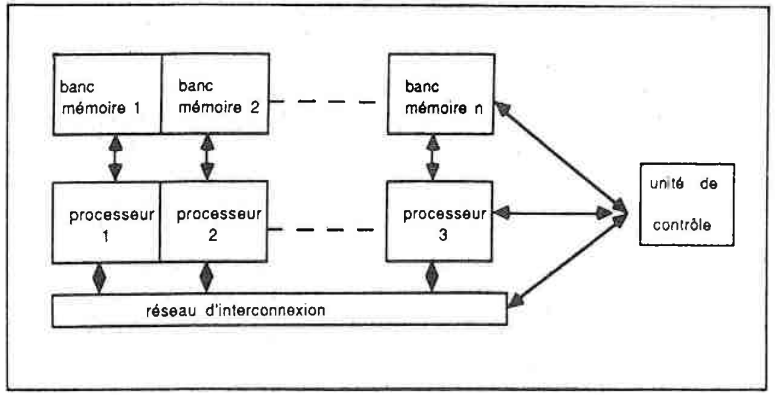


Figure 1.2: Organisation d'un tableau de processeurs

La machine CLIP4 (Cellular Logic Image Processeur) [Fou81,Cas85] fait partie d'une famille de machines parallèles développées par l'équipe de DUFF à Londres. Elle est essentiellement constituée d'une matrice de 96 x 96 processeurs booléens, chacun d'eux étant connecté à ses 8 plus proches voisins. CLIP4, initialement développée pour traiter des images binaires, est très performante pour les opérations logiques portant sur une fenêtre binaire 3×3. Il est également possible de configurer un processeur en additionneur pour traiter des images à plusieurs niveaux de gris. Une convolution 3×3 exige dans ce cas 20 ms. CLIP4 et CLIP5 sont les premières machines d'une grande série. CLIP6 acceptera en entrée des images tridimensionnelles [Eri82].

La machine MPP

La machine MPP (Massively Parallel Processor) a pour principale fonction le traitement des images de satellite. Elle est par conséquent plus orientée vers le calcul arithmétique que la machine CLIP. MPP est constituée d'une matrice de 128 x 128 processeurs. Chaque processeur possède des registres, de la mémoire locale (1 kbit), un additionneur, un opérateur logique, et est relié à ses quatre plus proches voisins.

MPP, grâce à une conception bien affinée, conduit à des résultats très intéressants. Ses performances sont 6553 MOPS (millions d'opérations par seconde) pour les additions sur des entiers de 8 bits et 77.625 millions de pixels par seconde pour les convolutions 3×3.

Machines utilisant le parallélisme des opérateurs

Le parallélisme des opérateurs est réalisé sur des machines animées par des processeurs dont chacun exécute ses propres instructions. Selon le mode de fonctionnement des processeurs, deux cas sont à distinguer :

- les processeurs sont synchrones. Chacun d'eux reçoit comme donnée d'entrée le résultat du processeur précédent et envoie son résultat comme donnée d'entrée du processeur suivant (technique du pipeline). Ceci nécessite le "tronçonnement" d'un processus image en tâches élémentaires. Une fois le traitement amorcé, toutes les tâches sont actives simultanément sur des données successives. La technique du pipeline fige d'une manière rigide un enchaînement d'opérations, ce qui explique l'orientation des machines pipeline vers une classe de traitement d'images.

La machine CYTOCOMPUTER

CYTOCOMPUTER est une machine caractéristique des processeurs pipeline. Elle est constituée d'une chaîne de 88 processeurs connectés en pipeline pour traiter des niveaux binaires et de 25 dans un deuxième pipeline qui traitent en parallèle des niveaux de gris. Ces processeurs effectuent des transformations sur un voisinage 3×3. Les transformations réalisées sont fondées sur la morphologie mathématique [Ser82].

- les processeurs sont asynchrones (cas des machines MIMD). Ils peuvent être banalisés ou spécialisés pour réaliser des fonctions particulières (opérations arithmétiques, convolution, etc...). Les machines MIMD sont les plus nombreuses.

La machine SYMPATI

La machine SYMPATI (Système Multi Processeur Adapté au Traitement d'Images) a été développée au laboratoire CERFIA de Toulouse. Sympati présente trois niveaux de parallélisme :

- un parallélisme des bits permettant de manipuler en parallèle 8 bits par pixel,
- un parallélisme SIMD effectuant des traitements au niveau pixel,
- un parallélisme MIMD réalisé par des processeurs standard connectés à la mémoire d'image, plus spécialisé dans le traitement des régions.

La mémoire est constituée de 16 blocs de 16 koctets tels que le bloc numéro i contienne les colonnes i (modulo 16) de l'image. Les blocs communiquent entre eux, ce qui permet un accès simultané aux voisins de 16 points.

La machine SATIR

La machine SATIR (Système d'Acquisition et de Traitement d'Images Robotiques) a été construite à l'université de Compiègne pour effectuer la partie traitement de bas niveau d'un système de vision pour robot. Satir utilise des images binaires obtenues après des traitements orientés "contour" d'images multiniveaux. Elle est constituée de processeurs MIMD câblés ou programmables et autorise un traitement pipeline des images reconfigurables à travers les différents modules.

La machine ICOTECH

Le système ICOTECH [ABD*82,DZW86] réalisé à L'Ecole Nationale Supérieure de Physique de Strasbourg est destiné au traitement numérique des images et à la conception assistée par ordinateur. Le système est modulaire, comprenant des unités de traitement interchangeables et reconfigurables.

La partie multiprocesseurs appelée PRIM est essentiellement constituée de modules connectés à un bus rapide. Elle comprend dans sa version actuelle cinq processeurs spécialisés, un processeur vidéo et une mémoire d'image.

L'ensemble du PRIM est utilisé à partir d'un ordinateur hôte. Le processeur vidéo est alors considéré comme un processeur spécialisé. Il peut aussi fonctionner seul et constituer dans ce cas un poste de travail comportant une mémoire d'image, des processeurs spécialisés fonctionnant en temps réel. L'ensemble est géré par un micro-ordinateur.

3.3 Les machines pyramidales

Les machines pyramidales sont des machines multiprocesseurs auxquelles le mode de connexion entre les différents processeurs confère une structure pyramidale. Les processeurs sont disposés suivant des étages dans lesquels chaque processeur communique avec ses voisins de l'étage (liens avec ses frères), ses voisins de l'étage en-dessous (liens avec ses fils) et ses voisins de l'étage au-dessus (liens avec son père).

Le type de connexion entre des étages consécutifs est généralement quaternaire : chaque processeur communique avec un père et quatre fils. Au sein d'un même étage, un fonctionnement SIMD des différents processeurs semble bien adapté aux problèmes de traitement

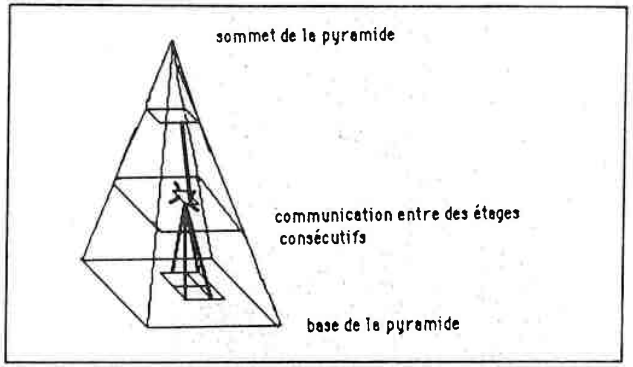


Figure 1.3: Organisation d'une machine pyramidale

d'images. La machine peut alors être vue comme un empilement de matrices de processeurs de type CLIP ou MPP.

Dans [Fou86], les machines pyramidales sont considérées comme une extension des tableaux de processeurs à deux dimensions. Elles sont utilisées pour des applications de vision de haut niveau telles que l'analyse de scènes. Au niveau de la base de la pyramide sont exécutés les algorithmes de traitement de bas niveau (extraction de contours par exemple). L'extraction des primitives se fait à travers les étages et au sommet de la pyramide sont recueillies les relations entre les différents objets de la scène.

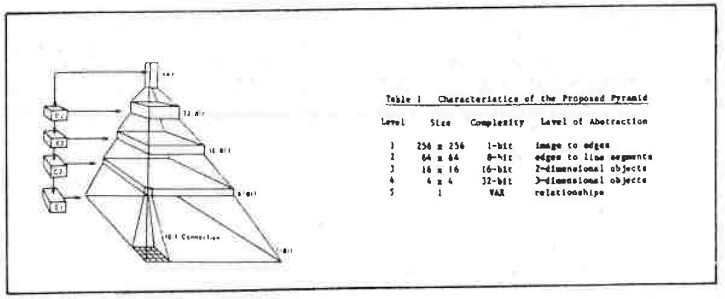


Table 1 Characteristics of the Proposed Pyramid

Level	Size	Complexity	Level of Abstraction
1	256 x 256	1-bit	image to edges
2	64 x 64	8-bit	edges to line segments
3	16 x 16	16-bit	2-dimensional objects
4	4 x 4	32-bit	3-dimensional objects
5	1	VAR	relationships

Un exemple de machines pyramidales est la machine SPHINX conçue et réalisée à l'I.E.F [Mer83]. Son architecture est basée sur une pyramide de processeurs avec un double mode d'interconnexion : une interconnexion en arbre binaire entre les différents étages et une interconnexion de type voisinage au sein d'un même étage. En plus du parallélisme (parallélisme SIMD entre les processeurs d'un même étage et MIMD entre étages), cette machine favorise l'exécution des algorithmes nécessitant des mouvements de données variés. Trois grands types de mouvements sont possibles :

- des mouvements ascendants permettant une centralisation de l'information,
- des mouvements descendants permettant une diffusion de l'information,
- des mouvements au sein d'un étage permettant une modification de l'information.

La pyramide est programmée par des commandes émises de l'extérieur par un ordinateur hôte.[1]

4 CONCLUSION

Dans cette étude, nous avons essayé de montrer les problèmes, fort bien connus d'ailleurs, rencontrés dans la programmation du traitement d'images et les solutions logicielles et matérielles auxquelles on a recours depuis quelques années.

Pour l'aspect logiciel, nous nous sommes surtout appliqués à dégager les différentes orientations prises pour la conception d'un logiciel spécialisé en traitement d'images. Nous nous sommes particulièrement intéressés aux techniques d'enrichissement de langages existants car elles nous paraissent le mieux correspondre à notre approche du problème. Toutefois, l'existence d'autres approches montre que ce domaine d'application est aussi un domaine de recherche en pleine évolution, dans lequel une grande contribution est encore possible.

D'autre part, la présentation de machines adaptées en traitement d'images, bien que très brève et loin d'être exhaustive, a permis de montrer l'étendue des progrès déjà réalisés. Caractérisées par une grande diversité dans leur architecture, ces machines s'adaptent bien aux différents types de problèmes à résoudre sur l'image : un point fondamental est celui du parallélisme qu'elles supportent et qui continue à en promouvoir les performances et les capacités.

Il ressort de cette étude que l'utilisation générale de ces machines spécialisées ne peut se faire sans la mise en place d'outils logiciels parfaitement adaptés. Le développement de compilateurs de qualité qui exploite parfaitement le parallélisme inhérent semble être une approche extrêmement brillante déjà adoptée par des constructeurs japonais [Lic85].

[1] vu l'intérêt particulier que nous accordons aux machines ICOTECH et SPHINX, nous reviendrons sur leur présentation dans le chapitre 4

CHAPITRE 2

Chapitre 2

PRESENTATION DU LANGAGE LPSI

1 INTRODUCTION

Dans ce chapitre, nous abordons l'étude de LPSI, un langage de programmation spécialisé dans le traitement des images. LPSI s'appuie essentiellement sur des types image et sur des opérations pour les manipuler aisément.

La définition de LPSI a été guidée par une méthodologie fondée sur les concepts de types abstraits. Une description détaillée de l'approche utilisée est donnée dans [Bel85a] [BB87].

La notion de type abstrait de donnée permet de caractériser un ensemble d'objets uniquement par la spécification des opérations applicables aux objets du type indépendamment de leur représentation [Gut77]

Dans le cas de LPSI, l'étude d'algorithmes usuels de traitement d'images, dans une étape préliminaire, a permis de dégager une liste d'objets. Une spécification formelle des opérations applicables sur ces objets a mis en évidence certaines redondances au niveau de leurs définitions. Un retour sur certaines définitions a finalement conduit aux types image suivants :

position, fenêtre, image, image binaire, région, contour, masque, séquence d'images et filtre.

Dans la suite, nous étudions les propriétés de ces types ainsi que les opérations qu'on peut leur associer dans LPSI.

2 PRESENTATION DES OBJETS DE LPSI

Les objets de LPSI ont été regroupés en trois classes suivant leurs propriétés communes.

La première englobe les objets élémentaires qui interviennent comme composants de base dans les autres objets de LPSI, ou comme outils simples de sélection. Il s'agit de la position et de la fenêtre.

Dans la deuxième classe sont regroupés les objets fondamentaux de LPSI qui constituent le noyau du langage. Ces objets présentent des caractéristiques morphologiques différentes : on y trouve essentiellement des objets surfaciques bidimensionnels (l'image, l'image binaire, la région et le masque) et des objets linéaires (le contour et le graphisme).

Enfin, la dernière classe comporte des objets auxiliaires (le filtre, les fonctions tabulées, la séquence d'images) qui viennent renforcer les primitives du langage.

Tous ces objets se caractérisent par deux types d'informations. Le premier, relatif à leur structure (leurs dimensions par exemple) ou à leur environnement (objet résident en mémoire) est connu dès leur définition et est figé pour toute leur durée de vie.

Le second type regroupe des informations se déduisant par calcul tels que l'histogramme d'une image ou sa moyenne et pouvant varier au cours d'une session de travail.

Comme ces informations sont utiles pendant toute l'utilisation de l'objet, il devient intéressant de les lui associer en termes d'attributs implicites ou calculés suivant le cas. Ainsi au cours d'une application, un objet sera toujours accompagné de ses attributs qu'il est possible de consulter à tout moment.

Dans les paragraphes suivants seront définis les différents objets de LPSI ainsi que leurs attributs.

2.1 Les objets élémentaires

La position

La position est définie par une abscisse et une ordonnée dans le repère de l'objet auquel elle appartient. Signalons à ce propos que chaque objet évolue dans un repère relatif, orienté de gauche à droite et de haut en bas.

Il existe un type d'opérations dans lequel le traitement d'une position dépend de son voisinage, c'est à dire de l'ensemble des positions qui lui sont directement connectées. Dans les objets bidimensionnels, le voisinage des positions situées sur les frontières n'est pas complètement défini ce qui incite dans certains cas à particulariser les traitements sur elles.

Les attributs implicites qui accompagnent une position sont son abscisse et son ordonnée.

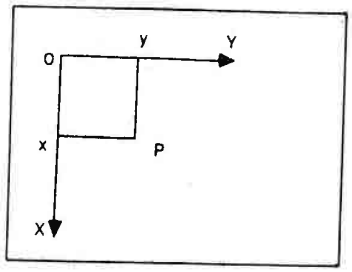


Figure 2.1: La position

La fenêtre

La fenêtre est un outil de sélection de zones rectangulaires. Elle peut s'étendre à tout l'objet pour donner son **cadre**.

Elle est caractérisée par une taille et un point de fixation (son coin supérieur gauche) donné à l'application donc qui lui est indépendant. Ce point n'appartient pas nécessairement au cadre de l'objet. Si la fenêtre n'est pas entièrement définie sur une image par exemple, elle est réduite à la seule partie définie.

Ses attributs implicites sont sa hauteur (H) et sa largeur (L).

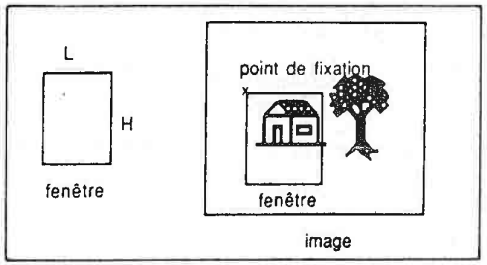


Figure 2.2: La fenêtre

2.2 Les objets fondamentaux

L'image

L'image est la codification, via un capteur, de l'intensité lumineuse émise ou réfléchie par des objets placés dans le champ de perception de ce capteur. Elle peut alors être considérée comme la discrétisation d'une fonction de brillance continue dans un domaine borné.

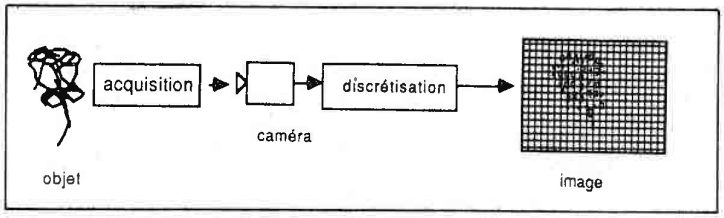


Figure 2.3: L'image

En effet, pour son traitement par ordinateur, une image doit avoir une représentation qui facilite son stockage et son utilisation. Un échantillonnage discret de la fonction d'intensité permet sa représentation sous une forme matricielle I[H, L].

Pour une image noir et blanc, l'intensité lumineuse est convertie en un certain nombre de mesures numériques appelées niveaux de gris, codées généralement entre 0 et 255.

Dans le cas d'une image couleur, à chaque point est associé un triplet de valeurs dont chacune sert à coder une couleur de base (rouge, vert, bleu).

Dans une image de distances, la valeur de chaque point correspond à une mesure de distance séparant l'objet du système d'acquisition.

Un problème concerne la taille des images. Ce paramètre variable d'un système d'acquisition à l'autre incite à ne pas imposer une taille identique à toutes les images.

L'image est riche en attributs qui sont utilisés suivant le type et l'étape de traitement. Les attributs implicites sont son cadre (H, L) et le type de ses valeurs (niveau de gris, couleur, etc...), La nature des traitements effectués sur les images conduit à penser que les attributs

calculés les plus généralement utilisés sont l'histogramme, la dynamique, la variance et la moyenne.

- l'histogramme d'une image est un tableau à une dimension indiquant la fréquence d'un critère mesurable quelconque (niveau de gris par exemple).
- la dynamique est la longueur de l'intervalle dans lequel l'histogramme est nul.
  L'histogramme et la dynamique sont utilisés pour déterminer les modifications des niveaux de gris à effectuer en amélioration d'images.
- la moyenne et la variance d'une image calculées à partir des valeurs de ses niveaux de gris représentent des mesures statistiques qu'il est intéressant de déterminer dans de nombreux domaines d'application telle que l'analyse de texture par exemple [GMT85].

L'image binaire

L'image binaire contient une information à deux niveaux. Une image de caractères, une image de contours, en sont des exemples. L'information dans ce cas est codée par deux valeurs généralement ramenées à 0 et 1.

L'image binaire se distingue de l'image par la nature des traitements effectués sur elle. A titre d'exemple, les approches de suivi de contours sont différentes dans les deux types d'images. De plus, l'image binaire nécessite moins de place pour sa mémorisation.

La région

Une région est un ensemble de points d'image vérifiant des critères d'homogénéité [MW87]. Ces critères peuvent être de nature physique (même intensité), géométrique (même description), topologique (appartenance à une même forme) ou statistique (mêmes indices de texture).

Elle est le résultat de la partition de l'image par un processus de segmentation. Une telle partition donne une description proche de la sémantique de l'image.

Dans LPSI, le concept "région" désigne la carte des régions résultant de la segmentation d'une image. La carte des régions est une représentation homomorphe de l'image segmentée dans laquelle on fait correspondre à chaque point de l'image le numéro de la région à laquelle il appartient. Cette représentation se prête bien aux opérations habituelles sur les régions.

Les attributs qu'il est intéressant de définir dans ce cas sont le nombre de régions et l'histogramme sur les tailles pour l'ensemble des régions d'une image et pour chaque région : sa taille en nombre de points, son périmètre et son cadre.

Le masque

Utilisé généralement dans des opérations de convolution et de morphologie mathématique entre autres, le masque est considéré comme une matrice de coefficients pondérateurs entiers,

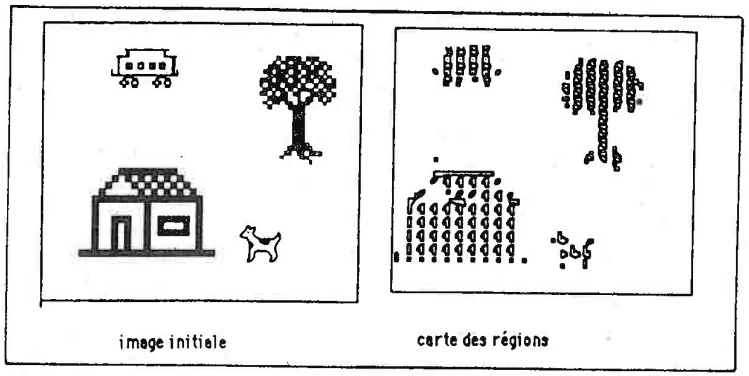


Figure 2.4: La région

réels ou binaires. Sa taille est relativement petite par rapport à celle des images.

Le masque de détection de contours de Sobel donné à la figure 2.5 en est un exemple. Il approche la dérivée en x d'une image faiblement lissée et sert à la mise en évidence de contours.

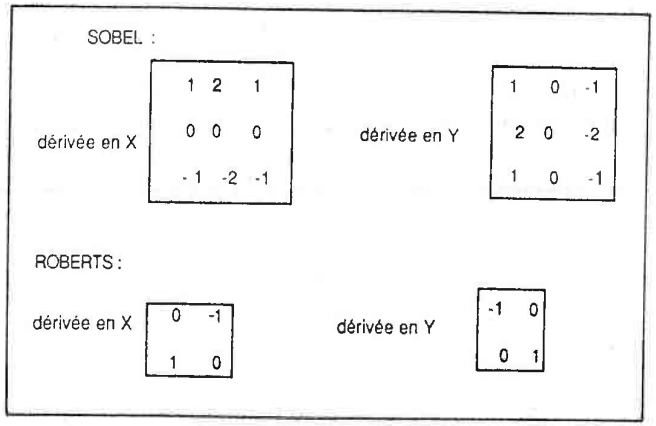


Figure 2.5: Exemples de masques

Le contour

En traitement d'images, il est souvent nécessaire de localiser dans les images des zones particulières pour les analyser par exemple. On parle alors de contours pour indiquer la frontière entre régions. Le contour est extrait de l'image aux lieux où le contraste est important et est donné par une liste de points.

En pratique, le contour n'est pas toujours net, parfois même discontinu, c'est pourquoi nous regroupons sous le concept de "contour" tous les sous-contours qui le composent. L'ordre défini sur ces sous-contours (voir figure 2.6) est arbitraire. Chaque sous-contour est constitué d'une suite ordonnée de positions connexes.

Comme pour la région, les attributs d'un contour peuvent être globaux tels que le nombre de ses composants ou le cadre qui l'englobe. Ce dernier, contrairement aux objets bidimensionnels, est mis à jour de manière dynamique. Il est possible de définir des attributs relatifs à chaque composante comme sa longueur, le nombre de segments qui l'approximent etc...

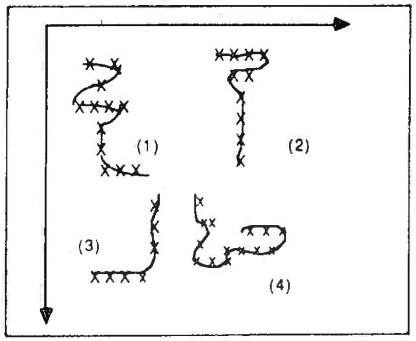


Figure 2.6: Le contour

2.3 Les objets auxiliaires

La séquence d'images

La séquence d'images ou n-image est une structure particulière d'images. Elle regroupe sous le même nom plusieurs sous-images représentant le même objet [Bel83]. Une image multispectrale est de ce type, chacune de ses composantes représente l'image d'une même

scène vue à travers un canal particulier. Pour représenter un objet en mouvement, on crée une séquence d'images temporelles formant ainsi une n-image de cet objet.

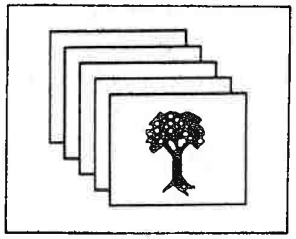


Figure 2.7: La n-image

Les fonctions tabulées

Les fonctions tabulées sont des objets particuliers permettant d'obtenir des informations globales sur les objets dont ils sont issus ou permettant une modification de leurs valeurs suivant une courbe bien définie.

Certaines fonctions tabulées sont une représentation monodimensionnelle du contenu de l'image ; c'est le cas de l'histogramme. D'autres servent à manipuler les intensités, à les coder, par exemple, en pseudo-couleur. Dans ce dernier cas, trois fonctions vont associer à chaque pixel, en fonction de son niveau de gris, un triplet d'intensités sur les canaux standards Rouge, Vert et Bleu, ce qui permet une visualisation en fausses couleurs.

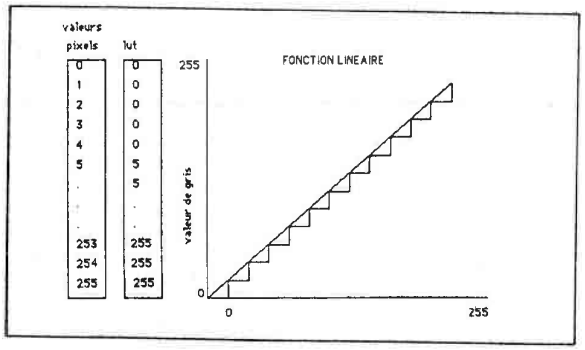


Figure 2.8: Exemple de fonction tabulée linéaire

Le filtre

Il est fréquent en traitement d'images de vouloir sélectionner un sous-ensemble de points et d'appliquer dessus un traitement particulier (manipulation de couleur dans une région, agrandissement d'un motif sélectionné par une fenêtre etc...). Ce sous-ensemble de points peut être défini globalement par la position ou la valeur associée à chacun des points qui le compose.

Plusieurs langages spécialisés (L, PIXAL) ont introduit des mécanismes d'accès globaux pour localiser des zones rectangulaires dans une image. Dans LPSI, le concept de filtre est défini pour généraliser ces mécanismes et permettre plusieurs types d'accès. Le filtre peut être une fenêtre, une condition booléenne portant sur l'emplacement et la valeur des points, une équation de surface ou même un type de donnée structurée tel que le contour.

Le concept de filtre conduit naturellement à la notion d'objet filtré. On appelle objet filtré (noté objet(filtre)), le couple (objet, filtre), dans lequel le filtre permet une lecture, une écriture ou une modification sélective des points de l'objet.

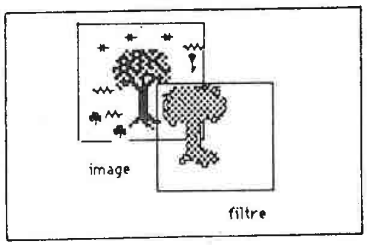


Figure 2.9: Le filtre

Les points sélectionnés ne définissent pas toujours un ensemble compact, ils peuvent au contraire être très dispersés. Le filtre est donc de forme quelconque et de compacité différente.

La sélection des points peut nécessiter des calculs importants mettant en oeuvre des algorithmes complexes (détermination des points à l'intérieur d'un contour par exemple), il est donc important d'envisager une solution qui évite les calculs à chaque utilisation d'un tel filtre. Cette solution consiste à coder les autorisations d'accès pour chaque position. Elle peut être réalisée par une image binaire donnant pour chaque position la valeur 1 ou 0 suiv-

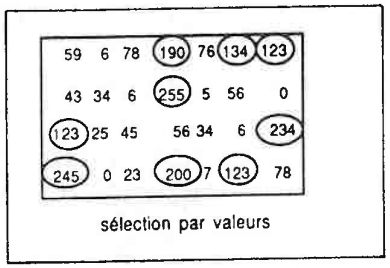
ant qu'elle est sélectionnée ou non.

Dans LPSI, les filtres sont classés suivant le mode d'accès qu'ils réalisent. Il existe trois types de filtres :

Les filtres à accès **associatif** sont à accès conditionnel sur les valeurs. Un prédicat portant sur les valeurs de l'objet doit être vérifié.

exemple

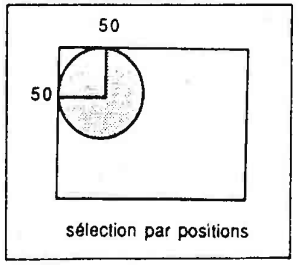
Nous pouvons imaginer un filtre qui sélectionne parmi les points d'une image ceux dont la valeur est supérieure à un certain seuil s. Ce filtre désignera les points suivants :



Les filtres à accès **direct** sont à accès sélectif sur les positions. Un prédicat quelconque portant sur les coordonnées des positions de l'objet doit être vérifié.

exemple

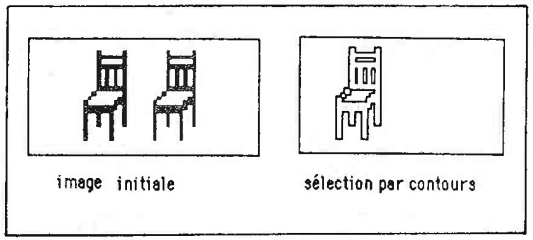
Le filtre qui localise dans une image les points appartenant à un disque centré au point (50,50) et vérifiant l'équation $((x-50)^2+(y-50)^2 < 50^2)$ est donné par la figure suivante :



Enfin, les filtres translatables permettent une sélection grâce à des objets intermédiaires. En plus de la fenêtre, le contour et la région peuvent également être utilisés comme filtres, le premier pour délimiter des frontières de régions dans une image, le second pour y localiser une région particulière.

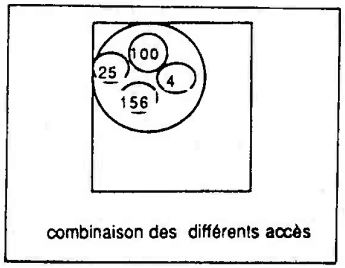
exemple

Le filtre suivant utilise un contour pour désigner dans une image les frontières d'une région particulière.



remarque

La combinaison des différents modes d'accès est possible. Elle conduit à la vérification simultanée de plusieurs prédicats (I(F1 ou F2 et F3)). Ainsi il devient possible de désigner globalement des points appartenant à un disque dont les valeurs sont supérieures à un seuil :



Allant de la simple condition aux équations complexes de surface, le filtre constitue dans le langage LPSI, un outil de sélection global très puissant.

Cette présentation informelle des objets image a permis de dégager pour chaque objet ses principales caractéristiques, elle facilite dans la suite, sa description en termes syntaxiques du langage.

3 DESCRIPTION DES TYPES EN LPSI

LPSI est une extension de PASCAL. Afin de maintenir la cohérence et l'homogénéité de l'ensemble, sa syntaxe est largement inspirée de celle de PASCAL.

Un programme LPSI a l'organisation générale d'un programme PASCAL. On y trouve :

- des sections de déclarations des constantes, types et variables du programme, similaires à celles de PASCAL [JW80].
- une section de définition des attributs pour les différents objets de LPSI. Si les attributs implicites d'un objet lui sont systématiquement associés dès sa déclaration, ce n'est pas le cas des attributs calculés. Ces derniers sont tellement différents d'un objet à un autre qu'il est impossible de les prédéfinir dans le langage. Cette section est alors prévue pour leur définition explicite.
- une section de définition des supports dans laquelle sont déclarés tous les supports utilisés dans le programme.
- une section de définition des procédures et des fonctions.
- une section pour le corps du programme.

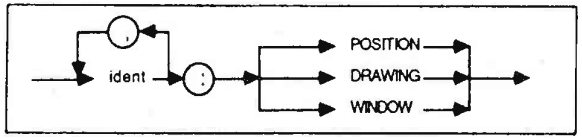
Nous allons exposer les principaux éléments syntaxiques du langage. Pour une compréhension immédiate, dans les cas nécessaires, la présentation se fera par des diagrammes très simplifiés. La syntaxe complète du langage est donnée à la fin de ce document.

3.1 La déclaration des objets

a) les objets simples

Les objets simples sont déclarés par la donnée d'un type prédéfini LPSI. Il s'agit de :
POSITION pour la position,
DRAWING pour le contour,
WINDOW pour la fenêtre.

syntaxe



attributs implicites

Une notation suffixée permet de rattacher chaque attribut à la variable de l'objet qui lui correspond.

<variable-position>**.X** : abscisse,
<variable-position>**.Y** : ordonnée,
<variable-window>**.H** : hauteur,
<variable-window>**.L** : largeur,
<variable-drawing>**.W** : cadre,
<variable-drawing>**.NS** : nombre de composantes,

exemple

p : **position**;
w : **window**;
d1, d2 : **drawing**;

b) les objets bidimensionnels

Pour déclarer ces objets, il convient de préciser :

- leur type de base :

IMAGE pour l'image,
BINARY pour l'image binaire,
REGION pour la région,
MASK pour le masque.

- leur cadre (hauteur, largeur), si le cadre n'est pas précisé alors les objets sont surdimensionnés.

- le type de l'information élémentaire :

  pour l'image
  GL (pour le niveau de gris), **REAL** (pour les images de distance).
  GL est le type pris par défaut.
  Pour le masque
  BIN (pour les masques binaires), **INTEGER** ou **REAL** (pour les masques de coefficients).
  BIN est le type pris par défaut.

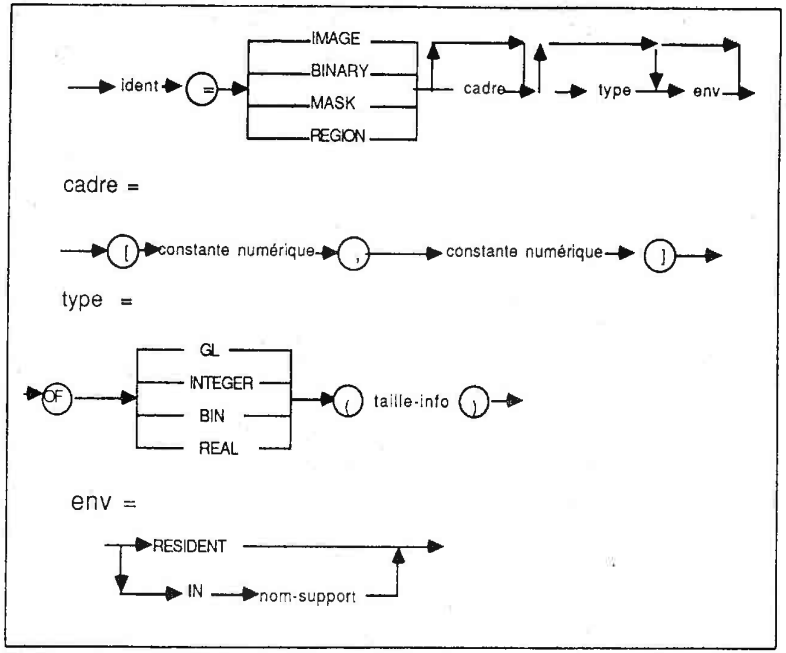
- la taille de l'information élémentaire prise entre 4, 8, 16, 32 bits (8 étant la valeur prise par défaut).

- leur environnement :

  devant l'impossibilité de mettre en mémoire centrale une très grande quantité d'informations, nous avons prévu la possibilité de définir les objets bidimensionnels à partir d'un support externe. Cette façon de procéder évite l'encombrement de la mémoire image et supprime toute contrainte sur la taille et le nombre d'objets que l'on peut manipuler dans un programme. Ainsi deux possibilités s'offrent à l'utilisateur :

  - la première consiste à déclarer des objets résidents en mémoire centrale grâce à l'attribut d'environnement **RESIDENT**,

  - la seconde permet de définir des objets externes qui seront traités par sous-ensembles plus petits lus et écrits à partir d'un support externe, le tout devant être géré à la manière d'une mémoire virtuelle. Le mot clé **IN** suivi d'un nom de support est la syntaxe utilisée pour la définition d'objets virtuels. Les caractéristiques du support sont définies dans la section correspondante.
    Par défaut, les objets sont résidents.



attributs implicites
<objet>**.W** : cadre de type window,
<objet>**.S** : taille de l'information élémentaire de type integer.

exemple
```
im1, im2 : image[256,256] of gl(4) resident;
reg  : region[256,256] in nom_support;
(*nom_support sera défini plus loin
reg.W vaut [256,256];*)
masque : mask[5,5] of bin;
```

c) la n-image

La n-image se définit par la donnée de :

- son type **NIMAGE**,

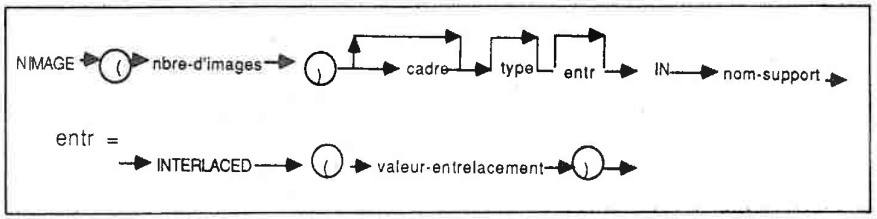
- le nombre d'images qui la composent,

- le type d'entrelacement sur le support,
  on code par 1 l'entrelacement point par point et par 2 l'entrelacement ligne par ligne,

- les caractéristiques des images qui la composent : leur cadre et le type de l'information élémentaire.

- son environnement.

syntaxe



attributs implicites
<variable-n-image>.**N** : nombre d'images composantes,
<variable-n-image>.**E** : type d'entrelacement.

exemple
nim : **nimage**(3)[128,128] of **gl**(8) **interlaced**(2) **in** nom_support;
nim.N = 3 et nim.E = 2

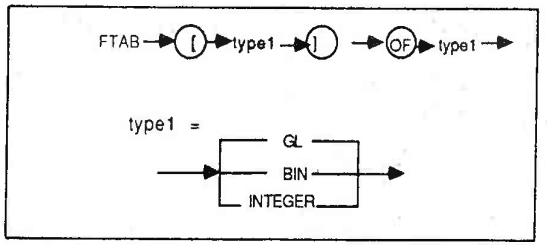
d) la fonction tabulée

Pour déclarer une fonction tabulée, il suffit de préciser :

- son type **FTAB**

- le type de ses valeurs d'entrée et celui de ses valeurs de sortie :
  BIN, **GL**, **INTEGER**,

syntaxe



exemple
lut1 : **ftab**[**gl**] of **gl**;

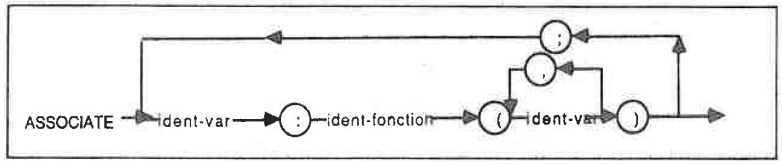
3.2 Association d'attributs

L'association d'un attribut est réalisée dans la section **ASSOCIATE** du programme par la donnée :

- du nom de la variable attribut,

- de la liste des objets auxquels il est associé.

syntaxe



exemple

var
cont : **binary**[256,256];
im1, im2 : **image**[256, 256] of **gl**;
histo : **ftab**[gl] of **integer**;
(* histo tabule pour chaque niveau de gris (gl), le nombre de ses occurences (integer) *)
associate
histo : histogram(im1);
cont : extcont(im2);

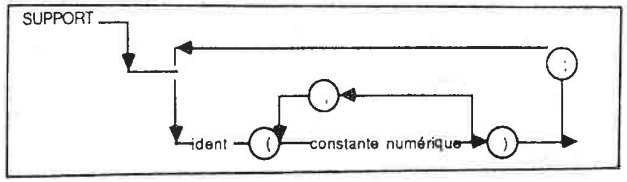
"histo " est l'attribut histogramme de l'image im1. Cet attribut est calculé par la fonction "histogram" fournie par l'utilisateur.
De la même façon, "cont" est l'image des contours de l'image im2, ces contours sont extraits par la fonction "extcont".

3.3 Définition des supports

Dans LPSI, les supports sont destinés à la mémorisation des objets virtuels et à l'archivage des objets en général. Inspiré du logiciel IMAGE 7[1] [Vog84] destiné à gérer les périphériques spécifiques au traitement d'images dans les machines ICOTECH, chaque support est repéré par un descriptif donnant ses caractéristiques. Les plus importantes sont les suivantes :

- la nature du support, (bande magnétique (0), disque magnétique (1), mémoire vidéo (2)).
- le mode d'utilisation,
  les supports sont autorisés en lecture (0), en écriture (1) ou en lecture/écriture (2). Si le support est destiné à un objet virtuel, il doit toujours être autorisé en lecture/écriture.
- les paramètres du point d'incrustation pour la mémoire vidéo,
- le nom du fichier physique dans le cas d'un support magnétique.

Cette description des supports est faite dans la section SUPPORT prévue à cet effet.
syntaxe



exemple
support
S(2, 1, 256, 256);
Il s'agit du support mémoire vidéo, autorisé en écriture avec un point d'incrustation égal à (256,256).
virt(1, 2, 0,0, fichim);
virt est un support magnétique autorisé en lecture/écriture, fichim étant le nom physique du fichier image.

[1] la version actuelle s'appelle IMAGE 8

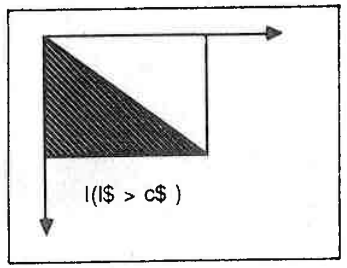
3.4 Définition du filtre

Il n'est pas prévu dans LPSI, un type spécial pour le filtre. Sa définition consiste en l'expression des propriétés d'accès qu'il réalise.

Dans un filtre à accès direct, des noms génériques (l$, c$, r$) sont utilisés pour désigner respectivement la ligne, la colonne et le rang des positions dans une structure bidimensionnelle ou linéaire. Ces noms constituent les variables de l'expression conditionnelle qui le réalise.

(l$ > c$) : sélectionne les points dont l'abscisse dépasse l'ordonnée.

(r$ > 3) : appliqué à une variable de type drawing, ce filtre sélectionne les sous-contours dont le rang est supérieur à 3.



Dans ce même cas, les accès peuvent être faits par des filtres constants.
(l1..l2,c1..c2), **(l1..l2,)**, **(,c1..c2)** réalisent des accès à travers des fenêtres constantes, l1 et l2 délimitant la hauteur de la fenêtre de sélection, c1 et c2 sa largeur.

(r), **(r1..r2)** permettent l'accès par rang à des sous-ensembles de points dans la région, le contour et la n-image.

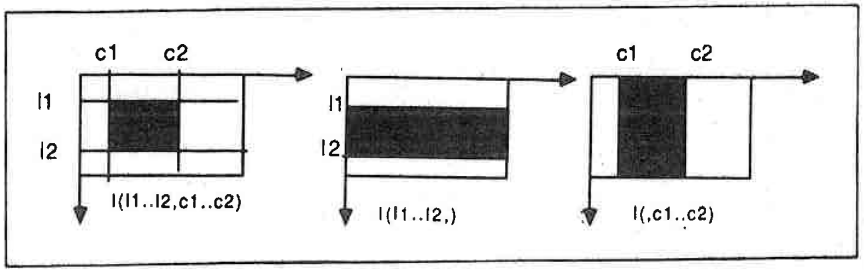
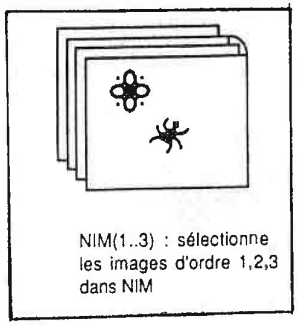
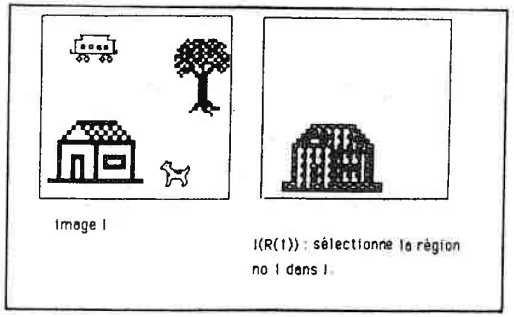


Figure 2.10: Exemples de filtres constants

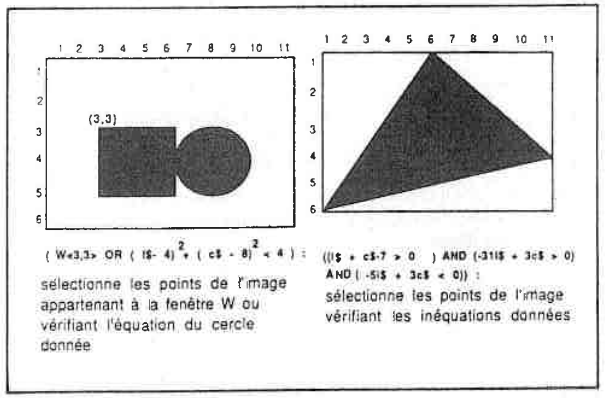


Dans un filtre à accès associatif, v$ est le nom générique de la valeur d'un point. v$ constitue la variable de l'expression qui réalise le filtre.
(v$ > 10) sélectionne les points dont la valeur excède 10.

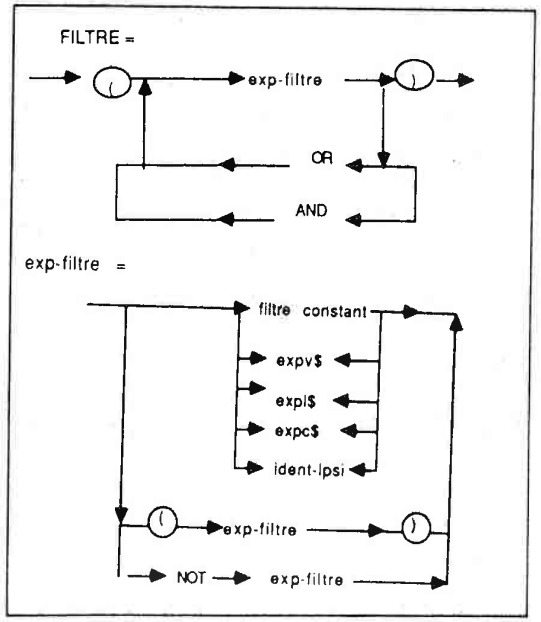
Un filtre translatable manipule des noms d'objets sélecteurs (la fenêtre, le contour, la région).
(r(i)) : si r est une région, ce filtre sélectionne tous les points ayant l'étiquette i dans r.



Les différents filtres peuvent être combinés par les opérateurs logiques classiques : **AND** réalise leur intersection, **OR** leur union et **NOT** calcule leur complément.



syntaxe



expv$: toute expression arithmétique dont la variable est v$
expl$: = = = = = = = l$
expc$: = = = = = = = c$
ident-lpsi : identificateur de binaire, de région, de région ou de tracé

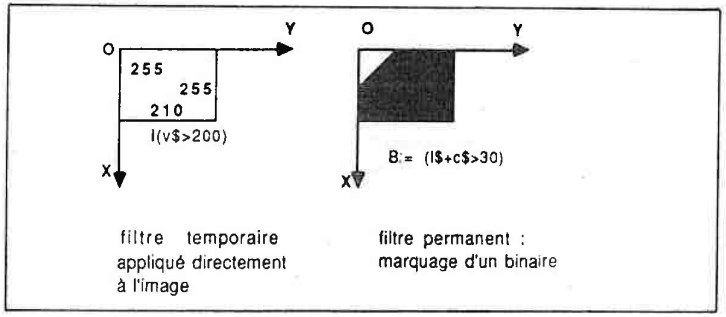
3.5 Utilisation du filtre

Le filtre peut s'utiliser de deux manières :

- l'expression filtre est appliquée directement sur l'objet au moment de son utilisation :
  I ((l$ mod 5 = 0) OR (c$ mod 5 = 0)),
- le filtre est matérialisé par une variable binaire déclarée dans le programme et initialisée à une expression filtre. Cette variable donne un filtre permanent appliqué à l'objet en tant que filtre translatable.
  B := ((l$ mod 5 = 0) OR (c$ mod 5 = 0));
  I(B);

La première manière est recommandée lorsque le filtre est temporaire, c'est-à-dire intervenant une seule fois dans le programme. Cela évite une allocation de place en mémoire pour la variable binaire. Dans le cas d'un filtre permanent, le passage par un binaire est le meilleur moyen d'éviter son interprétation systématique à chaque utilisation, comme nous le verrons dans le chapitre suivant.

exemple



remarque

Un filtre à accès associatif ne peut s'utiliser que de la première manière car il ne peut être considéré indépendamment de l'objet auquel il est appliqué.

Après avoir mis au point la description des objets de LPSI, il est normal de se préoccuper de la façon de les utiliser dans un programme. Le rôle des prochains paragraphes consiste à définir syntaxiquement et sémantiquement toutes les actions possibles sur eux.

4 LES OPERATIONS DE BASE

Les opérateurs de base de LPSI se partagent en plusieurs groupes :

- les opérateurs arithmétiques exprimés sur les images,
- les opérateurs logiques exprimés sur les images, les binaires et les masques,
- les opérateurs sur les fenêtres, les régions et les contours.

Les expressions qui en découlent LPSI sont de type image c'est à dire combinant deux ou plusieurs objets image.

Dans la plupart des cas, les opérateurs manipulent globalement les objets.

4.1 Les opérations sur les objets élémentaires

Les opérations sur les objets élémentaires sont simples. Ce sont essentiellement des opérations de création et de modification.
Pour la création, il suffit de donner la valeur des attributs implicites. Nous donnons ici des exemples de création de position et de fenêtre.

p := [abscisse,ordonnée] pour la position,
w := [hauteur,largeur] pour la fenêtre.

La modification est réalisée grâce à :

- une mise à jour partielle des attributs implicites

$$\left\{\begin{array}{l}\mathbf{p.X}\\ \mathbf{p.Y}\\ \mathbf{w.L}\\ \mathbf{w.H}\end{array}\right\} := \textbf{expression de type entier}$$

- une mise à jour globale des objets élémentaires

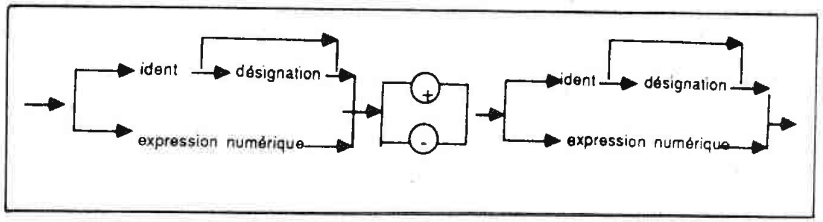
$$\left\{\begin{array}{l}\mathbf{w}\\ \\ \mathbf{p}\end{array}\right\} := \textbf{expression de type fenêtre ou position}$$

4.2 Les opérations sur les objets bidimensionnels

Les opérations arithmétiques

Les opérateurs utilisés sont les opérateurs arithmétiques classiques. Si une opération utilise plusieurs images, celles-ci doivent se rapporter au même objet (deux vues d'une même scène, l'image et sa convoluée etc...).

a) somme et différence d'images



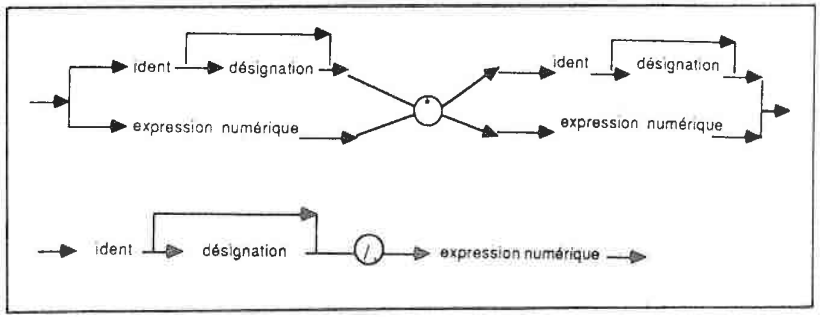
sémantique

Si A et B sont deux images et v une valeur numérique, la somme et la différence sont définies de la manière suivante :

$$A \pm \left\{\begin{array}{l}B\\ \\ v\end{array}\right\} = A_{ij} \pm \left\{\begin{array}{l}B_{ij}\\ \\ v\end{array}\right\} \forall (i,j) \in cadre(A)[\cap cadre(B)]$$

b) produit et division d'images

syntaxe



sémantique

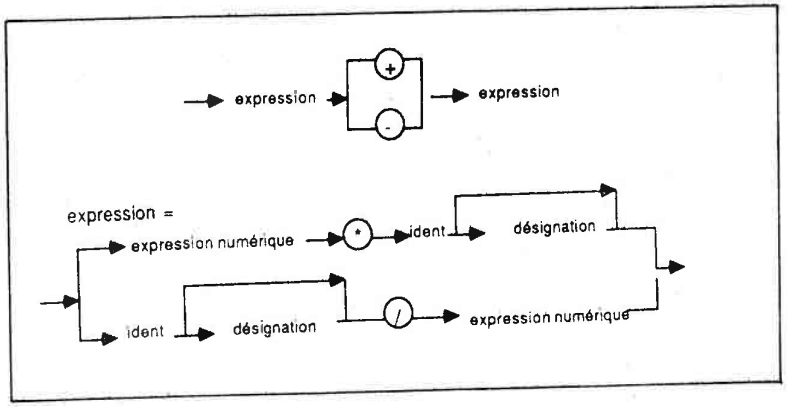
Si A est une image et c une valeur numérique,

$c \times A = c \times A_{ij}\ \forall (i,j) \in cadre(A)$;

$$A \left\{ \begin{matrix} \text{div} \\ / \end{matrix} \right\} c = A_{ij} \left\{ \begin{matrix} \text{div} \\ / \end{matrix} \right\} c \quad \forall (i,j) \in cadre(A);$$

c) les combinaisons linéaires d'images

syntaxe



sémantique

Si A, B sont des images et c et d des constantes numériques :

$c \times A + d \times B = c \times A_{ij} + d \times B_{ij}\ \forall (i,j) \in cadre(A) \cap cadre(B)$;

Le résultat est encore une image.

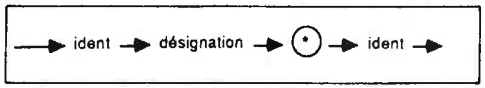
Une combinaison linéaire souvent utilisée est la suivante :

$$I3 := 1/2 \times I1 + 1/2 \times I2;$$

C'est un moyen simple et rapide pour réduire le bruit blanc introduit par le capteur ; plus généralement on prend n images et on calcule leur moyenne.

d) Le produit d'une image par un masque

syntaxe



sémantique

Si A est une image, M un masque de convolution et W une fenètre de même taille que le masque appliquée au point p :

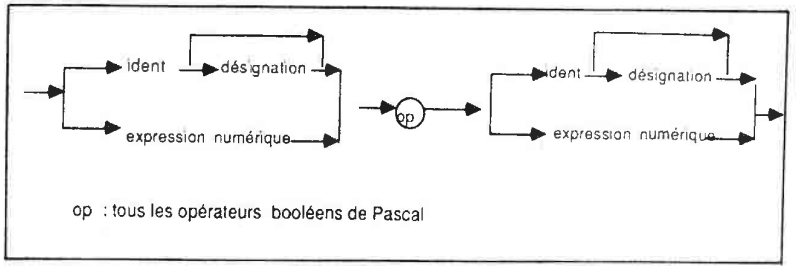
$A(W < p >) \times M = A_{ij} \times M_{ij}\ \ \forall (i,j) \in cadre(A) \cap W < p >$

avec W<p> = CADRE(M)

Le résultat est une image intermédiaire de la taille de W.

Les opérations logiques

syntaxe



Elles utilisent les opérateurs logiques classiques et sont de deux sortes :

a) opérations logiques sur les images

1. elles interviennent dans les conditionnelles globales de LPSI (primitive IFG), sont évaluées point à point et donnent un résultat **booléen global**.

sémantique

Si A et B sont des images, v une valeur numérique et opl un des opérateurs de comparaison :

$$\text{A opl } \begin{Bmatrix} \text{B} \\ \text{v} \end{Bmatrix} = \begin{cases} \text{vrai} & si\ \forall\ i,j\ \in\ cadre(A)\ [\cap\ cadre(B)]\ ,\ A_{ij}\ opl\ \begin{Bmatrix} B_{ij} \\ \text{v} \end{Bmatrix} = \text{vrai} \\ \text{faux} & \text{sinon} \end{cases}$$

Le résultat est une image booléenne.

2. elles interviennent dans une opération d'affectation pour la création d'un binaire, résultat des tests **partiels** point à point entre deux objets bidimensionnels quelconques.

sémantique

$$\text{A opl B} = \begin{cases} 1 & \forall\ (i,j) \in cadre(A) \cap cadre(B) \\ & \text{tel que } A_{ij}\ opl\ B_{ij}\ =\ vrai \\ 0 & \text{sinon} \end{cases}$$

Le résultat est une image binaire.

Les opérations sur les régions

Inspirées des algorithmes de segmentation d'images, les opérations suivantes réalisent dans LPSI les opérations usuelles sur les régions (création, fusion, destruction de régions etc...).

Si $P, P_1, P_2 ... P_n$ désignent des positions, I une image, R une région et F un filtre quelconque alors :

$$R[P]$$

donne le numéro de la région à laquelle appartient le point P.

$$P\ IN\ R(i)$$

vérifie l'appartenance du point P à la région i.

$$R(i_1)\ := R(i_1) + R(i_2)\{+.....R(i_n)\}$$

fusionne deux ou plusieurs régions de rangs quelconques.

$$R(i_1)\ := R(i_1..i_2)$$

fusionnent des régions de rangs consécutifs.
Les régions fusionnées sont affectées du même numéro i_1. Les autres numéros sont libérés et peuvent être réutilisés plus tard.

$$R(i)\ := \left\{ \begin{matrix} R(i) + P1\{+P2... + P_n\} \\ I(F) \end{matrix} \right\}$$

réalise l'adjonction d'un ou de plusieurs points à une région.

$$R\ := \left\{ \begin{matrix} R + P1\{+P2 + P3...Pn\} \\ I(F) \end{matrix} \right\}$$

crée une nouvelle région à partir d'un ou de plusieurs points.

$$R[P]\ := 0$$

élimine un point d'une région.

$$R(i)\ := 0$$

détruit une région.

remarque

La gestion des numéros d'ordre internes des régions est assurée par LPSI. Les numéros libérés au moment de la destruction ou de la fusion de régions sont réutilisés au cours d'une création. Il est néanmoins possible à l'utilisateur d'accéder à l'étiquette attribuée à un point quelconque.

Les opérations sur les contours

Afin de conserver une homogénéité syntaxique avec les régions, nous reprenons les mêmes notations pour les contours bien que la sémantique en soit tout autre.

$$P\ IN\ C(i)$$

vérifie l'appartenance d'un point à un sous-contour.

$$C(i) + C(j)$$

concatène deux sous-contours.

$$C(i) := 0$$

détruit un sous-contour.

$$C := C + P$$

crée un nouveau sous-contour à partir de la position P.

$$C(i) := C[i] + P$$

rajoute un point à la fin d'un sous-contour.

$$C(i) := P + C(i)$$

rajoute un point au début d'un sous-contour.

remarque
Pour accèder à un sous-contour, il faut toujours préciser son numéro.
Dans toutes les opérations qui modifient un contour, les critères de connexité doivent rester vérifiés.

les opérations sur le masque

Comme le masque est généralement composé de coefficients qui se répètent, une syntaxe particulière a été adaptée pour sa création :

$M := \{a_1 \times\}[ligne_1]\{, a_2 \times [ligne_2], ..a_n \times [ligne_n]\}$

a_i est le coefficient de répétition d'une ligne, et $ligne_i$ est définie par

$ligne_i = \{b_1\} \times [coeff_1]\{, b_2 \times [coeff_2], ..b_n \times [coeff_n]\}$

b_i est le coefficient de répétition d'un coefficient dans une ligne.[3]

exemple
$M := 3 \times [1, 0, 1]$;
M est un masque 3 x 3 dans lequel les trois lignes sont identiques.

Les entrées/sorties

Deux ordres (READG, WRITEG) rendent possibles les transferts entre les différents supports et l'archivage des objets de LPSI. Tous les détails techniques de bas niveau sont masqués à l'utilisateur. Son intervention se limite à la définition des supports.

[3]Les éléments entre {} sont optionnels.

READG(liste-supports l$,c$) liste-objets;
WRITEG(liste-supports l$,c$) liste-objets;

où

- **liste-supports** est une liste de supports définis dans la section **SUPPORT**,
- **liste-objets** est une liste d'objets LPSI bidimensionnels. Il doit exister une correspondance en nombre entre les supports et les objets : l'objet d'ordre i dans la liste des objets est lu/écrit sur le support d'ordre i dans la liste des supports.
  l$ et c$ conditionnent le parcours (ligne par ligne ou colonne par colonne) dans les supports.

5 LES INSTRUCTIONS DE LPSI

Les instructions de LPSI comportent essentiellement l'affectation et les structures de contrôle :

- l'affectation est une opération de base dans laquelle les objets ne sont pas nécessairement de même type et de même taille,
- les structures de contrôle comportent une conditionnelle globale et des itérations parallèle et séquentielle. Ces structures sont justifiées par la manipulation globale des objets et les traitements répétitifs sur leurs valeurs.

remarque
Il est également possible d'accéder, à partir d'un programme LPSI, à toutes les structures de contrôle usuelles de PASCAL.

Etant donné les variantes introduites au moment de la définition des objets (objets résidents ou virtuels, filtrés ou non), il est nécessaire de préciser certaines règles pour leur utilisation :

règle 1

Qu'ils soient résidents ou virtuels, les objets sont utilisés avec la même syntaxe et présentent un comportement identique dans un programme.

règle 2

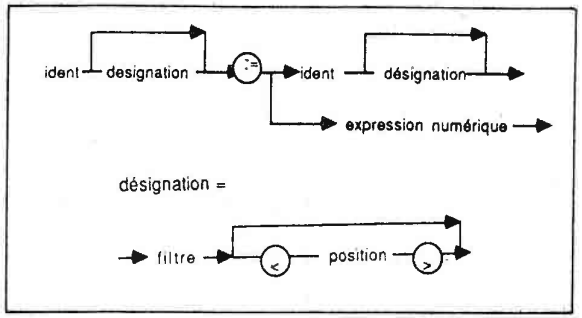
Aucune contrainte sur les tailles n'est imposée si plusieurs objets sont utilisés dans une même opération. Cette dernière a lieu par rapport au cadre minimal. Si les objets sont filtrés, seuls les points sélectionnés sont affectés. L'opération a lieu sur les points sélectionnés simultanément dans l'ensemble des objets.

Dans les paragraphes qui suivent, la notation CADRE(I) désigne la plus grande fenêtre qui englobe l'objet I. Le cadre minimal est défini comme étant la plus petite fenêtre commune à l'ensemble des objets intervenant dans une même expression.

5.1 L'affectation

L'affectation d'images

syntaxe



sémantique

1. l'affectation d'une constante à une image est utilisée pour une initialisation globable de celle-ci à une même valeur. Cette valeur peut être, par exemple, une couleur de fond pour l'image.

2. l'affectation entre deux images permet de créer une image à partir d'une autre si celles-ci sont de même taille ou d'incruster une image sur une autre dans le cas contraire. Dans les deux cas, elle est effectuée point à point suivant les règles déjà annoncées.

L'affectation entre contours et binaires

sémantique

1. L'affectation d'un contour à un binaire permet de définir une image binaire des contours. Cette opération est intéressante dans la mesure où il est souvent utile de superposer après une extraction de contours, l'image des contours à l'image de départ.

2. L'affectation inverse est plus difficile à mettre en oeuvre car elle nécessite des techniques de suivi de contours pour pouvoir construire la liste ordonnée des points. De plus, elle n'a pas de signification si ce binaire est une surface.

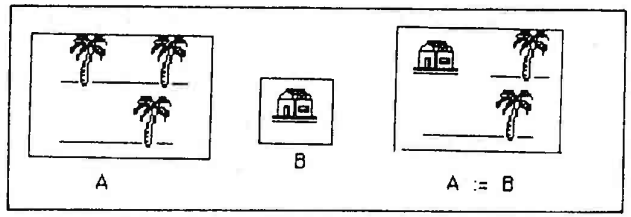
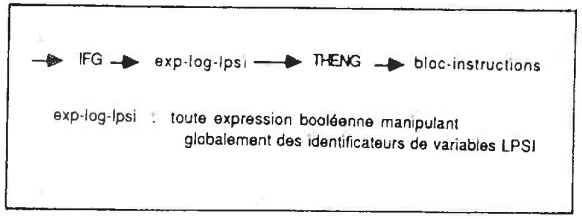


Figure 2.11: Affectation d'images

5.2 Les structures de contrôle

La conditionnelle globale

syntaxe



sémantique

La condition a lieu pour tous les points sélectionnés dans l'expression globale. Elle ne sera égale à vrai que si elle est partout vérifiée.

exemple

ifg A(W1) > B(W2) <> 0
theng C := A(W1) - B(W2)
elseg C := 0;

Dans cet exemple simple, C vaudra 0 si au moins un point de B dépasse son homologue dans A.

Dans LPSI, il n'existe pas une structure propre pour une conditionnelle locale car elle

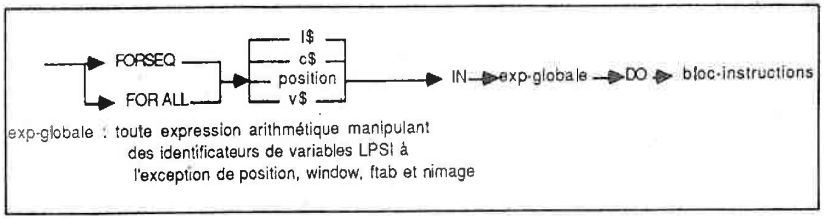
peut être obtenue en combinant l'itération parallèle avec la conditionnelle de PASCAL comme nous le verrons dans le paragraphe qui suit.

Les itérations

En traitement d'images, il existe deux modes d'application pour une fonction. Dans le premier, les points sont traités en même temps car pour calculer un point on n'a pas besoin des résultats obtenus pour les autres. On peut donc gagner énormément de temps d'exécution en partageant l'objet en régions qui seront traitées en parallèle. Dans le second cas, les points sont traités dans un ordre imposé car ils dépendent les uns des autres.

Il est donc nécessaire que le langage offre des structures qui puissent exprimer ces deux modes de traitement. Dans LPSI, les itérations séquentielle et parallèle ont été étudiées et introduites.

syntaxe



sémantique

Dans l'itération séquentielle, le parcours est conditionné par les valeurs de l'expression :

- **I$** impose un balayage ligne par ligne (dans les objets bidimensionnels).
- **c$** impose un balayage colonne par colonne.
- ident-position et **v$** utilisent l'ordre prédéfini par le langage, **v$** désigne la valeur du point examiné.

Dans l'itération parallèle, l'ordre de parcours est sans importance. L'essentiel est d'atteindre tous les points.

exemple 1

Cet exemple calcule un zoom en extension de l'image A par duplication de la valeur de ses points. Si n est le facteur d'extension et C l'image résultat, on obtient :


```
for all [x,y] in A do
begin
x1 := (x-1)*n +1;
y1 := (y-1)*n +1;
C(W<[x1,y1]>):= A[x,y]
(* la valeur du point A[x,y] sera répétée dans la partie d'image sélectionnée par
la fenêtre W *)
end;
```

exemple 2

Cet exemple calcule l'histogramme histo d'une image I.

```
histo := 0;
forseq v$ in I do
histo[v$] := histo[v$] + 1;
```

6 CONCLUSION

Le langage pour le traitement d'images LPSI comprend des primitives et des structures de contrôles dont la syntaxe étend celle de PASCAL. Il permet la déclaration de types image et introduit les opérations nécessaires pour les manipuler aisément. Il apporte une solution au problème d'encombrement de la mémoire en permettant la définition d'objets virtuels. Les mécanismes d'association d'attributs contribuent quant à eux, à l'amélioration des temps de traitement en évitant les calculs inutiles.

LPSI permet de désigner globalement une ou plusieurs régions dans une image de manière directe et/ou associative. Il devient alors possible de localiser un traitement dans les parties sélectionnées. Des instructions globales et des itérateurs sur des ensembles de points viennent compléter les primitives qui font de LPSI un outil suffisamment général pour exprimer le traitement d'images dans sa plus grande étendue.

L'efficacité de LPSI repose sur la souplesse de son utilisation mais aussi sur la rapidité avec laquelle sont exécutés les traitements qu'il décrit. Si la puissance de la machine joue un rôle primordial dans ce dernier point, il n'en reste pas moins vrai que les mécanismes adoptés dans l'implantation y contribuent considérablement.

Comment implanter LPSI de façon efficace sans pénaliser le temps d'exécution ? Cet aspect du problème est synonyme d'optimisation du code produit en tenant compte des possibilités de la machine cible. Le chapitre suivant consacré au traducteur de LPSI s'étendra sur tous ces problèmes en justifiant les raisons qui nous ont guidés dans nos choix.

CHAPITRE 3

Chapitre 3

LE PRE-COMPILATEUR LPSI

1 INTRODUCTION

Ce chapitre étudie le pré-compilateur LPSI. Rappelons qu'un pré-compilateur admet en entrée un programme dans un langage de haut niveau pour générer du code dans un autre langage de haut niveau. Dans le cas de LPSI, le langage cible choisi est PASCAL.

Dans cette étude, la partie relative aux aspects classiques de la compilation est introduite brièvement. Nous insistons beaucoup plus sur la réalisation proprement dite en essayant de mettre en évidence tous les mécanismes utilisés. Nous nous intéressons dans un premier temps au choix des structures d'accueil adéquates aux objets LPSI en tenant compte de leur environnement et de leurs liens d'association. Dans une seconde étape, nous présentons les possibilités d'implantation des opérations et des structures de contrôle.

La pré-compilation des programmes LPSI s'effectue en deux étapes : l'analyse et la traduction

La phase d'analyse permet de vérifier si un programme LPSI est conforme aux règles syntaxiques et sémantiques induites par la grammaire du langage. Cette phase est prise en charge par l'analyseur lexico-syntaxique.

La phase de traduction transforme dans un programme les instructions propres à la couche LPSI en instructions PASCAL. La transformation s'effectue à partir d'une représentation arborescente du programme (l'arbre de syntaxe abstraite) engendrée par l'analyseur.

Les paragraphes suivants sont consacrés à la description de l'analyseur et du traducteur de LPSI.

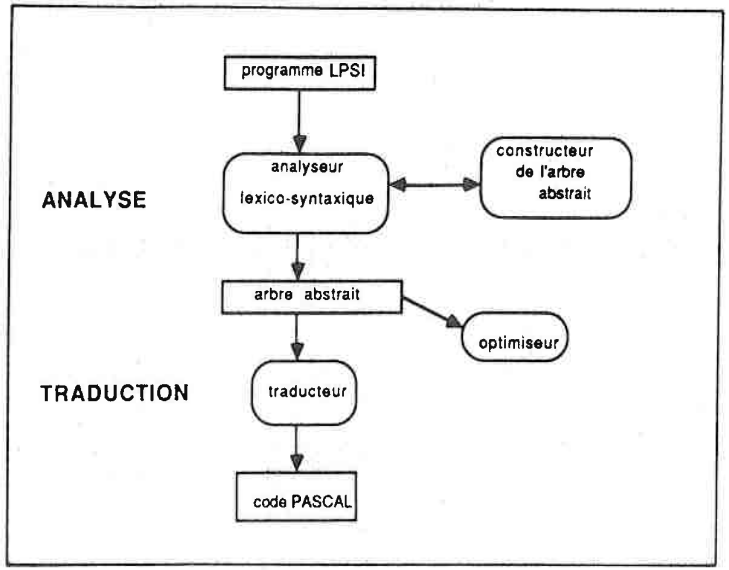


Figure 3.1: Schéma du pré-compilateur LPSI

2 L'ANALYSEUR LEXICO-SYNTAXIQUE DE LPSI

Les analyseurs lexical et syntaxique ont été réalisés par deux outils logiciels LEX et YACC.

2.1 Présentation sommaire de l'analyseur lexical de LPSI

L'analyseur lexical d'un langage a pour rôle de reconnaitre les unités lexicales faisant partie du texte source. Dans le cas de LPSI, cet analyseur est écrit dans le formalisme de LEX. Chaque unité lexicale du langage est décrite par une règle de la manière suivante :

<expression-régulière> {action-associée}

où expression-régulière désigne la chaîne à retrouver dans le texte source, et action-associée, un bloc d'instructions écrit dans le langage C, exécuté toutes les fois que l'expression régulière est reconnue [LS75]. La principale action de ce bloc consiste à renvoyer le code de l'unité identifiée.

exemple

[a-za-z][a-za-z_]* { return(identificateur);}

En plus des unités lexicales de PASCAL, LPSI introduit les mots clés suivants :

position window image binary region mask nimage drawing ftab gl bin interlaced resident support associate ifg theng elseg forall (for all) forseq readg writeg l$ c$ r$ v$.

A partir d'une telle spécification, LEX génère l'automate qui réalise l'analyseur proprement dit.

2.2 Présentation sommaire de l'analyseur syntaxique de LPSI

L'analyseur syntaxique d'un langage est chargé de déterminer si une suite d'unités lexicales peut être engendrée par l'ensemble des règles syntaxiques du langage. Dans le cas de LPSI, il est écrit dans le formalisme de YACC [Joh81], dans lequel doivent être spécifiés :

- l'axiome de la grammaire,
- le vocabulaire terminal ou "tokens",
- les non terminaux,
- l'ensemble des règles syntaxiques, dans lequel chaque règle a la forme suivante :

<non-terminal> : <corps de la règle> {action associée};

Une règle est donc écrite sous une forme proche de la BNF (Backus Naur Form) à laquelle sont associées des actions sémantiques exécutées à chaque réduction par cette règle. Les symboles ":" et ";" appartiennent à la syntaxe de la règle. Les composants d'une règle sont séparés par le signe "|". L'analyseur engendré est ascendant, la récursivité à gauche est donc autorisée dans les règles.

exemple

```
type-image  : IMAGE cadre type env ';'
            ;
cadre       : {règle vide}
            | '[' constante ',' constante ']'
            ;
type        : {règle vide}
            | type_info taille_info
            ;
type_info   : INTEGER
            | REAL
            | GL
            ;
```

etc...

LEX et YACC sont conçus pour s'articuler entre eux. La communication est réalisée par l'intermédiaire de variables globales prédéfinies [Bou86]. Les analyseurs engendrés travaillent parallèlement. A chaque fois que l'analyseur syntaxique a besoin d'une unité lexicale, il appelle l'analyseur lexical qui la lui fournit. En cas d'erreur, YACC offre un mécanisme de récupération d'erreurs qui permet la poursuite de l'analyse.

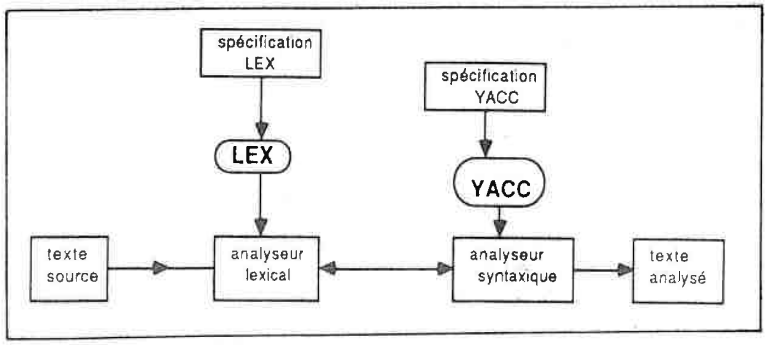


Figure 3.2: Interface de LEX avec YACC

2.3 Construction de l'arbre de syntaxe abstraite

L'arbre syntaxique abstrait d'un programme est une forme de représentation simplifiée de ce programme ; seule l'information sémantique essentielle étant représentée. Dans un tel arbre, les symboles inessentiels tels que les signes de ponctuation n'existent pas, de même que les noeuds qui ne servent qu'à l'identification de la structure du programme [CGV80].

Pour la grammaire LPSI, l'arbre est construit de manière ascendante au fur et à mesure du déroulement du processus de l'analyse syntaxique par l'intermédiaire des actions asociées aux productions. La fonction de construction admet comme arguments les sous-arbres associés aux différents symboles de la partie droite d'une règle. Elle renvoie un arbre qui sera associé au non terminal de la partie gauche pour être transmis à la suite du processus.

Dans l'arbre, chaque noeud est caractérisé par son nom et son type. Il existe cinq types de noeuds :

- des noeuds feuilles ou terminaux construits pendant la phase d'analyse lexicale. ces noeuds peuvent être valués (identificateur, nombre...) ou non (INTEGER, GL..),
- des noeuds unaires,
- des noeuds binaires,
- des noeuds ternaires : certaines règles ont été transformées afin de limiter à trois le nombre de fils par règle,
- des noeuds liste dont le nombre de fils est variable. de tels noeuds sont construits pendant la réduction par une règle récursive.

Exemples de constructions des différents noeuds

```
exp_globale : exp_globale op exp_globale
              { $$ = fnoeud2(op, $1, $3); }
            | .....
            ;
exp_forall  : forall exp in exp_globale do traitement ';'
              {$$ = fnoeud3 (forall, $2,$4,$6);}
              ;
```

Les fonctions fnoeud2 et fnoeud3 (de même que fnoeud1) construisent respectivement des noeuds à deux et trois fils et renvoient le sous-arbre associé. Chacune d'elles est paramétrée par le nom du noeud et les sous-arbres associés aux fils.

```
support : decl_supp_lpsi ';'
          { $$ = creerliste (listsup);}
          | support decl_supp_lpsi ';'
          {$$ = insererliste($1,$2);}
          ;
```

Les fonctions creerliste et insererliste permettent la création d'un noeud liste par insertions successives des sous-arbres qui le composent. Elles renvoient la tête de la liste créée.

Les pseudo-variables $\$_i$ sont utilisées par yacc pour véhiculer l'information à travers les règles syntaxiques. $\$_i$ correspond au ième symbole de la partie droite d'une règle et \$\$ au non terminal de la partie gauche.

A la fin de l'analyse, l'arbre abstrait relatif à un programme est complètement construit. Parallèlement à ceci est élaborée une table des symboles propre aux objets de LPSI. Celle-ci évite au compilateur de parcourir l'arbre pour rechercher la déclaration des différents objets au moment de leur utilisation.

Nous donnons ici un exemple d'une partie de programme LPSI avec sa représentation interne.

ifg a(f1) > b(f2)
theng c := a(f1) - b(f2)
elseg c := a(f1);

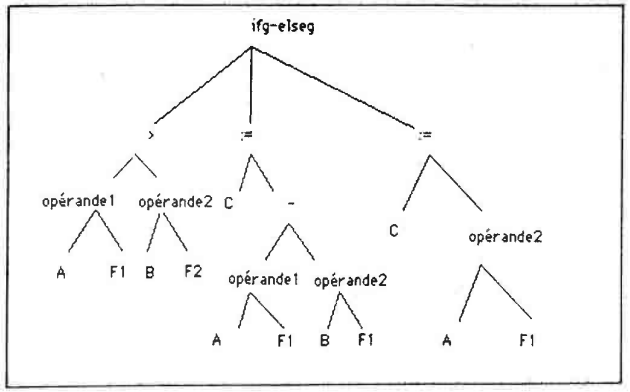


Figure 3.3: Représentation interne d'un programme

3 LE TRADUCTEUR DE LPSI

Un programme LPSI combine des structures image et des structures PASCAL. Le rôle du traducteur de LPSI est de reconnaître les structures du premier type et de les transformer de telle sorte que l'ensemble puisse être traité par le compilateur PASCAL.

Le traducteur opère à partir de l'arbre abstrait car la représentation arborescente d'un programme est fondamentalement plus facile à manipuler que sa forme textuelle. L'étiquetage des noeuds facilite la recherche de schémas précis. Nous nous intéressons dans ce cas aux objets de LPSI et aux instructions qui les utilisent. Il s'agit de trouver dans un premier temps la meilleure représentation interne de ces objets compte tenu de leur diversité et de leur richesse. Il faudra traduire par la suite de manière efficace les opérations qui les utilisent.

3.1 Traduction des types LPSI

Pour les différents types de LPSI, notre objectif est de fournir des structures internes plus ou moins "standards" afin de pouvoir homogénéiser les accès sur elles. Ces structures doivent être élaborées en fonction des considérations suivantes :

- les objets peuvent être définis dans des environnements différents (résidents, virtuels). Cette distinction est faite au moment de leur définition afin d'assurer une bonne gestion de la mémoire. Les traitements sont appliqués d'une manière identique dans les deux cas. La structure interne doit assurer une compatibilité entre ces deux types d'un même objet.
- les objets peuvent être filtrés,
- au niveau de l'association des attributs, le principal problème consiste à établir un lien entre l'objet et ses attributs. Pour tout objet, il est nécessaire de pouvoir atteindre rapidement ses attributs et vice versa.
- quand les objets image sont passés en paramètre, il est important que les procédures les acceptent quelle que soit leur taille ce qui n'est pas permis par PASCAL dont le typage est assez strict.

L'idée est de regrouper autour de la structure de base de l'objet toutes les informations qui lui sont propres de façon à ce qu'elles soient accessibles à n'importe quel endroit du programme. La structure de base d'un objet dépend de sa classe (bidimensionnelle ou linéaire).

Traduction des objets bidimensionnels

A la suite des problèmes cités, on peut considérer qu'un objet bidimensionnel se compose :

- d'une structure de base : une **table à deux dimensions**
- de structures annexes.

1. la table à deux dimensions n'est valable que dans le cas des objets résidents. Dans le cas contraire, elle est remplacée par le **descripteur** du support qui définit l'objet. Ainsi, la structure de base doit être accompagnée d'un sélecteur qui permettra de retrouver le type de l'environnement.

2. la structure annexe 1 précise les attributs implicites qui sont normalement fixés pour chaque objet.

3. la structure annexe 2 précise les liens de l'objet avec ses attributs associés. Les objets et leurs attributs communiquent par consultation d'un **indicateur commun**. L'objet repère une liste d'indicateurs (à raison d'un indicateur par attribut) tandis que l'objet attribut repère l'indicateur qui lui correspond dans la liste par l'intermédiaire d'une structure annexe 3.

4. un objet peut avoir des attributs et être lui-même attribut associé à un autre objet LPSI, la structure annexe 4 prévoit ce lien d'association.

5. enfin, la dernière structure annexe est destinée à mettre en place les mécanismes d'accès sélectif pour les objets filtrés. Autrement dit, elle repèrera la structure filtre éventuelle pour l'objet considéré.

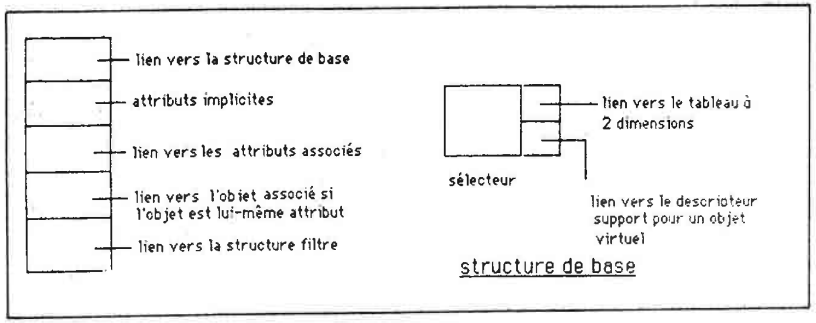


Figure 3.4: Structure interne d'un objet bidimensionnel

remarques

1. par souci d'optimisation, les différentes structures sont créées dynamiquement quand cela est nécessaire.

2. à cause des mécanismes mis en place pour l'association d'attributs, nous tenons à préciser que seuls les objets LPSI (par opposition aux objets PASCAL) peuvent être attributs.

Une réalisation possible en PASCAL


```
struct_principale   = ^environnement ;
environnement       = record
                      case selecteur : env of
                      resident :(objet_bidim_resid = ^table);
                      virtuel  :(objet_bidim_virt = ^desc_sup)
                      end;
attribut_implicite  = record
                      W : window; (* attribut cadre *)
                      S : integer; (* taille du pixel *)
                      T : enum_type; (* type de l'information : GL, INTEGER..*)
                      end;
attributs_associes  = ^liste_indicateurs;
liste_indicateurs   = record
                      indicateur: boolean;
                      indic_suiv: attributs_associes
                      end;
attribut            = ^indicateur;
objet_filtre        = ^structure_filtre;
```

Dans la suite, cette réalisation va être précisée pour les diff'rents types image bidimensionnels.

a) cas de l'image

Rien de particulier n'est à signaler dans ce cas puisque l'image est l'objet le plus représentatif de la classe bidimensionnelle. L'exemple qui suit illustre ce que nous venons de présenter :

```
var
I : image[256, 256] of gl(8) resident;
R : region[256,256] in Sup;
associate
R : segmentation(I);
```

La représentation interne de l'image I est donnée par la figure 3.5.

b) cas du binaire

Que le binaire soit utilisé en tant qu'image ou tant que filtre, La structure interne est la même. La sémantique des opérations appliquées dans les deux cas se déduit directement du

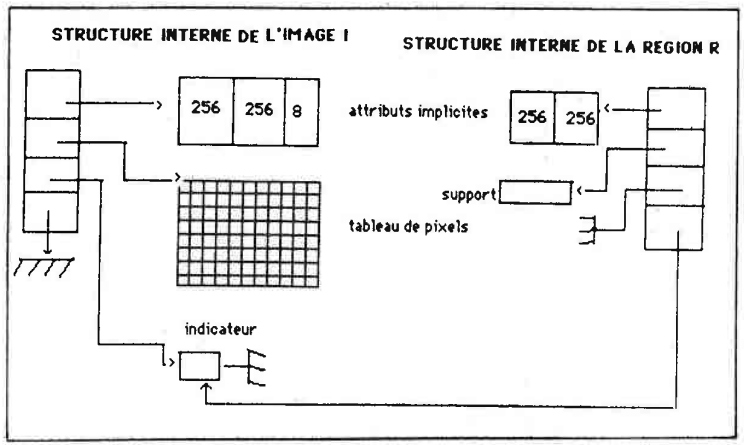


Figure 3.5: Exemple de représentation interne de l'image

contexte.

c) cas de la région

La structure principale proposée dans le cadre général est insuffisante dans le cas de la région puisque le langage doit assurer la gestion des étiquettes qui la composent. Celle-ci est donc enrichie par une **table d'indicateurs**. Chaque indicateur précise la validité (existence) de la région qui lui correspond.

d) cas du masque

La structure principale du masque est simplifiée. Le masque sera toujours résident vu ses dimensions réduites.

Traduction des objets linéaires

La structure de base prévue pour les objets linéaires est la **liste**. La nature des opérations exprimées sur eux a montré la nécessité d'une structure de donnée dynamique.

Traduction du contour

Dans le contour, la notion de liste intervient à deux niveaux, le contour étant constitué

d'une suite de sous-contours dont chacun est une suite de positions. Le rang d'un sous-contour est celui qu'il occupe dans la liste.

une réalisation possible en PASCAL

```
contour          = ^liste_scontours;
liste_scontours  = record of
                   sc       : scontour;
                   suivant  : contour
                   end;
scontour         =        ^liste_positions;
liste_positions  =        record
                   p: position;
                   suivant : scontour
                   end;
```

Mise en oeuvre de la solution

La fonction essentielle du traducteur est de parcourir l'arbre en générant du texte à chaque noeud. Dans le sous-arbre associé aux déclarations, chaque noeud (IMAGE, REGION, MASK, POSITION etc...) correspond à un type LPSI. A l'identification de tels noeuds, le traducteur opère de la manière suivante :

- chaque déclaration de type est remplacée par la déclaration PASCAL équivalente,
- chaque déclaration de variable déclenche une initialisation de la structure qui la décrit :
  1. initialisation des champs relatifs aux attributs implicites à partir des informations qui apparaissent dans la déclaration. C'est à ce niveau que sont initialisées les valeurs par défaut.
  2. initialisation du sélecteur d'environnement,

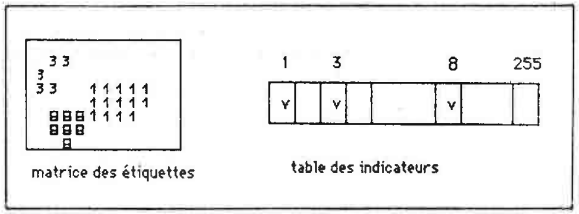


Figure 3.6: Structure de base de la région

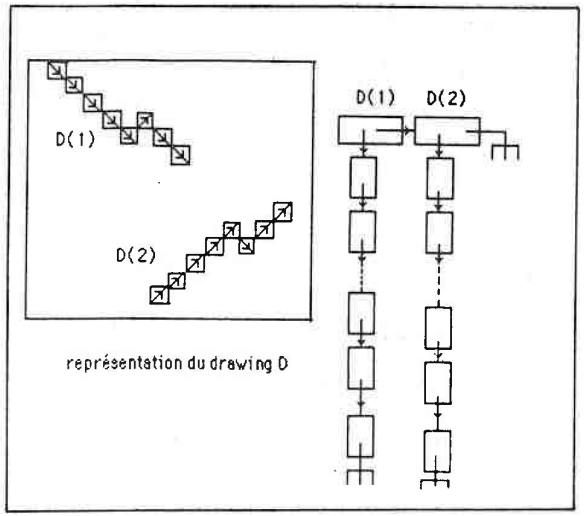


Figure 3.7: Représentation interne d'un "drawing"

3. si l'objet est résident, il faut générer sa place en mémoire centrale,
4. dans le cas contraire, on établit le lien avec le support qui le définit,
5. initialisation à vide de la structure filtre et de la structure prévue pour les liens d'association. On considère initialement que l'objet n'a pas d'attributs associés et qu'il n'est pas filtré.

3.2 Traduction du filtre

La traduction du filtre est une opération relativement délicate étant donné la complexité de ce dernier. L'idée de base consiste en l'interprétation directe des expressions logiques qui le composent au moment de l'utilisation. Ainsi l'expression

A (expression-filtre) := c;

se traduit par :

pour i de 1 à hauteur(A) faire

pour j de 1 à largeur(A) faire

si interprétation(expression-filtre) valide (i,j) alors
A[i,j] := c;
fin si

fin pour

fin pour

Cette idée simple à mettre en oeuvre n'est pas totalement satisfaisante pour deux raisons essentielles :

1. cette solution ne se généralise pas en ce sens qu'elle devient inexploitable lorsque l'objet filtré est passé en paramètre,
2. elle n'est pas performante en temps d'exécution puisqu'une même expression filtre est interprétée systématiquement à chaque utilisation.

Le premier problème nous a conduits à une forme compacte pour la transmission du filtre. L'idée du **marquage d'une image binaire** au fur et à mesure de l'interprétation de l'expression filtre s'est avérée satisfaisante à tous les points de vue.

Pour le second problème, nous donnons à l'utilisateur la possibilité de pouvoir distinguer les filtres temporaires des filtres permanents. L'expression d'un filtre temporaire sera appliquée directement à l'objet. Le filtre permanent passe par l'initialisation d'un binaire à l'expression filtre désirée qui sera traduite une seule fois mais qui peut être utilisée dans un programme autant de fois qu'il est nécessaire.

Pour homogénéiser la solution, dans tous les cas, un filtre est traduit en un binaire. Dans le cas d'un filtre temporaire, le système génère un binaire de la taille de l'objet auquel il sera appliqué. Pour éviter un encombrement de la mémoire, il est desalloué dès qu'il est utilisé. Si le filtre est permanent, il sera traduit dans le binaire prévu par l'utilisateur.

a) mise en oeuvre de la solution

1. dans les filtres à accès direct, les noms génériques (l$, v$) sont remplacés par les coordonnées des points examinés,
2. dans les filtres à accès associatif, le nom générique v$ est remplacé par la valeur du point examiné.

L'exemple suivant illustre les différentes étapes de la traduction d'un filtre temporaire utilisé sur une image I :

I(v$ ≥ 200 and l$ > 150 and c$ > 90)

1) génération d'un binaire B de la taille de I

2) initialisation de B

```
pour i de 1 à hauteur(B) faire

    pour j de 1 à largeur(B) faire
        si i > 150 et j > 90 et I[i,j] ≥200 alors
        B[i,j] := 1
        sinon B[i,j] := 0;
        fin si

    fin pour

fin pour
```

3) association du binaire à I par l'intermédiaire de la structure annexe 5
4) application de l'opération sur I filtrée
5) libération du binaire

Si le filtre est permanent, (B := I$>150 et c$ > 90), sa traduction se limite aux étapes (2, 3 et 4).

b) limite de la solution

Dans un filtre permanent, les accès associatifs sont interdits car on ne sait pas les interpréter indépendamment de l'objet sur les valeurs duquel portent les tests. C'est ce qui est prévu dans la définition du langage afin de permettre cette traduction statique.

c) constitution de la structure filtre

L'association du filtre à l'objet se fait par l'intermédiaire de la structure annexe 4 (voir section 3.1). Le binaire en lui-même est insuffisant car il faut prévoir l'information concernant son point d'application. D'autre part, comme la sélection par une fenêtre est très courante, nous avons jugé préférable de distinguer ce cas du cas général. La structure filtre complète est donc constituée par :

1. un champ pour le point de fixation du filtre,
2. un sélecteur indiquant la nature du filtre,
3. et suivant le cas, une fenêtre ou un binaire.

d) une réalisation possible en PASCAL

```
structure_filtre = record
                   point_fixe  : position;
                   case select : type_filtre of
                   fenetre : (fen : window);
                   filtre  : (filt: binaire)
                   end;
```

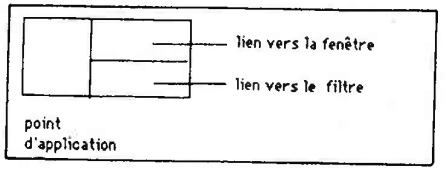



Figure 3.8: Représentation de la structure filtre

Le filtre est un bon moyen d'accéder sélectivement à un objet mais au prix d'un ralentissement de l'exécution pendant son interprétation. Pour obtenir une efficacité maximale, il est préférable d'éviter les interprétations systématiques et d'essayer d'optimiser en considérant certains facteurs. Un facteur à prendre en compte serait son point d'application. On réalise alors qu'il devient, dans certains cas, inutile de le traduire complètement. L'optimisation des filtres est un point important sur lequel nous reviendrons dans le chapitre consacré à ce sujet.

3.3 Réalisation de l'association des attributs

A chaque déclaration d'association est créé un indicateur dans la structure annexe 2 de l'objet (cf section 3.1). Un lien est établi avec l'attribut associé.

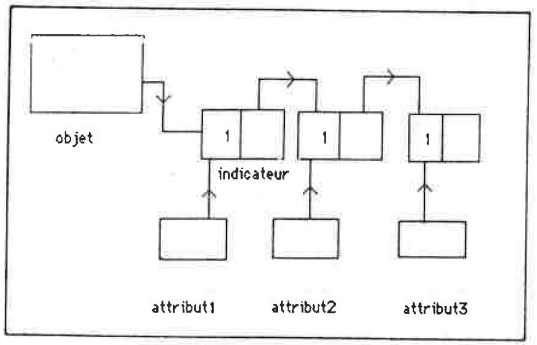


Figure 3.9: Association entre objets et attributs

Toute modification d'un objet s'accompagne d'un parcours systématique de la liste de ses indicateurs pour une mise à jour de leur état.

Toute utilisation d'un attribut entraîne systématiquement une consultation de l'indicateur qu'il repère. L'attribut ne sera recalculé que si l'objet auquel il est associé a été modifié. Un objet peut être éventuellement modifié dans deux cas :

- par lecture,
- s'il apparaît comme membre gauche d'une affectation.

Cela peut se produire dans plusieurs situations :

- l'objet est une variable interne du programme qui le définit,
- l'objet est paramètre d'un sous-programme interne au programme qui le définit,
- l'objet est paramètre d'un sous-programme compilé séparément.

Dans tous les cas, la structure interne proposée apporte une solution dans la mesure où l'objet et ses attributs restent toujours liés.

Au vu de ce que peut coûter un calcul d'attributs (histogramme par exemple), cette solution nous semble satisfaisante à tout point de vue.

3.4 Traduction d'une déclaration de support

La traduction d'un support consiste simplement à initialiser son descripteur. Un descripteur est un tableau à une dimension dans lequel sont mémorisées les caractéristiques du support. Dans le cas d'un support disque, le descripteur doit en plus être accompagné du nom physique du fichier destiné à l'objet.

3.5 Traduction des instructions de LPSI

Une fois les représentations internes pour les objets LPSI choisies, on peut définir les séquences de code pour les instructions qui leur sont associées. Il n'est pas nécessaire de produire du code en ligne pour les instructions les plus fréquentes (affectation, opérations arithmétiques et logiques, ordres de lecture/écriture), cela rallongerait considérablement la taille du code généré. Il suffit d'engendrer pour chaque action une séquence d'appels à une fonction qui la réalise. De telles fonctions sont répertoriées dans une bibliothèque disponible au moment de l'exécution.

Dans ce paragraphe nous donnons les grandes idées de résolution, tous les problèmes d'optimisation seront discutés dans le chapitre qui suit.

Traduction de l'affectation

Considérons A et B deux variables image de LPSI. La traduction de l'affectation (A:=B) entre ces deux images entraîne une série de contrôles. Certains sont gérés statiquement (tests sur l'environnement par exemple) car les informations impliquées sont connues dès la compilation. Les autres seront effectués au moment de l'exécution. La traduction d'une telle instruction passe par les étapes suivantes :

1. mise à jour des indicateurs relatifs aux attributs éventuels de A,
2. vérification de l'indicateur d'association de B car en tant qu'attribut, ce dernier doit être éventuellement calculé avant son utilisation,
3. détermination du cadre par rapport auquel s'effectue l'opération dans le cas où les objets sont de tailles différentes,
4. si A et B sont résidents, l'accès aux éléments A[i,j] et B[i,j] est direct, dans le cas contraire cela nécessite un calcul d'adresse et un accès dans leurs fichiers respectifs,
5. l'opération n'a lieu que pour les points sélectionnés simultanément dans A et B. Ce test est dynamique car il est subordonné à l'évaluation du filtre pendant la phase d'exécution.

Ces différentes étapes se résument de la manière suivante :

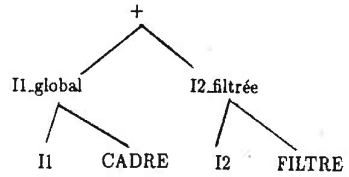
source	contrôles statiques	schéma de traduction
A(F1):= B(F2)	si possede_attribut(A) alors générer si est-attribut(B) alors générer	pour tout indicateur θ dans liste_attribut(A) faire θ := faux; fin pour si non(β) alors calcul(B); β := vraie; fin si β est l'indicateur associé à B

	x:=min(hauteur(A),hauteur(B))
	y:=min(largeur(A),largeur(B))
	
	si resident(B) alors
	générer
	sinon générer
	fin si
	si resident (A) alors
	générer
	sinon générer
	fin si
		pour i de 1 à x faire
		pour j de 1 à y faire
		si F1[i,j] et F2[i,j] alors
		
		z:= B[i,j];
		z :=fichier$_B$(i,j);
		
		A[i,j]:= z;
		fichier$_A$(i,j) := z;
		fin si
		fin pour
		fin pour

Evaluation des expressions LPSI

La représentation d'un programme sous forme d'un arbre facilite la détection des opérateurs globaux sur les images et une certaine forme d'optimisation.

Une expression est représentée par un arbre dont les noeuds sont étiquetés par des symboles d'opérations et les feuilles par des symboles de constantes ou des symboles de variables. Dans le dernier cas, il s'agit de variables image globales ou filtrées.



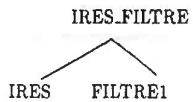
Les fonctions arithmétiques et logiques sur les images s'expriment de la manière suivante :

$$\text{op(I1,I2)} = \begin{cases} \text{1- évaluation de l'expression filtre commune à I1 et I2 :} \\ FILTRE(op(I1,I2)) = \cap(FILTRE(I1), FILTRE(I2)) \\ \\ \text{2- évaluation de l'opération op :} \\ \forall\ (i,j) \in FILTRE(op(I1,I2)), \\ valeur(op(I1,I2),i,j) = op(valeur(I1,i,j), valeur(I2,i,j)); \end{cases}$$

Si I1 est une variable globale, son filtre est ramené à son cadre.

Dans le filtre résultant de l'intersection, un point est sélectionné si les points qui lui correspondent dans les filtres initiaux son sélectionnés simultanément.

Au cours d'un parcours de l'arbre, chaque noeud opérateur de l'arbre est remplacé par un symbole de variable obtenu en appliquant les règles 1 et 2 de l'évaluation d'une opération. Les noeuds non atomiques seront traités récursivement.
Le sous-arbre précédent sera transformé en :



avec FILTRE1 = ∩ (CADRE,FILTRE)
et IRES = ADDIM (I1,I2)

Parallèlement, pour chaque opérateur est généré un appel à une fonction (par exemple ADDIM pour l'opérateur +). Cette fonction se charge de la détermination de l'image résultat et du filtre correspondant. Pour des raisons d'optimisation du temps d'exécution, l'image et

le filtre résultats sont évalués point par point. Ainsi pour l'exemple précédent, nous aurons :

texte source	schéma de traduction
I1 + I2(F)	structure_filtre(I2) := F; génération de la variable intermédiaire IRES ADDIM(I1,I2,IRES)

En réalité, l'appel de la fonction ADDIM ne se fait pas aussi simplement car il nécessite une préparation préalable des paramètres effectifs. Nous n'en dirons pas plus ici car ce point sera détaillé au moment de la discussion du passage en paramètres des objets image.

Réalisation des opérations sur les régions

Rappelons qu'une région est essentiellement constituée

- d'un tableau d'étiquettes homomorphe de l'image segmentée dans laquelle on fait correspondre à chaque point de l'image un numéro de région (étiquette de la région),
- d'une table d'indicateurs permettant de savoir si une étiquette donnée est disponible ou non.

Actuellement, on peut créer 255 régions numérotées de 1 à 255. L'étiquette 0 est réservée au système : à la déclaration d'une région, la table des étiquettes est initialisée à 0 ; par la suite, tous les points qui n'appartiennent à aucune région porteront cette étiquette.

Les opérations sur les régions sont relativement faciles à mettre en oeuvre quoiqu'elles nécessitent parfois des temps de calcul assez importants.

Le test d'appartenance d'un point à une région est immédiat : la valeur en ce point doit être identique à l'étiquette de la région. Ainsi appartient((x,y), R(i)) = (R[x,y] =i);

La création d'une région est subordonnée à la recherche dans la table des indicateurs d'une étiquette libre. Cette étiquette sera réservée par le système et son numéro affecté à tous les points de la nouvelle région. Par un accès direct dans la structure région (R[x,y]), l'utilisateur peut consulter l'étiquette affectée à un point. Les étiquettes ne sont pas nécessairement affectées dans un ordre précis.

La fusion de plusieurs régions affecte la même étiquette à l'ensemble des points des régions impliquées. Cette mise à jour nécessite le parcours total de la table des étiquettes. L'étiquette choisie est celle qui apparait en premier dans l'opération de fusion. La table des indicateurs est mise à jour afin de libérer les autres étiquettes.

La destruction d'une région est réalisée suivant le même principe. Les points qui la constituent sont libérés donc affectés de l'étiquette 0. De la même façon est libérée l'étiquette qui lui correspond dans la table des indicateurs.

Traduction des instructions de contrôle

a) cas de l'itération séquentielle

Rappelons la forme de l'itération séquentielle :

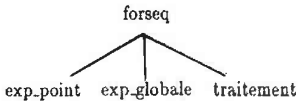
forseq exp_point **in** exp_globale **do**
traitement;

dans laquelle **exp_point** peut être une position p ou une variable générique (v$, c$, l$) qui conditionne le parcours, et **expression_globale** une expression de type image, région ou masque.

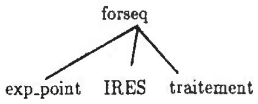
D'une manière générale, l'itération séquentielle est traduite par deux boucles imbriquées. Les éléments de contrôle de ces boucles dépendent de l'expression globale :

- dans le cas d'une variable simple, les bornes des indices d'itération sont déterminées statiquement,
- dans le cas contraire une évaluation préalable de l'expression devient nécessaire.

La séquence de code est produite à partir du sous-arbre suivant :



L'évaluation de l'expression est effectuée selon le principe discuté dans la section 3.5, et l'arbre transformé en conséquence est :



La structure intermédiaire IRES contient toutes les informations nécessaires au contrôle des deux boucles (voir exemple plus loin).

Si exp_point est la variable générique l$ (respectivement c$), le parcours se fera suivant les lignes (respectivement les colonnes). Dans le bloc "traitement", chaque occurrence de l$ (respectivement c$) sera remplacée par les coordonnées du point examiné. A chaque occurrence de v$ sera substituée la valeur du point examiné.

exemple

texte source	schéma de traduction

s := 0; n := 0;
forseq v$ in A(F1)+B(F2) do
begin
 	
s := v$+s;
n := n+1;
end;
moyenne := s/n;
	s := 0; n := 0;
	
	% évaluation de A(F1) + A(F2) %
	structure_filtre(A1) := F1;
	structure_filtre(A2) := F2;
	ADDIM(A,B,IRES);
	pour i de 1 à hauteur(IRES) do
	pour j de 1 à largeur(IRES) do
	si valide(i,j) alors
	s := s+ IRES[i,j];
	n:= n+1
	fin si
	moyenne := s/n;
	fin pour
	fin pour

remarque

Etant donné le mécanisme de traduction, il devient inutile de rappeler l'expression filtre au moment de l'accès à un point de l'expression globale. Ainsi dans l'exemple suivant :

forseq [x,y] **in** A(w1 or w2) **do** s := A(w1 or w2)[x,y] +s;

l'expression A(w1 or w2)[x,y] peut être avantageusement remplacée par A[x,y].

b) cas de l'itération parallèle

Rappelons la forme d'une telle itération :

for all exp_point **in** expression_globale **do**
traitement;

L'itération parallèle permet d'exprimer des algorithmes dans lesquels les points sont traités de manière identique indépendamment les uns des autres.

Ce type de traitement se prête bien à une implantation parallèle dans laquelle chaque point constituant une donnée locale pour un processeur est pris en charge par lui. Tous les processeurs exécutent la même action. Une machine SIMD parfaitement synchrone est tout indiquée.

Sur une machine séquentielle, l'itération parallèle se ramène à l'itération séquentielle précédemment introduite.

c) cas de la conditionnelle globale

La conditionnelle globale de LPSI est ramenée à la conditionnelle simple de PASCAL. Cette transformation est subordonnée à l'évaluation de l'expression booléenne globale. Chaque opérateur de comparaison est remplacé par un appel à une fonction booléenne. Il faut également assurer un enchaînement correct entre les séquences de code associées à la condition, à la partie "si " et à la partie "sinon".

Traduction des entrée-sorties

Les ordres d'entrée-sortie sont réalisés par deux procédures paramétrées par le descripteur du support de lecture/écriture et l'objet image à traiter. Des contrôles sont effectués en fonction de la nature du support et de ses caractéristiques :

- toute tentative d'écriture sur un support protégé échoue,
- l'utilisation de la mémoire image se fait à travers des fonctions qui assurent l'interface entre l'ordinateur hôte et le système de traitement d'images.

exemple de traduction :

source	schéma de traduction
writeg(image)support	si non protégé(support) alors cas support = disque : debut cas image = virtuelle: transfert disque->disque; residente: transfert m.c->disque; fin cas fin video : debut appel aux fonctions d'interface; cas image = virtuelle: transfert disque->video; residente: transfert m.c->video; fin cas fin fin cas fin si

Passage en paramètres des objets image

Dans le passage en paramètres, la difficulté provient du fait que les objets image ont des tailles variables, ce qui n'est pas permis par certains langages (PASCAL par exemple). De plus, dans LPSI ces objets peuvent être filtrés. A titre d'exemple, pour la procédure suivante

procedure traitement(A:t1);
où t1 est un type image défini de la manière suivante :
t1 = image of gl(8);

il faut prévoir toutes les formes d'appel qui suivent :

1. traitement(I1);
2. traitement(I1(l$+c$>30));
3. traitement(I2(W<p>));
   où I1 et I2 sont des images et W une fenêtre appliquée en p.

L'appel à la procédure "traitement" nécessite une préparation préalable des paramètres effectifs. Il est d'abord nécessaire d'interpréter l'expression filtre éventuelle dans un binaire.

Ensuite suivant le cas, le binaire ou la fenêtre est associé à la structure filtre du paramètre effectif. Ainsi la deuxième forme d'appel de "traitement"

I1(l$+c$>30)

sera traduite par

1. interprétation de la structure filtre (l$+c$ >30) dans le binaire B

2. structure_filtre(I1) := B, dans laquelle B est un binaire de la taille de I1;

Le problème de compatibilité entre paramètres formels et paramètres effectifs persiste. En effet, si le paramètre formel est un objet bidimensionnel, il est surdimensionné par le traducteur car à sa déclaration, son cadre est volontairement omis. Les paramètres effectifs ne correspondent plus en taille aux paramètres formels. Ils sont alors passés par adresse grâce à la fonction PASCAL *ADDR* [DW83]. Celle-ci est compatible avec tout type pointeur de PASCAL. Finalement l'appel

traitement(I1(l$+c$>30))

est substitué par

traitement(ADDR(I1));

A l'intérieur de la procédure, les attributs implicites renseignent sur les dimensions et l'environnement des paramètres. Il faut pouvoir assurer à chacun d'eux un traitement adéquat sachant que quatre possibilités sont à envisager (objet résident ou virtuel, filtré ou non). Une solution consiste à générer un code différent pour chacun d'eux. Cette solution n'est pas la meilleure, le nombre de cas augmentant linéairement avec le nombre de paramètres acceptables par une procédure.

La solution adoptée consiste à introduire à l'exécution des tests systématiques à chaque utilisation du paramètre. Une approche générale du code généré pour la procédure "traitement" déjà introduite est la suivante :

	contrôles introduits dans le texte généré pour "traitement"
traitement(A:t1)	pour chaque instruction qui utilise A faire cas environnement(A) = resident : si structure_filtre(A) = vide alors traitement1; sinon traitement2; fin si virtuel : si structure_filtre(A) = vide alors traitement3; sinon traitement4; fin si fin cas

De cette façon, tous les appels à la procédure "traitement" se trouvent justifiés.

4 CONCLUSION

Nous venons de présenter les principales idées d'implantation des primitives LPSI. Ceci a permis de voir tous les mécanismes introduits pour une traduction adéquate dans le langage cible : l'idée par exemple de représenter une image par une matrice de points s'est tout de suite avérée insuffisante. Il fallait une structure plus riche qui contienne toutes les caractéristiques de celle-ci à savoir :

- ses attributs implicites,
- ses liens d'association avec ses attributs,
- la nature de son environnement,
- sa structure filtre.

La structure proposée a résolu tous les problèmes induits par la complexité des objets de LPSI. En particulier, elle a rendu possible la création de fonctions indépendantes de la taille des objets et de leur environnement. Ce point est important car en traitement d'images, la manipulation de grands tableaux de pixels et la nature répétitive des traitements sur eux conduit à une approche de type fonctionnel. Cet aspect est confirmé dans les algorithmes de traduction que nous avons élaborés.

Cette phase de traduction a permis de voir de près les problèmes concernant la manipulation massive des données. Cela est apparu à travers toutes les itérations qui traduisent les opérations globales : les temps de calculs varient en conséquence. Il est donc nécessaire de

se préoccuper de l'aspect optimisation de ces traitements. En considérant certains critères, il est possible d'éliminer les calculs inutiles ou redondants. Par ailleurs, certains aspects doivent être revus en fonction des capacités de calcul de la machine cible. Il existe actuellement des machines parfaitement adaptées au traitement d'images possédant des capacités de calculs importantes.

Le chapitre suivant est destiné à l'étude de l'optimisation du code de LPSI dans le cas général puis dans le cas particulier d'une architecture spécialisée.

CHAPITRE 4

Chapitre 4

L'OPTIMISATION DANS LPSI

1 INTRODUCTION

Nous avons présenté dans le chapitre précédent des algorithmes de génération de code pour les constructions LPSI les plus importantes. Ce code a été produit sans tenir compte des situations d'optimisation et des possibilités de calcul de la machine cible. Or l'optimisation du code engendré peut conduire à un gain de performance non négligeable : le facteur de gain est estimé à 4 dans les programmes de calcul numérique [CGV80]. Le principe est de produire du code simplifié qui occupe moins de place ou qui est exécuté plus rapidement.

Certaines optimisations sont effectuées pendant la phase de génération de code en utilisant les structures de données inhérentes au compilateur. Pour cela des algorithmes d'optimisation sont en partie intégrés à la phase de génération de code. La plupart des compilateurs optimisants fonctionnent sur ce principe.

D'autres optimisations correspondent à des techniques plus avancées et conduisent à une transformation du programme source et de l'arbre abstrait. Des systèmes de transformation de programmes sont alors définis.

Le système de R. BURSTALL et J. DARLINGTON pour la transformation des programmes récursifs en programmes itératifs est le plus important dans ce domaine.

F. BELLEGARDE utilise un système de réécriture pour la suppression de séquences intermédiaires inutiles dans un programme itératif [Bel85b] dont le principe est le suivant :

Quand on construit un programme itératif structuré, on introduit des séquences intermédiaires inutiles. En utilisant la méthode de construction de programmes MEDEE [Duc84], le produit scalaire par exemple, de deux séquences <x1,x2,.....,xn> et <y1,y2,y3,....,yn> est obtenu en construisant la séquence

S1 : <x1y1,x2y2,...,xnyn>

puis la séquence

S2 : <x1y1,....,x1y1+x2y2+.....+xnyn>

dont le produit scalaire est le dernier terme. La séquence S1 est inutile.

Le système de transformation proposé est capable de supprimer de telles séquences dans des programmes itératifs. Le problème peut être résolu par des optimisations au moment de la compilation. Cette méthode ne convient malheureusement pas car son objectif est d'étudier les transformations à effectuer sur le programme source afin d'améliorer l'enseignement des méthodes de construction structurées des programmes.

Le problème reste la détermination des situations d'optimisation. Celles-ci dépendent généralement du langage à compiler et de la machine cible. On s'efforce alors de paramétrer les algorithmes d'optimisation généraux afin de les rendre les plus indépendants possible de la machine cible et du langage source.

Le traitement d'images est un domaine où l'optimisation trouve tout son intérêt et dans lequel l'architecture de la machine cible peut jouer un grand rôle. Le caractère prohibitif des temps de calcul explique l'intérêt d'augmenter les possibilités de calcul en "parallèle " conduisant ainsi à la conception de processeurs spécialisés dans l'exécution rapide d'opérations caractéristiques. On peut prétendre à un gain plus important si dans un programme, des opérations sont indépendantes, donc exécutables en même temps par plusieurs processeurs.

L'utilisation optimisée de l'architecture mise en place dépend dans une grande mesure du logiciel qui l'exploite et des ses moyens d'exprimer et de détecter le parallélisme qu'elle supporte. Cette détection peut être immédiate si le langage spécifie syntaxiquement ces situations particulières. Dans les langages que nous avons introduits au chapitre 1, des notations comme :

|| dans PascalPL,
PAR...PAREND dans PIXAL,
FOR ALL dans L,
FOR ALL et **IFG...THENG** dans LPSI,

sont utilisées pour exprimer explicitement l'indépendance des traitements sur l'ensemble des points d'un objet image. Il s'agit là d'un parallélisme typique mis en évidence par des indices syntaxiques et par l'introduction d'objets globaux et d'opérateurs pour les manipuler. Naturellement, des possibilités moins évidentes existent, que seule une analyse élaborée du programme permet d'identifier.

Dans le cas de LPSI, nous envisageons deux types d'optimisation :

1. le premier type dépend du langage et consiste à éliminer certains calculs inutiles au moment de la génération de code,

2. dans le deuxième type, les optimisations sont effectuées en fonction des capacités de calcul de la machine cible. Nous nous sommes intéressés à deux machines : la machine **ICOTECH** [DZW86] et la machine **SPHINX** [Mer83], qui présentent des degrés de parallélisme différents :

- la première est basée sur une architecture multiprocesseurs constituée de plusieurs modules, chacun d'eux est un processeur spécialisé dans l'exécution rapide d'une fonction de traitement d'images,
- la seconde a une architecture pyramidale dans laquelle des processeurs cellulaires élémentaires sont interconnectés. Elle est spécialisée dans le traitement associatif des données.

Dans le premier chapitre, une présentation générale des machines spécialisées en traitement d'images a permis de situer celles-ci dans leur contexte. Nous nous proposons dans ce chapitre, de donner leurs principales caractéristiques ainsi que l'environnement matériel et logiciel dans lequel elles évoluent.

Pour ce deuxième type d'optimisation, le pré-compilateur LPSI est enrichi par des fonctions de filtrage, qui travaillent essentiellement sur la représentation arborescente d'un programme. Ces fonctions sont systématiquement déclenchées à des endroits précis d'un programme. Se rattachant à des indices prédéfinis, elles essaient d'identifier des situations particulières. Si une telle identification réussit, une transformation locale est opérée sur le sous-arbre traité. Ce dernier est remplacé par un sous-arbre équivalent répondant à un schéma de traduction précis et faisant appel aux capacités de calcul de la machine cible. La traduction se poursuit normalement en tenant compte des transformations effectuées.

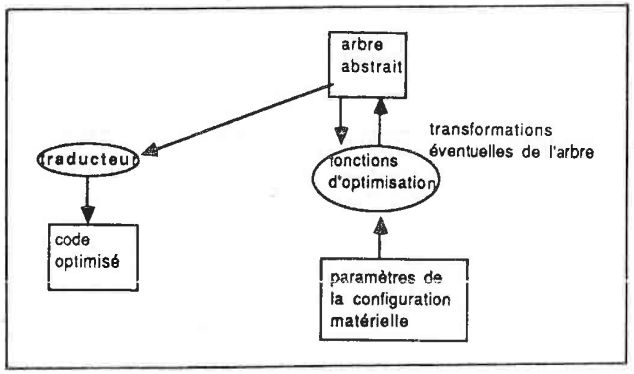


Figure 4.1: Optimiseur LPSI

2 OPTIMISATIONS INDUITES PAR LE LANGAGE

Il existe de nombreuses possibilités d'optimisation qui dépendent uniquement du langage LPSI. Elles concernent en particulier l'interprétation des filtres et l'évaluation des expressions globales. Elles sont faciles à exploiter sans gros moyens techniques pour leur mise en oeuvre.

2.1 Optimisation du filtre

Le filtre est massivement utilisé dans un programme LPSI. Compte tenu du coût de son interprétation, il est opportun de se préoccuper de son optimisation. Considérons l'expression suivante :

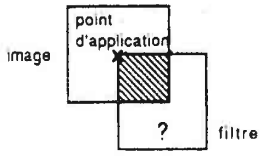
$I(I\$^2 + c\$^2 < a^2) < x, y >$

Si I est une image n × n, l'interprétation entière du filtre ($I\$^2 + c\$^2 < a^2$) nécessite 9 n^2 opérations.

Or nous constatons qu'une partie de ce dernier est inutilisée. Une optimisation possible consiste à tenir compte de son point d'application <x,y> au moment de l'interprétation.

exemple

texte source	texte généré
$I(I\$^2 + c\$^2 < a^2) < x, y >$	(* génération d'un binaire B de la taille de I *) **n:= hauteur(I)-x;** **m:= largeur(I)-y;** pour i de 1 à n faire pour j de 1 à m faire si valide(i,j) alors B[i,j] := 1 sinon B[i,j] := 0; fin si fin pour fin pour



Pour x = y = n/2, le facteur de gain est de l'ordre de 4, ce qui conduit à un gain de performance non négligeable.

Plus généralement, si l'expression combine plusieurs filtres :

I(exp-filt1<x1,y1> et exp-filt2<x2,y2> ou exp-filt3<x3,y3>.....),

un calcul préalable du point d'application <X,Y> du filtre résultat s'impose. <X,Y> est donné par :

$$X = inf(x_i) \text{ et}$$
$$Y = inf(y_i).$$

2.2 Optimisation des expressions image

Il existe de nombreuses possibilités d'optimisation des expressions image :

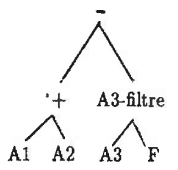
1. une première amélioration consiste à prendre en compte le cadre minimal des objets intervenant dans l'expression. En effet, vu les règles de calcul définies, il devient inutile d'évaluer l'expression en dehors de ce cadre.
   Le calcul du cadre minimal se fait au cours d'un pré-parcours du sous-arbre abstrait qui représente l'expression et permet par conséquent de figer la taille de l'image résultat.
   Ainsi pour l'expression :

   A1 - A2 + A3(F)

   où A1, A2, A3 sont des images et F un filtre, l'évaluation se fera de la manière suivante :

   min(CADRE(A1),INTER(CADRE(A3),CADRE(F)) ->
   =
   CADRE(A1)

   min(CADRE(A1),CADRE(A2) ——->
   =
   CADRE(A1)

2. la recherche d'expressions image identiques calculées à différents endroits d'un programme et la mise en facteur de ces calculs est une autre forme d'optimisation. Dans les langages universels, plusieurs algorithmes d'élimination des redondances existent. Ces algorithmes opèrent généralement à partir du graphe d'exécution d'un programme. Les techniques utilisées sont largement explicitées dans [AU77,ASU85,CGV80].

   Le problème est de déterminer les endroits du programme où ces calculs restent valables.

   Dans le cas de LPSI, une telle optimisation peut s'accompagner d'effets de bord. Considérons l'exemple suivant :

A := B + C;
D := B + C;

Si A et D sont des images de tailles différentes, l'évaluation de l'expression B + C dans la première opération (cf chapitre 2) ne fournit pas nécessairement le même résultat que la seconde.

3 OPTIMISATIONS INDUITES PAR LA MACHINE CIBLE

Ce type d'optimisation dépend de la machine cible et de sa structure interne : le nombre de registres d'une machine, l'existence d'instructions câblées en remplacement de séquences de code ayant le même effet, ont une influence certaine sur la réduction des temps d'exécution.

A un niveau plus élevé, l'existence de plusieurs unités de calcul au sein d'une même machine augmente la rapidité des traitements. En traitement d'images, l'apparition de processeurs spécialisés a rendu possible le développement à moindres coûts de machines parallèles autour d'ordinateurs conventionnels. Ces processeurs sont conçus comme un organe spécialisé associé à un calculateur hôte. L'interface est réalisé par des logiciels qui assurent une intégration cohérente de l'ensemble. Dans cette approche, la machine spécialisée est complètement déchargée des tâches ordinaires parfaitement acquitées par l'ordinateur hôte.

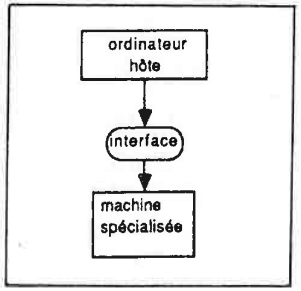


Figure 4.2: Intégration d'une machine spécialisée à un système conventionnel

Le pré-compilateur LPSI disponible sur l'ordinateur hôte se charge de détecter du code pouvant s'exécuter sur la machine spécialisée. Destiné à deux types de machines, il sera paramétré par la configuration matérielle dans laquelle il évolue.

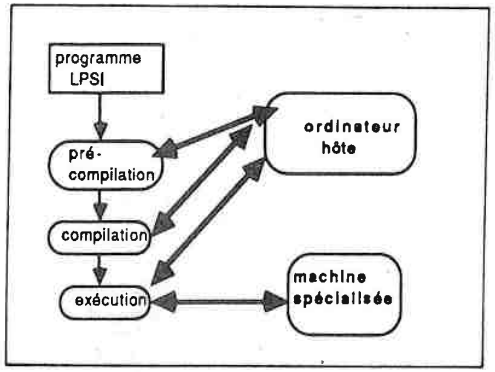


Figure 4.3: Traitement d'un programme LPSI

3.1 Cas de la machine ICOTECH

Description de la machine ICOTECH

La machine ICOTECH développée à l'Ecole Nationale Supérieure de Physique de Strasbourg évolue dans un environnement matériel et logiciel suffisamment complet pour s'adapter aux différents problèmes de traitement d'images [DZW86].

a) environnement matériel

Dans sa configuration générale, le système est modulaire et comporte des unités de traitement reconfigurables.

Le système de base est un processeur vidéo connecté à un micro-ordinateur et contenant des modules spécialisés fonctionnant en temps réel (LUT, histogramme etc...). Ce noyau constitue un poste de travail et peut fonctionner indépendamment du reste. Il évolue dans un environnement contenant des modules plus importants. L'ensemble de ces modules est organisé suivant une **architecture multiprocesseurs** gérée par un ordinateur hôte intégré au système. La partie multiprocesseurs appelée **PRIM** comporte dans sa version actuelle les processeurs spécialisés suivants :

- un processeur d'images permettant d'effectuer toute sorte d'opérations sur 2, 3 ou 4 images (+,-,*,/, opérations logiques),
- un processeur de convolution par un masque de taille variable, sur des images de 8 ou 16 bits,
- un processeur de transformée de Fourier sur des images de taille rectangulaire. La

sortie du résultat se fait sous forme complexe avec partie réelle et partie imaginaire sur 22 bits,

- un processeur de transformation géométrique permettant d'effectuer des translations en x et en y, des loupes en x et en y et des rotations.

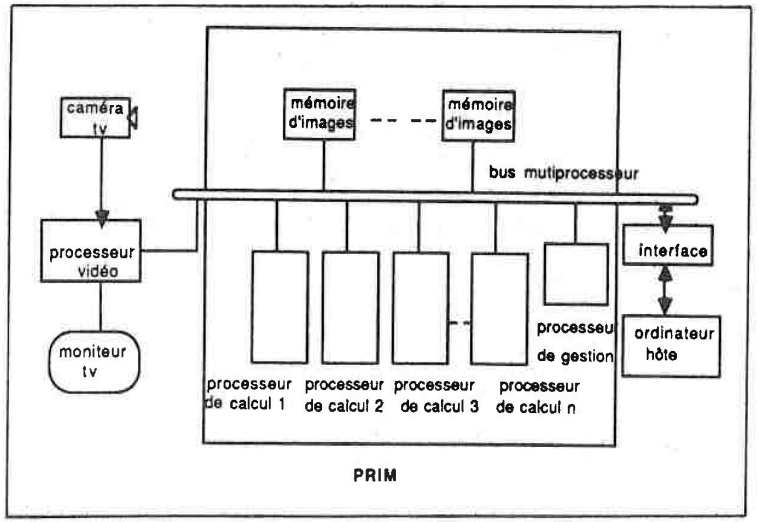


Figure 4.4: Configuration matérielle du système ICOTECH

b) environnement logiciel

L'interface du PRIM avec l'ordinateur hôte est réalisé par un logiciel appelé GESPRIM86 [TCW86]. Le multiprocesseurs reçoit des requêtes sous la forme d'une liste de fonctions et d'une liste d'arguments qu'il répartit sur les différents processeurs. Il prend en charge la gestion des ressources et la synchronisation des différentes tâches. GESPRIM86 est en interaction avec IMAGE 7 [Vog84], un logiciel de définition d'images qui lui facilite l'accès aux arguments.

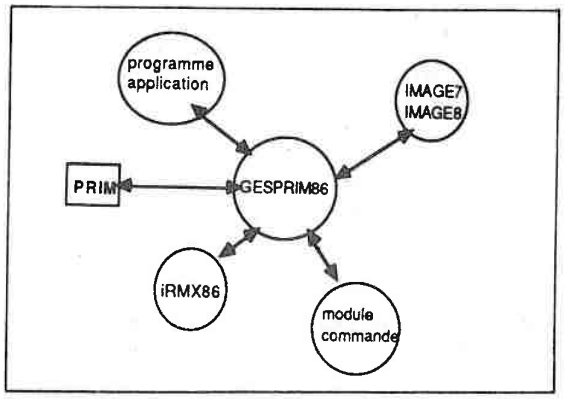


Figure 4.5: Environnement logiciel du système ICOTECH

Optimisation des opérations globales de LPSI

Nous nous proposons de détecter dans un programme LPSI des opérations pouvant s'exécuter sur les processeurs spécialisés de la machine ICOTECH. Cette détection peut être immédiate si le langage prévoit des opérateurs globaux répondant aux fonctions des processeurs spécialisés et des identificateurs globaux pour les utiliser [MB86]. C'est le cas dans LPSI des opérations arithmétiques et logiques.

En cas d'absence d'opérateurs, il faut rechercher une séquence répondant aux fonctionnalités d'un processeur existant. Cette recherche réussit si certains critères sont vérifiés. Il s'agit en particulier des combinaisons linéaires d'images et de la convolution d'une image par un masque.

Le multiprocesseurs est vu à partir d'un programme LPSI comme un ensemble de procédures de traitement et de manipulation d'images. Chaque processeur est accessible par l'intermédiaire d'une fonction dont la syntaxe d'appel est la suivante :
nom_fonction (liste-paramètres, code-erreur);

a) cas des opérations arithmétiques et logiques

La détection est relativement simple dans ce cas. Pour chaque noeud de l'arbre abstrait étiqueté par un opérateur, il suffit de vérifier la sémantique de ces noeuds fils et de générer un appel à la fonction spécialisée correspondante. Le problème principal consiste à préparer l'appel à cette fonction. Afin de décharger au maximum les fonctions d'interface, nous simplifions les objets en les ramenant dans tous les cas à un tableau de pixels.

Dans une expression, les objets manipulés peuvent avoir des tailles différentes. Un pré-parcours du sous-arbre associé à l'expression permet de déterminer le cadre minimal (cf paragraphe 2.2). Ce dernier a deux incidences sur cette étape :

- il évite les calculs inutiles,
- il permet de fixer la taille des buffers. Ces buffers sont initialisés à partir des opérandes :

1. l'opérande est une valeur numérique simple : le buffer est alors entièrement initialisé à cette valeur,
2. l'opérande est une image résidente : seule l'information nécessaire est transférée si le buffer et l'image sont de tailles différentes,
3. l'opérande est une image virtuelle : le transfert est effectué à partir du support externe associé à l'image,
4. l'opérande est filtré : il n'est pas possible de conditionner les calculs d'un processeur. Notre solution consiste à retarder l'application du filtre. Ce dernier sera évalué pour l'ensemble des opérandes et appliqué au résultat final.

La fonction d'interface se charge d'acheminer les données vers la mémoire de travail du multiprocesseurs et de restituer le résultat une fois le calcul terminé.

b) cas des combinaisons linéaires d'images

Les combinaisons linéaires retenues sont les expressions de la forme :

$$\lambda \times exp + \nu \times exp$$
$$\lambda \times exp + exp$$
$$exp + \nu \times exp$$

avec pour "exp" toute expression à résultat final une image et λ, ν deux constantes numériques.

La détection d'une combinaison linéaire est envisagée quand le noeud examiné est étiqueté par '+' et l'un des fils est l'opérateur '×'. Le tableau suivant illustre quelques cas traités. Pour une compréhension immédiate, les arbres et le code généré sont simplifiés.

expression	arbre associé	code généré
a*(I+J)+L	+ / * L / a IRES1	ADDIM(I,J,IRES1) CLINEAIRE(a,IRES1,1,L,IRES2) .
a*(I+J)+b*(K+L)	+ / * * / a IRES1 b IRES2	ADDIM(I,J,IRES1) ADDIM(K,L,IRES2) CLINEAIRE(a,IRES1,b,IRES2,IRES3)

IRES1, IRES2, IRES3 sont des images résultat.

c) cas de la convolution d'une image par un masque

La convolution d'une image I par un masque M est une opération définie de la manière suivante :

$$(I * M)(x,y) = \sum_{\mu} \sum_{\delta} I(x+i, y+j) \times M(m/2+i, n/2+j)$$

m, n sont les dimensions du masque M et "/" la division entière. Le domaine de sommation est restreint aux dimensions du masque.

La convolution consiste à sommer pour chaque fenêtre de taille m × n, n × m résultats de multiplication indépendants. Il est clair que si nous disposions de n × m processeurs de multiplication parallèles, le temps de calcul serait réduit d'autant.

Dans LPSI, il n'est pas prévu d'opérateur global pour la convolution. Afin de ne pas nous limiter à ce type d'opérations, nous avons opté pour un opérateur de produit point à point d'une sous-image par un masque grâce auquel nous exprimons naturellement toutes les opérations image × masque (convolutions, opérations de la morphologie mathématique etc...). Cela confère une plus grande puissance au langage.

Si I et C sont des images, M un masque, W une fenêtre de la taille de M, une écriture LPSI possible de cette opération serait :

```
1-  FOR ALL l$ IN I DO
2-    begin
3-    FOR ALL c$ IN I DO
4-    begin
5-      s := 0;
```

```
6-        FOR ALL v$ IN I(w<l$-m div 2, c$-n div 2>) * M DO
7-          s := s + v$;
8-        C[l$,c$] := s;
9-       end;
10-    end;
```

Cette séquence de programme exploite convenablement les possibilités du langage en exprimant l'indépendance des traitements sur les points de l'image donnée et l'indépendance des calculs pour obtenir chaque point de l'image résultat. C'est un tel profil que nous nous proposons d'identifier pour l'opération de convolution. La recherche se fonde, en premier lieu, sur des critères généraux tels que :

- l'imbrication de boucles parallèles,
- l'utilisation de l'opérateur de produit d'un masque par une fenêtre d'image de même taille.

Une analyse plus fine est ensuite requise pour les instructions 5, 6, 7 et 8 car il est possible de les exprimer de différentes manières.

Les sous-arbres (1) (2) (3) représentent la séquence LPSI donnée. La première étape du processus d'identification consiste à repérer les noeuds "*forall*" dans l'ordre d'imbrications donné.

1) identification du noeud y

L'identification du noeud y donné par la figure (3a) s'avère relativement aisée étant donné la présence éventuelle de l'opérateur "*". A ce niveau, il suffit de vérifier que :

1) ident1_filt est l'identificateur d'images donné dans les boucles "*forall*" et "$forall_1$", accessible à travers une fenêtre de la taille de ident2,

2) ident2 est un identificateur de masque.

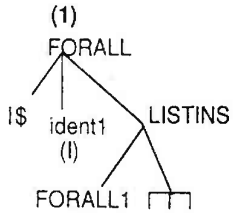
2) identification des noeuds u, z et v

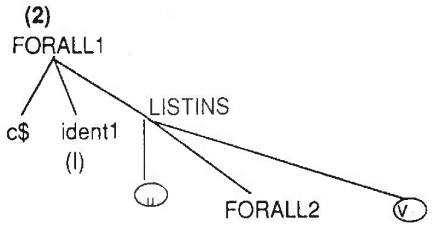
Le noeud z exprime le calcul de la somme des pondérations des points de l'image I qui peut

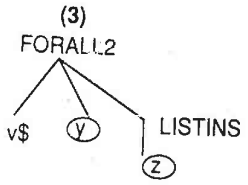
s'effectuer de plusieurs manières. Nous avons retenu deux possibilités :

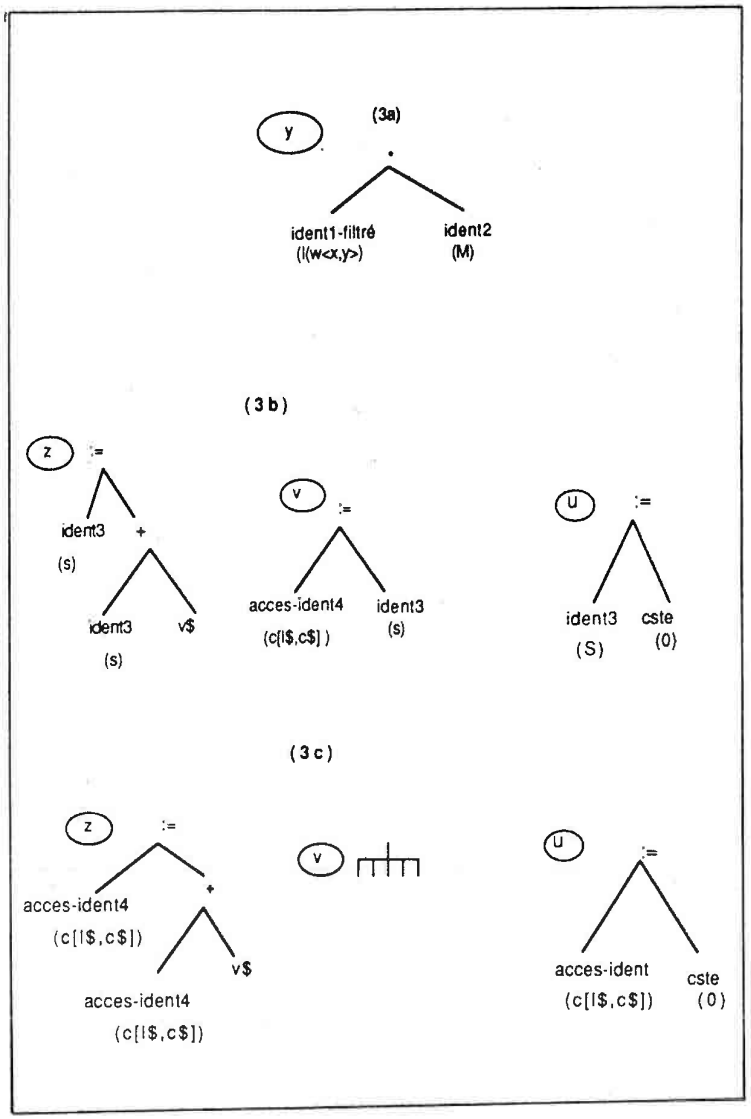
1) le calcul est effectué dans une variable intermédiaire (en l'occurrence s dans notre programme). Dans ce cas les noeuds z, u et v doivent répondre respectivement aux profils donnés par la figure (3b),

2) le calcul est directement établi dans l'image résultat. z, u et v sont alors donnés par la figure (3c).







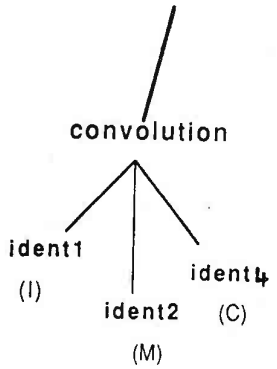


3) conséquences

Si les identifications précédentes réussissent, la transformation devient :

convolution(I,M,C)

dans laquelle convolution est un appel au convolueur spécialisé admettant comme paramètres le nom de l'image d'origine, le masque de convolution et l'image résultat. L'arbre du programme est transformé en conséquences. Il devient :



Nous nous sommes limités à l'identification d'un profil particulier pour l'opération de convolution. Cette recherche est déclenchée systématiquement à chaque passage dans l'arbre par un noeud "FORALL". Si les conditions posées sur les différents noeuds du sous-arbre (1) ne sont pas satisfaites, cette vérification échoue et la séquence est traduite dans le code de la machine hôte. Dans le cas contraire, le sous-arbre (1) est remplacé par un sous-arbre simplifié étiqueté par un appel au convolueur. Afin de rechercher toutes les convolutions, quel que soit le résultat de l'identification précédente, le processus est itéré pour chaque noeud "forall" rencontré.

3.2 Cas de la machine SPHINX

Un autre type d'optimisation peut être envisagé pour une machine fondée sur la programmation associative. Ce type de programmation est caractérisé par le fait que c'est le contenu du mot mémoire qui conditionne la transformation qu'il peut subir, et l'indépendance qui caractérise ces traitements conduit souvent à les paralléliser. Belaid dans [Bel87] donne une description complète des différents degrés de parallélisme proposés par les machines associatives existantes.

Description de la machine SPHINX

La machine SPHINX est réalisée à l'I.E.F. Elle se présente logiquement comme une pyramide de processeurs cellulaires élémentaires. Deux types de liaisons existent entre les différents processeurs : une liaison binaire entre les processeurs d'étages différents et une liaison de type 4-connexité entre ceux d'un même étage. Chaque processeur communique avec les voisins, le père et les fils . A chacun d'eux est attaché une UAL, une mémoire locale de faible capacité (64 mots de 1 bit) et un registre d'inhibition [Mer83].

Les processeurs d'un même étage exécutent d'une façon synchrone la même instruction (parallélisme SIMD). En revanche, ils exécutent des instructions différentes sur des étages différents (parallélisme Multi-SIMD).

Les mouvements de données dans la pyramide se font à partir de trois voies :

- la voie supérieure utilise la communication vers le père du dernier étage (mouvement ascendant),
- la voie inférieure assure le transfert entre la pyramide et la mémoire d'image (mouvement descendant),
- la voie horizontale met en jeu les communications entre voisins.

Les avantages d'une telle organisation sont nombreux : le parallélisme et le mode d'interconnexion assurent de bonnes performances même pour des algorithmes manipulant de grandes quantités de données [Mer83].

La machine pyramidale peut être considérée comme un organe spécialisé intégré à un calculateur hôte et programmable à partir de ce dernier. L'environnement de programmation

comprend un interpréteur de commandes qui assure le contrôle de flots d'instructions (CFI) vers les étages et vers les processeurs d'un même étage, une unité d'entrée-sortie et une unité de gestion de la mémoire d'image. Cette dernière contient les différentes images source ou résultat ainsi que les données intermédiaires.

Deux types de langages sont développés pour la machine SPHINX : le premier est un langage pyramidal de très bas niveau servant d'interface de programmation du CFI associé à la pyramide. L'autre langage est considéré comme l'assembleur de la machine permettant la programmation directe d'actions associatives. Ce dernier peut utiliser les fonctions d'interface pour programmer les mouvements de données entre les différents étages.

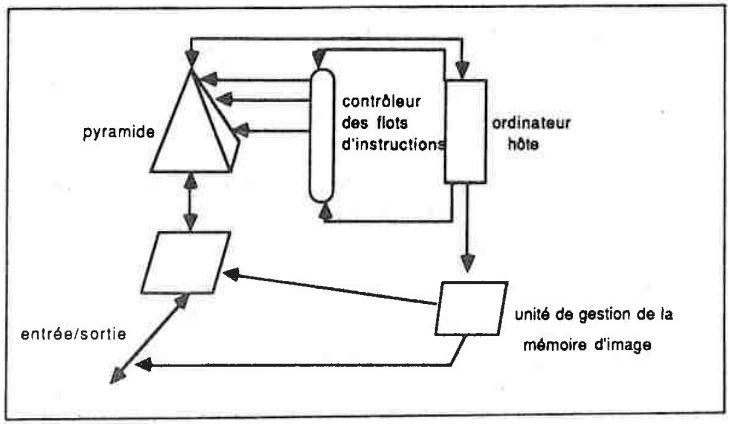
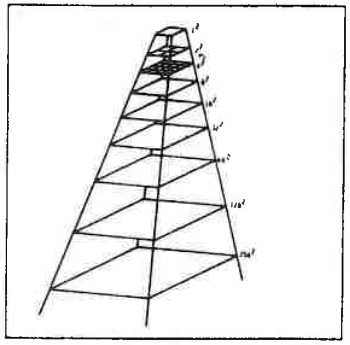


Figure 4.6: Environnement matériel de SPHINX

Optimisation des opérations de réduction

Plusieurs algorithmes de traitement d'images ont été abordés suivant une approche pyramidale [MZD84].

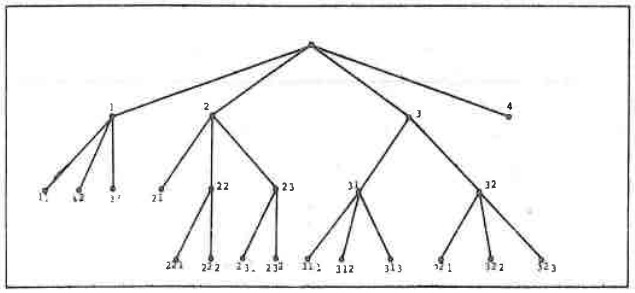
Shapiro dans [Sha79] décrit un algorithme pyramidal de détection de contours dans lequel on utilise les étages d'une pyramide virtuelle pour mémoriser une image à différents niveaux de résolution. A un niveau donné, un opérateur de détection de contours est appliqué à tous les points de l'image. Les points qui dépassent un seuil de confiance sont marqués dans la pyramide résultat et le même processus est itéré sur leurs descendants à un niveau plus fin.



Par ailleurs, Pavlidis et Horowitz [PH74] présentent un algorithme de segmentation d'images en partant d'une représentation en arbre ternaire d'une image n × n :

1. à la racine de l'arbre (niveau 0) est localisée l'image entière,
2. au niveau $l_n (l_n = log_2(n))$, chaque feuille de l'arbre représente un pixel de l'image,
3. à un niveau z quelconque ($0 < z < l_n$), chaque noeud correspond à une image localisée au point (x,y) (coordonnées du coin supérieur gauche), a des côtés de dimension $(n/2^z)$ et quatre successeurs localisés en (x,y), (x+z/2y), (x,y+z/2) et (x+z/2, y+z/2).

L'algorithme proposé utilise cette structure hiérarchique de l'image pour réaliser en deux passes (la première divise l'image, la seconde fusionne les régions similaires résultant de la première passe), sa segmentation.



Dans le cas de LPSI, toute opération globale peut s'exécuter sur une machine pyramidale. Ces opérations n'exploitent malheureusement pas toutes les ressources de l'architecture en place car elles n'utilisent qu'un seul étage de celle-ci. Or, les opérations pyramidales nécessitent généralement une décomposition en tâches élémentaires pouvant être réparties sur les différents étages de la pyramide.

Dans le cas de la machine SPHINX, nous nous sommes intéressés aux opérations de "réduction" dans lesquelles un même traitement se répète sur des données de plus en plus réduites dans une approche "diviser pour régner". Ce type d'opérations réalise à la fois :

- la division des données,
- leur traitement
- la fusion des résultats partiels obtenus.

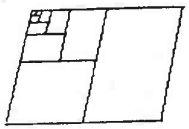
Pour programmer ce type d'opérations, nous disposons dans LPSI de structures de données complexes et d'instructions de haut niveau pour les manipuler. Malheureusement, les processeurs élémentaires de la machine SPHINX n'offrent pas un haut niveau de programmation. D'autre part, les dimensions réduites de la pyramide imposent des contraintes supplémentaires à la programmation (segmentation des données image, leur traitement partiel, problèmes de recouvrement aux frontières etc...). Pour ne pas nous restreindre aux opérations élémentaires de LPSI, nous sommes contraints de nous éloigner en quelque sorte de la structure très limitative de SPHINX.

a) <u>principe général</u>

Nous nous proposons de détecter dans un programme LPSI des profils d'algorithmes pour lesquels nous disposons d'un schéma de traduction pour la machine pyramidale. Il s'agit d'algorithmes récursifs, rappelés sur des subdivisions d'un paramètre image. Ces appels doivent être indépendants, donc exécutables parallèlement.

A cause de la nature récursive de ces algorithmes, la recherche est localisée dans les fonctions et les procédures de LPSI. Elle se fonde sur deux critères principaux :

1) la division du paramètre image est réalisée par un opérateur LPSI. Le facteur de découpage est choisi égal à 2 à cause de l'architecture en arbre binaire de la machine SPHINX. Cet opérateur utilise la forme de l'image pour déterminer le sens du découpage.



2) la procédure (ou la fonction) est activée sur chacun des deux paramètres délivrés par l'opérateur de division.

Toute procédure satisfaisant ces deux critères est réorganisée de manière à être facilement implantable sur la machine pyramidale. Sa transformation est effectuée sous certaines conditions et ne doit en aucun cas changer sa sémantique. Le schéma de traduction retenu est le suivant :

```
fonction φ(I, D) retourne R;
  .
  .
  si condition alors R := θ(I,D)
    sinon
    Φ(I,D);
    DIVISE(I,I1,I2);
    R1 := φ(I1,D1);
    R2 := φ(I2,D2);
    Ψ(R1,R2,R);
    fin si
  .
fin φ;
```

dans lequel

I : est le paramètre image de la fonction ϕ,
D : un paramètre donnée,
R : le résultat retourné par la fonction ϕ,
DIVISE : l'opérateur de division d'images,
θ : une séquence d'instructions calculant le résultat,
Φ : une séquence d'instructions traitant les données,
Ψ : une séquence d'instructions fusionnant les résultats des deux appels récursifs.

Sur la pyramide, un algorithme d'exécution possible pour cette procédure serait :

à un étage donné :

1) dans un mouvement de données descendant :
- réception de I, D
- exécution de Φ(I,D)
- transmission de I1, D1, I2, D2;

2) dans un mouvement de données ascendant :
- réception de R1,R2;
- exécution de Ψ(R1,R2);
- transmission de R;

b) <u>transformations des fonctions LPSI</u>

La transformation que nous proposons est fondée sur l'indépendance des deux appels récursifs. Cette dernière traduit la possibilité de leur exécution simultanée. Le principe est alors de regrouper ces appels en déplaçant si nécessaire les séquences LPSI intermédiaires. Ce travail, comme nous allons le montrer sur l'exemple suivant, est effectué sous certaines conditions.

Soit la fonction LPSI dans laquelle $\{c_i\}$ sont des séquences d'instructions LPSI :

```
function P(I:image; d: donnee) : resultat;
  .
  .
  if condition then P
  else
  begin
    {c1}
    divise (I,I1,I2);
    {c2}
    r1 := P(I1,d1);
    {c3}
    r2 := P(I2,d2);
    {c4}
  end;
  .
  .
```

qui doit être transformée en

```
function P(I:image; d: donnee ): resultat;
.
.
if condition then P
else
begin
  {c1}
  divise(I,I1,I2);
  Φ(d1,d2);      <——    niveau 1
  r1 := P(I1,d1);
  r2 := P(I2,d2);
  Ψ(r1,r2);      <——    niveau 2
end;
.
```

Le regroupement des appels récursifs de P nécessite le déplacement de la séquence {c3} sous les conditions suivantes :

1) {c3} ne doit contenir aucun appel récursif à P. Cette condition doit par ailleurs être vérifiée pour les séquences {c1}, {c2} et {c4}.

2) le fait que la séquence {c3} peut utiliser le résultat r1 du premier appel et calculer le paramètre d2 du second nous amène, d'une part à séparer les instructions qui calculent d2 de celles qui utilisent r1 et d'autre part, à établir leur indépendance.
Plus précisément si l'on convient de noter :

A : l'ensemble des instructions conduisant à d2,
B : l'ensemble des instructions utilisant r1,
C : les autres instructions

Si A et B sont disjoints, il devient alors possible de :

- remonter A au niveau 1. La fonction qui calcule les données devient :
  $\phi(d1,d2) = \{c2\} +^1 A$

- déplacer B au niveau 2. La fonction qui fusionne les résultats devient :
  $\Psi(r1,r2) = B + \{c4\}$

- C n'ayant aucune influence sur les deux appels peut indifféremment être déplacé dans l'un ou l'autre sens. Nous convenons de le remonter au niveau 1. Ce qui donne finalement :
  $\Phi(d1,d2) = \phi(d1,d2) + C$

Une instruction I conduit à d2 si d2 dépend d'au moins un élément de I.

Une instruction J utilise r1 si de r1 dépend au moins un élément de J.

La vérification de la disjonction des ensembles A et B est liée à l'indépendance des paramètres d2 et r1 et par voie de conséquence à celle des deux appels récursifs.

[1] le symbole "+" exprime une concaténation d'instructions

c) position du problème

Le but est de définir entre deux points du programme, plus précisément entre les deux appels récursifs, une relation de dépendance entre les objets utilisés.

Si nous établissons un ordre sur l'ensemble des relations de dépendance à construire, la borne supérieure serait la relation (la plus grossière) qui lie tous les objets entre eux. Cette solution a l'avantage d'être **calculable**. Elle est cependant trop **simplificatrice** et très peu **informative** !

La borne inférieure correspond à la relation la plus fine, dans laquelle un objet x ne dépend d'un objet y que s'il est modifié par lui. En raison de certaines situations conflictuelles, il n'est pas toujours possible de lier deux objets entre eux. En particulier, que peut-on dire de la dépendance de l'objet x dans l' exemple qui suit :

exemple

```
si p(a,b) alors
        x := f(u,v,w)
sinon
        x := g(x,y,z);
fin si
```

Un tel problème est **indécidable** car il est possible que la condition p(a,b) soit toujours fausse et si elle est vraie, la fonction f peut ne pas utiliser tous ses paramètres.

Il faut alors trouver un compromis entre les deux cas extrêmes que nous venons de décrire. Il s'agit pour cela, de construire une relation **intermédiaire** qui soit

1- informative
2- calculable

d'où la nécessité d'introduire des hypothèses de simplification raisonnables dans les parties de programme à analyser.

d) définition de la relation de dépendance

Avant de calculer la relation de dépendance, précisons d'abord certains concepts et notations utilisés.

La relation de dépendance notée R, est définie comme étant un ensemble de couples (u,v) dans lesquels u dépend de v.

Nous définissons sur R, la fonction *modif* de la manière suivante :

$$\textbf{modif(R)} = \{\textbf{u} \;/\; \exists\ \textbf{v},\ \textbf{(u,v)} \in \textbf{R}\}$$

De manière intuitive, *modif* définit pour la relation R l'ensemble des objets modifiés. Un objet est modifié s'il est :

1. membre gauche d'une instruction d'affectation
2. paramètre passé par adresse dans une fonction

Enfin, l'union de deux relations R1 et R2 est définie simplement comme étant l'union des couples qui les composent

$$R1 \cup R2 = \{(u,v) \;/\; (u,v) \in R1 \text{ ou } (u,v) \in R2\}$$

Le calcul de la relation de dépendance est effectué en deux étapes :

- nous calculons dans un premier temps, la relation de dépendance *directe* R pour les types de constructions suivants :

  1. l'affectation
  2. la conditionnelle
  3. les itérations conditionnelles
  4. les appels de procédures

  Lorsqu'une construction de programme fait intervenir une instruction non prévue ci-dessus (instruction de branchement par exemple), on prendra brutalement la relation "grossière" définie précédemment.

- la relation recherchée est *la fermeture transitive de R* (notée R^*), c'est à dire la plus petite relation réflexive et transitive qui contient R.

• cas de l'affectation

Soit α l'instruction d'affectation suivante :

x := f(x1,x2,...,xn);

La relation de dépendance directe associée à α est donnée par :

$$R_\alpha = \bigcup_i \{(x, x_i)\}$$

exemple

x := y + z; $\qquad R_\alpha = \{(x,y),(x,z)\}$;

Si x est mis à jour plusieurs fois dans la partie de programme à analyser, nous simplifions le calcul en supposant que x dépend de l'ensemble des objets qui le modifient dans toutes les instructions d'affectation considérées.
Plus généralement si S est une séquence d'instructions quelconques $\alpha_1;\alpha_2;..\alpha_n$, la relation de dépendance directe associée à S est donnée par :

$$R_S = R_{\alpha_1} \cup R_{\alpha_2} \cup .... \cup R_{\alpha_n}$$

• cas de la conditionnelle

Soit C la conditionnelle suivante :

si p(x1,x2,...,xn) **alors** α;

Tout objet x modifié dans α est également dépendant des objets x_i qui calculent la condition p. Nous obtenons alors pour R_C la formule suivante :

$$R_C = R_\alpha \cup \{(x,y)/x \in modif(R_\alpha) \text{ et } y \in \{x1, x2, ...xn\}\}$$

Généralisons ce cas en introduisant le bloc "sinon" :

si p(x1,x2,..x3) **alors** α
sinon β;

Pour les objets modifiés dans les deux blocs α et β, l'indécidabilité est résolue en élargissant la relation comme suit :

$$R_C = R_\alpha \cup R_\beta \cup \{(x,y)/x \in modif(R_\alpha \cup R_\beta) \; et \; y \in \{x1, x2, ...xn\}\}$$

exemple

si a **alors** x := x-u
sinon y:= u-v;

$R_C = \{(x,a),(x,u),(y,a),(y,u),(y,v)\}$

• cas de l'itération conditionnelle

Ce cas se déduit directement du précédent. Si I est l'itération conditionnelle suivante :

tant que p(x1,x2,...,xn) **faire**
α;

Notons R_I la relation de dépendance directe associée à I :

$$R_I = R_\alpha \cup \{(x,y)/x \in modif(R_\alpha) \; et \; y \in \{x1, x2, ..xn\}\}$$

A ce stade, nous savons construire une relation de dépendance directe associée à un morceau de programme α que nous notons $D_1(\alpha)$, la relation de dépendance recherchée étant la fermeture transitive :

$$D(\alpha) = [D_1(\alpha)]^*$$

exemple

Soit α, la séquence suivante pour laquelle nous construisons $D_1(\alpha)$ et $D(\alpha)$:

.
.

x := a * b;
tant que x > y **faire**
si x > 0 **alors**

z := z+x-y;
fin si
x := x-1;
fin tant que

$D_1(\alpha) = \{(x,a),(x,b),(z,x),(z,y)\}$
$D(\alpha) = \{(x,a),(x,b),(z,x),(z,y),(z,a),(z,b)\}$

• cas des appels de procédure

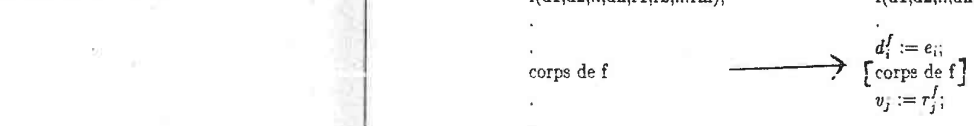
Soit f(e1,e2,...,en,v1,v2,..,vm) un appel à la procédure f, dans lequel nous supposons que ei (i = 1,..n) sont des expressions associées aux paramètres données et vj (j=1,..m) correspondent aux paramètres résultats.

La relation de dépendance associée à f dépend des variables introduites par l'appel d'une part, et de la relation induite par le corps de f (R_f) d'autre part. Pour la construire, deux cas sont à envisager suivant que la définition de f est itérative ou récursive. En effet pour le second cas, à cause du nombre potentiellement infini des appels (d'où un nombre infini de variables), il faut définir une relation de dépendance qui intègre cet aspect récursif.

• cas des appels non récursifs

La relation induite par l'appel de f est égale à la relation induite par le corps de f reécrit en introduisant les affectations suivantes :

- à l'entrée de f, toutes les expressions ei sont recopiées dans les paramètres formels qui leur correspondent,
- à la sortie de f, les résultats partiels calculés par f sont recopiés dans les paramètres effectifs qui leur correspondent.



La relation de dépendance de l'appel est alors :

$$R_{retour} \cup R_f^* \cup R_{appel}$$

dans laquelle :

R_f^* désigne la fermeture transitive de la relation de dépendance induite par le corps de f :

$$R_f^* = D[corps(f)]$$

R_{retour} est la relation induite par l'affectation du paramètre rj^f au résultat vj :

$$R_{retour} = \bigcup_{\lambda} \{R(vj := rj^f)\}$$

R_{appel} est la relation induite par l'affectation de la donnée ei au paramètre di^f

$$R_{appel} = \bigcup_{\lambda} \{R(d_i^f := e_i\}$$

Nous noterons $\psi(R_f^*, e1, e2, ...en, v1, v2, ...vm)$ cette transformation.

remarque :

Il est bien entendu nécessaire de distinguer les identificateurs des variables suivant les blocs qui les contiennent d'où la notation d_i^f et r_j^f.

exemple

Soit l'appel de procédure :

f(a+b,c,x,y)

auquel on associe la procédure suivante :

```
procedure f(d1,d2,r1,r2);
debut
      r1 := d1 + d2;
      si d2 > 1 alors r2 := d2
                sinon r2 := 0;
      fin si
fin
```


$$R_f^* = \{(r1, d1), (r1, d2), (r2, d2)\}$$

$$\begin{aligned}\psi(R_f^*, a+b, c, x, y) &= \{(x, r1), (y, r2)\} \cup \{R_f^*\} \cup \{(d1, a), (d1, b), (d2, c)\} \\ &= \{(x, a), (x, b), (x, y), (y, c)\}\end{aligned}$$

- **cas des appels récursifs**

Ce que nous venons d'établir pour les procédures non récursives n'est pas valable dans ce cas à cause du nombre infini des appels introduit qui crée un nombre infini de variables. Pour ce faire, nous allons pour chaque procédure f, construire une suite de relations Ri correspondant à la relation R_f si i appels imbriqués de f sont faits.

R_0 correspond à l'absence d'appel donc

$R_0 = \emptyset$ (aucun appel donc aucune dépendance)

R_1 correspond à la relation induite par un appel non récursif, c'est la relation construite précédemment

$R_0 \subset R_1$

Plus généralement, on peut construire R_n à partir de (R_{n-1}).
Il suffit de considérer dans le corps de f, les appels de procédures. Par construction, chacun de ces appels f(e1,e2,..,en,v1,v2,....,vm) correspond à $\psi(R_{n-1}, e1, e2, ..., en, v1, v2, .., vm)$ car nous n'autorisons qu'au plus n-1 appels imbriqués dans f.

Soit $\Psi_f(S)$, la relation de dépendance calculée sur le corps de f où chaque appel de procédure de la forme :

f(e1,e1,..en,v1,v2,..,vm)

est associé à $\psi(S, e1, e1, ...en, v1, v2, ..vm)$.

Par construction
$Rn = \Psi_f(R_{n-1})$

Ψ_f est croissante ; or c'est une relation dans un ensemble fini donc il existe une limite qui correspond au plus petit point fixe de la relation calculée. Cette limite correspond

à la relation cherchée. Elle est obtenue en dupliquant dans le programme n appels imbriqués et en remplaçant chaque appel par le corps de la procédure dans lequel les noms des variables sont changés.

exemple

Soit f, une procédure définie de la manière suivante :

```
procedure f(n,r);
debut
        .
        .
        si n = 0 alors r:= 0
                 sinon
                        f(n-1,v);
                        r := v+1;
        fin si
        .
fin
```

dont l'appel initial est : f(a+b,c).

A l'appel de f(n-1,v) est associé $\psi(R_{n-1}, n-1, v)$ calculée par récurrence.
$\Psi_f(S) = \{(n,a),(n,b),(r,n)\} \cup \psi(R_{n-1}, n-1, v) \cup \{(r,v),(c,r)\}$

remarque

Le cas du calcul de la relation induite par un appel de procédure a été abordé de manière théorique et n'a pas été implanté.

e) mise en oeuvre

Les fonctions de détection de code pour la machine pyramidale sont déclenchées aux noeuds étiquetés par une déclaration de fonction. La vérification des critères imposés est effectuée au cours du parcours. Si la vérification des conditions posées réussit, un sous-arbre répondant à la transformation est construit. Il sera greffé à la place du noeud examiné. Par ailleurs, tout appel à cette fonction dans le corps du programme est remplacé par un appel à une fonction d'interface paramétrée par le nom de la fonction, la liste de ses paramètres "donnée", et celle de ses paramètres "résultat" :

reduction(nom_fonc, liste-par-val, liste-par-var);

Connaissant la structure de la fonction paramètre, cette fonction d'interface se charge de la transcrire dans l'assembleur pyramidal en respectant l'algorithme d'exécution donné à la section 3.2.

Dans la version actuelle de la machine SPHINX, la liste des paramètres doit être limitée à un seul paramètre image, éventuellement filtré. L'association d'un registre inhibiteur à chaque processeur élémentaire permet de conditionner les accès à ce dernier, ce qui n'était pas possible pour la machine ICOTECH.

f) exemple

L'exemple suivant extrait un contour filiforme continu (liste de points chaînés) à partir d'une image binaire de contours. L'extraction se fait récursivement dans chaque moitié de l'image paramètre. Les résultats partiels sont ensuite concaténés. La concaténation de deux sous-contours est effectuée s'ils présentent une extrémité commune. Dans le cas contraire, l'ensemble des sous-contours est renvoyé au niveau supérieur. Pour mieux comprendre son fonctionnement, nous commençons par donner l'algorithme général suivi de la partie du programme LPSI qui répond au profil prédéfini :

```
fonction excat(B : binaire) retourne contour:
        si B    est réduit à un point alors
                excat retourne ce point
        sinon
                diviser(B,B1,B2);
                d1 := excat(B1);
                d2 := excat(B2);
                (* concaténation des contours d1 et d2 obtenus *)
                si d1 et d2 présentent une extrémité commune alors
                excat retourne d1 concaténé avec d2
                sinon
                  (*X est une structure qui contient tous les contours
                  n'ayant pas pu être concaténés à une étape donnée*)
                   si il existe un contour y dans la structure X tel que
                   d1 (respectivement d2) et y possèdent une extrémité commune alors
                   X := X - {y};
                   excat retourne d1 (respectivement d2) concaténé avec y
                   sinon
                    X := X+ {d1}
                    excat retourne le contour vide;
                   fin si
                fin si
        fin si
fin excat
```

```
Procedure excat(B :binary ; D :drawing);
var
B1, B2 : binary;
D1, D2 , D11: drawing;
last1, last2, first1, first2 : position;
begin
   if (B.H =1) and (B.L =1) then
      if B = 1 then
      begin
      D := B (*B est reduit a un point*)
      end
      else begin
      DIVISE(B,B1,B2);
      D1 := nil;                {c2}
      excat(B1,D1);
      D2 := nil;                {c3}
      excat(B2,D2);
      (* determination du dernier element de D1 *)
      last (D1, last1);
      (* determination du premier element de D1 *)
      first (D1, first1);
      (* determination du dernier element de D2 *)
      last (D2, last2);
      (* determination du premier element de D2 *)
      first (D2, first2);
      (* concatenation *)
      if last1 = first2
      then
      D := D1 + D2
      else if last2 = first1
      then
      D := D2 + D1
      else if first1 = first2
      then (* inversion de D1 *)
      begin
      forseq p in D1 do
      D11 := p + D1;
      D := D11 + D2
      else if last1 = last2
      then (* inversion de D2 *)
      begin
      forseq p in D1 do
      D11 := p + D2;
      D := D1 + D11
      end
```



```
      else
      (* retarder la concatenation *)
      .....
      end
end;
```

4 CONCLUSION

L'optimisation des programmes est une préoccupation fondamentale dans la réalisation d'un compilateur. Elle peut conduire à un important gain de performance. Le gain obtenu est plus important dans un domaine comme le traitement d'images où les possibilités de la machine cible y contribuent pour une grande part.

Dans le cas de LPSI, un type d'optimisation lié au langage consiste à éliminer les calculs inutiles, notamment dans l'interprétation du filtre et l'évaluation des expressions image. La mise en oeuvre d'une telle optimisation pendant la phase de traduction est simple et efficace.

Un second type dépend des capacités de calcul de la machine cible. Il s'agit dans ce cas d'identifier des séquences pouvant s'exécuter sur une machine spécialisée. Cet aspect de l'optimisation est lié au parallélisme inhérent au traitement d'images. Une étude a été effectuée sur deux types de machines :

Dans le cas de la machine ICOTECH, les opérations globales de LPSI sont prises en charge par les processeurs spécialisés de celle-ci. En effet, la plupart des opérations sur l'image ont une complexité de l'ordre du nombre de ses points. Comme ce nombre est généralement grand, les temps de traitement sont rapidement prohibitifs. Ceci explique l'intérêt d'intégrer des processeurs vectoriels à des machines séquentielles : ces dernières, même si elles présentent beaucoup d'avantages, restent très peu adaptées au traitement d'images.

Il faut alors détecter dans un programme les opérations à prendre en charge par ces processeurs rapides. Cela peut être fait dans la phase de pré-compilation afin de produire du code optimisé.

C'est dans cette optique que nous avons enrichi le pré-compilateur LPSI de fonctions dans le rôle est d'identifier les séquences pouvant s'exécuter sur les processeurs spécialisés de la machine ICOTECH. Cette opération est effectuée à partir de la représentation arborescente d'un programme et conduit parfois à la modification de sa structure.

Nous pensons que cette approche contribue à l'accélération des temps de traitement. Elle est toutefois subordonnée à une bonne gestion de l'interface entre le PRIM et la machine hôte car les temps de transfert des données entre ces deux machines risquent d'être très con-

traignants. La solution consiste alors à les exécuter par la machine hôte pendant les temps de calcul du PRIM [TCW86].

Pour la machine SPHINX, nous nous sommes intéressés à un type d'algorithmes dont l'implantation se prête bien à son architecture pyramidale. Il s'agit d'opérations de réduction pour lesquelles nous avons un schéma de traduction précis. Le but est d'extraire de la structure d'un programme des séquences pouvant s'exécuter indépendamment sur les différents étages de la pyramide. Elles sont alors réorganisées sous certaines conditions afin de faciliter leur prise en charge par la machine cible.

Le calcul de dépendance introduit déborde largement du cadre de cette application et peut être utilisé pour tester des indépendances dans des morceaux de progammes quelconques.

Dans cette optique, une perspective serait d'avoir un outil général d'aide à la transformation des programmes. Cet outil doit extraire de la forme de l'arbre abstrait d'un programme LPSI, la forme de la transformation correspondante. Il s'agit selon les cas d'un arbre standard ou d'une famille de structures (en termes d'un langage d'arbres).

L'élaboration d'un tel outil peut être facilitée par l'existence de langages de manipulation d'arbres. Ces langages utilisent comme objets de base les différents constituants d'un arbre et expriment des traitements sur eux (cheminement, recherche de schémas donnés, substitutions, permutations de noeuds etc...). Un exemple est MENTOL, le langage de manipulation d'arbres du système MENTOR [Mel83]. MENTOL utilise comme structure de données l'arbre syntaxique abstrait d'un programme généré par MENTOR sur lequel il est possible de définir des fonctions de filtrage qui substituent chaque schéma donné par la transformation correspondante.

CONCLUSION

CONCLUSION

L'insuffisance des langages existants à programmer efficacement le traitement d'images nous a amenés à proposer un langage spécialisé appelé LPSI. LPSI introduit des types spécifiques au traitement d'images sur lesquels ont été définis des modes d'accès globaux conduisant à une syntaxe lisible. Il offre des opérateurs globaux et des itérateurs sur des ensembles de points permettant d'exprimer des traitements séquentiels et parallèles.

La réalisation de son pré-compilateur a permis de tester et de valider la faisabilité de nos propositions. Le pré-compilateur est écrit dans le langage C [Dax83] sur un VAX 785. Le code PASCAL généré s'exécute sur une SM90 où est installé le système de traitement d'images. Pour la visualisation des résultats des programmes LPSI, nous utilisons le logiciel de base de ce système. A cet effet, un interface a été réalisé afin de permettre l'utilisation des fonctions de ce logiciel par le programme généré.

Les fonctions susceptibles d'être prises en charge par des processeurs spécialisés sont simulées par logiciel. Cette solution ne permet malheureusement pas d'évaluer l'apport réel d'une machine spécialisée et les temps de traitement restent lents pour les programmes testés. Nous restons néanmoins convaincus que le gain serait appréciable sur un système réel à travers un bon interface entre la machine spécialisée et l'ordinateur hôte.

Afin de développer un outil essentiellement portable sur différentes machines spécialisées, il est nécessaire de paramétrer le pré-compilateur par la configuration matérielle dans laquelle il évolue, solution que nous avons évoquée dans le chapitre 4.

Pour son haut niveau de structuration et ses possibilités d'allocation dynamique de la mémoire, le langage PASCAL, malgré son typage strict s'est révélé être un bon support pour LPSI.
Une expérience similaire a été tentée avec le langage ADA en tenant compte de ses propriétés de généricité pour implanter les types abstraits image définis [Bel87].

Le logiciel réalisé est un prototype sur lequel beaucoup d'améliorations restent possibles. En effet la nécessité de mettre en place un outil puissant et général est tellement grande qu'il

est impératif de continuer à améliorer :

- il serait intéressant de passer sur un système réel afin de pouvoir évaluer concrètement les performances envisagées ce qui permettra de réaliser l'interface entre les deux machines.
- il nous semble essentiel d'étudier des méthodes de transformations automatiques de programmes afin d'élargir les schémas de traduction que nous avions prédéfinis dans le cas de la machine SPHINX. Nous pourrons alors exploiter au mieux toutes les possibilités qu'elle offre.

ANNEXES

ANNEXES

1 EXEMPLES

Nous présentons ici quelques exemples commentés de programmes LPSI ainsi que les codes PASCAL correspondants.

1.1 EXEMPLE 1

Le programme LPSI qui suit réalise une extension d'histogramme : le but de cette opération est d'améliorer la dynamique d'une image (image1) en étendant les niveaux de gris pour lesquels l'histogramme (histo) présente une valeur nulle aux autres valeurs possibles [Tom83].

```
var
lut, histo : ftab[0..255] of integer;
image1 : image[256,256] of gl(8) resident;
max , min , i, inter, x : integer;
support
(* definition d' un support disque*)
disque1 (1,0,0,0,image1);
(* definition d'un support memoire image*)
video (2,2,0,255);
begin
(* lecture de image1 a partir de disque1 *)
readg(disque1) image1;
(* visualisation de image1 sur video *)
writeg(video) image1;
histo := 0;
```

```
(* calcul de l'histogramme de image1 *)
for all v$ in image1 do
  histo[v$]  := histo[v$] +1;
max  := 0;
min  := 0;
for all x in histo do
  if histo[x] <> 0 then
  begin
    inter  := x div histo[x];
    if inter > max then max  := inter
    else if min > inter then min  := inter;
  end;
(* calcul de la table de transfert lut *)
for i  := min to max do
  lut [i]  := 255 *(i-min)div(max - min);
(* transformation de image1 a partir de lut *)
for all v$ in image1 do
  v$  := lut[v$];
(*visualisation de image1*)
writeg(video)image1;
end.
```

Ce programme est transformé en vue d'une exécution sur une machine séquentielle. les variables terminées par "_" sont des variables internes générées par le pré-compilateur. nous pouvons particulièrement remarquer l'allocation dynamique de la mémoire pour une image résidente et sa manipulation globale dans les opérations de lecture/écriture.

```
use globaux;
(* type ftab *)

type
tint0_  = record
        ch3 : li_;
        ch4 :boolean;
        ch2  :packed array[0..255] of integer;
        end;


(* type image *)


tint1_ = record
       ch1  : record (* 1er champ : attributs implicites *)
           w  :window; (* cadre de l'image *)
```



```
           ti :integer; (* nbre de bits/pixel *)
           end;
       ch2  : envi2_; (* support image *)
       ch3  : li_; (* liste d'attributs associes *)
       ch4  : boolean; (* image elle_meme attribut associe *)
       ch5  : filter_; (* image filtree passee en paramete *)
       end;
envi2_= record (* environnement de l'image *)
       case etat  : env_ of
       res  :(imr : i3_); (* image residente *)
       nres  :(imn : des_); (* image non residente *)
       end;
i3_    = packed array [1..256 ,1..256 ] of gl8 ;
var
(*** variables de travail ***)
work1_,work2_,work3_              : li_;
indl_,indh_,adresse_,valeur_      : short;
adr1_,adr2_ :                     address;
(*****************************)
lut,histo  : tint0_;
image1  : tint1_;
max,min,i,inter,x  : integer ;
(*********************************)
(*** declaration des supports ***)
disque1 ,video  : des_;
(*********************************)
begin
(*** initialisation des attributs implicites d'une image ***)
with image1 do
begin
       ch1.w.h  := 256;
       ch1.w.l  := 256;
       ch1.ti  := 8;
       ch3  := nil;
       ch4  := nil;
       ch5  := nil;
       new(ch2);
       ch2.etat  := res;
       new(ch2.imr);
end;
(*** initialisation des supports ***)
disque1.tabsup_[1]  := 1;
disque1.tabsup_[2]  := 0;
disque1.tabsup_[9]  := 0;
disque1.tabsup_[10] := 0;
```

```
strcpy(disque1.nom_, 'image1');
video.tabsup_[1]  := 2;
video.tabsup_[2]  := 2;
video.tabsup_[9]  := 0;
video.tabsup_[10] := 255;
(*********************************)
(* lecture de image1 *)
adr1_ :=addr(image1);
readg(disque1,adr1_);
(* ecriture de image1 *)
adr1_ :=addr(image1);
writeg(video,adr1_);
forindh_ := 0 to 255 do
   histo.ch2[indh_] := 0;
for indh_ := 1 to 256 do
   for indl_ := 1 to 256 do
   begin
   valeur_ := image1.ch2.imr[indh_,indl_];
   histo.ch2[valeur_] := histo.ch2[valeur_]+1;
   end;
max := 0;
min := 255;
   for x := 0 to 255 do
   if histo.ch2[x] <> 0 then
   begin
   inter := x div histo.ch2[x];
   if inter > max then max := inter
   else if min > inter then min := inter;
   end;
for i := min to max do
   lut.ch2 [i] := 255 *(i-min)div(max - min);
for indh_ := 1 to 256 do
   for indl_ := 1 to 256 do
   begin
   valeur_ := image1.ch2.imr[indh_,indl_];
   image1.ch2.imr[indh_,indl_] := lut.ch2[valeur_];
   end;
adr1_ :=addr(image1);
writeg(video,adr1_);
end.
```


1.2 EXEMPLE 2

La visualisation de l'image (image1) de l'exemple précédent à travers le filtre ($I\$ + c\$ > 50$), donnée par **writeg(video)image1(I$+c$ >50)**;

est traduite par :

```
(* GENERATION D'UN FILTRE TEMPORAIRE *)
new(filter1_);
new(filter1_.f);
new(image1.ch5);
image1.ch5.filt := fbin_;
image1.ch5.fb := filter1_;
(* interpretation du filtre *)
for indh_ := 1 to image1.ch1.w.h do
   for indl_ := 1 to image1.ch1.w.l do
   if (( indh_ + indl_ ) >50)
       then filter1_.f[indh_,indl_] :=1
       else filter1_.f[indh_,indl_] :=0;
(* INITIALISATION DU POINT D'APPLICATION DU FILTRE *)
image1.ch5.pf.x := 1;
image1.ch5.pf.y := 1;
adr1_ :=addr(image1);
writeg(video,adr1_);
(* destruction du filtre *)
release(filter1_);
image1.ch5 := nil;
```

1.3 EXEMPLE 3

Enfin ce dernier exemple réalise une incrustation d'images suivant le principe énoncé dans le chapitre précédent. Les images utilisées sont virtuelles (c'est à dire définies à partir d'un support externe).

```
var
image1 : image[256,256] of gl(8) in support1;
image2 : image[64,64] of gl(8) in support2;
support
(* supports disque pour des images virtuelles *)
support1 (1,2,0,0,maison);
support2 (1,2,0,0,arbre);
begin
image1 := image2
end.
```

Dans le code généré, nous ne présentons que les traitements spécifiques aux images virtuelles :

```
var
.
.
support1 ,support2 : des_;
**********************************)
fichiers disques contenant les images virtuelles *)
image1f_,image2f_ : file of short;
begin
(**DEBUT INSTRUCTIONS ***)
(** INITIALISATION DES ATTRIBUTS IMPLICITES D'UNE IMAGE ***)
with image1 do
begin
    ch1.w.h  := 256;
    ch1.w.l  := 256;
    ch1.ti  := 8;
    ch3  := nil;
    ch4  := nil;
    ch5  := nil;
    new(ch2);
    ch2.etat  := nres;          ch2.imn  := addr(support1);
end;
(** INITIALISATION DES ATTRIBUTS IMPLICITES D'UNE IMAGE ***)
with image2 do
begin
  ch1.w.h  := 64;
  ch1.w.l  := 64;
  ch1.ti  := 8;
  ch3  := nil;
  ch4  := nil;
  ch5  := nil;
  new(ch2);
  ch2.etat  := nres;       ch2.imn  := addr(support2);
end;
(** INITIALISATION DES SUPPORTS ***)
support1.tabsup_[1]  := 1;
support1.tabsup_[2]  := 2; (* support image virtuel  : autorise en
support1.tabsup_[9]  := 0;
support1.tabsup_[10] := 0;
strcpy(support1.nom_, 'maison');
support2.tabsup_[1]  := 1;
support2.tabsup_[2]  := 2;
```



```
support2.tabsup_[9]  := 0;
support2.tabsup_[10] := 0;
strcpy(support2.nom_, 'arbre ');
(**********************************)
adr1_  :=addr(image1);
adr2_  := addr(image2);
affim_(adr1_,adr2_);
end.
```

2 SYNTAXE DU LANGAGE LPSI

Nous donnons ici la grammaire LPSI écrite avec le formalisme de YACC. Les règles de production sont accompagnées de fonctions de construction de l'arbre abstrait. Les règles propres à PASCAL sont volontairement omises.

```
%{
#include  <stdio.h>
#include  <ctype.h>
 /* inclusion du fichier contenant les variables globales   */
#include  "globales.ext"
 /* inclusion du fichier contenant les fonctions utilisees  */
#include  "entetefonc1.c"
%}
 /* declaration de l'axiome      */
%start pgm

 /* definition de la structure representant l'arbre abstrait */
%union {    /* type de yylval                */
          struct noeud *tnoeud;
       }

    /* tokens non types   */
%token  PROGRAM TYPE IMAGE NIMAGE BINARY MASK REGION FTAB  SUPPORT
        ASSOCIATE BEGINL IFG THENG ELSEG FORALL FORSEQ READG WRITEG
        IN OF DO DIV MOD SC IE DIFF AFF BIP AND OR NOT END

    /* tokens types   */
%token <tnoeud> IDENT ENTIER REEL VAR DRAWING  REAL
                INTEGER WINDOW POSITION GL
                BIN INTERLACED RESIDENT L1 C1 V1 R1 NIL

   /* non terminaux types          */
%type <tnoeud> pgm pgm_lpsi ident_pgm part_pgm part_decl part_decl1
               part_decl2 part_decl3 decl_type type typ_pascal_lpsi
               type_pascal_lpsi type_pascal type_lpsi lpsi_predef decl_var
               var var_pascal_lpsi  type_bid type_image type_binaire
               type_region type_mask type_ftab type_nimage
               type_taille environnement decl_supp support decl_supp_lpsi
               assoc_att association decl_ass bloc_inst
               suite_inst instruction inst_lpsi_compose
               inst_lpsi_simple expression exp_simple elt_filtre
               exp_point variable variable1 val_efmt suit_elt suite_var
               suite_expression suite_constante
```

```
suite_designation designation e_s type_indice constante
constante1 suite_ident liste_rubriques listrub1 listrub2
listrub3 ijmaxr ident_sup nom_fichier
ident_fct_lpsi cadre info_mask type_info entrelacement
type_entr taille_info nbm nb_image no_sup flag nb_oct
nb_can no_can jmax imax jo io position liste_par_eff


/* précédence des opérateurs */

%left '=' '<' '>' SE IE DIFF IN
%left '+' '-' OR
%left '*' '/' MOD AND DIV
%left UMOINS     /* moins unaire   */
%left NOT
%%
 /*  regles de production  */
pgm                 :   pgm_lpsi
                        {$<tnoeud>$ = $1;
                         /* appel au décompilateur  */
                         decompile($1);}
                    ;
pgm_lpsi            :   PROGRAM   ident_pgm  part_pgm '.'
                        {$<tnoeud>$ = node2 (PGM_LPSI,$2,$3);}
                    ;
ident_pgm           :   IDENT '(' suite_ident  ')' ';'
                        {$<tnoeud>$ = node2(IDENT_PGM,$1,$3);}
                    ;
part_pgm            :   part_decl bloc_inst
                        {$<tnoeud>$ =node2(PART_PGM,$1,$2);}
                    ;
                    /*---------------------*/
                    /* partie declarations */
                    /*---------------------*/
part_decl           :   part_decl1 part_decl2   part_decl3
                        {$<tnoeud>$ = node3 (PART_DCL,$1,$2,$3);}
                    ;
part_decl1          :    decl_label decl_const decl_type
                         {$<tnoeud>$ = node3 (PART_DCL1,$1,$2,$3);}
                    ;
part_decl2          :    decl_var decl_sp assoc_att
                        {$<tnoeud>$ = node3 (PART_DCL2,$1,$2,$3);}
                    ;
part_decl3          :   decl_supp
                        {$<tnoeud>$ = $1;}
                    ;
```



```
                    /*----------------------------*/
                    /* DECLARATION DE TYPES       */
                    /*----------------------------*/
decl_type           : TYPE type
                        {$<tnoeud>$ =$2;}
                    |   {$<tnoeud>$ = listevide(LISTTPL);}
                    ;
type                :   typ_pascal_lpsi ';'
                        {$<tnoeud>$ =creerliste(LISTTPL,$1);}
                    ;
typ_pascal_lpsi     :   IDENT '=' type_pascal_lpsi
                        {$<tnoeud>$ =node2(DCL_TPL,$1,$3);}
                    ;
type_pascal_lpsi    :   IDENT
                        {$<tnoeud>$ =$1;}
                    |   type_pascal
                        {$<tnoeud>$ = $1;}
                    |   type_lpsi
                        {$<tnoeud>$ = $1;}
                    ;
                    /*----------------------------*/
                    /* declaration des types lpsi  */
                    /*----------------------------*/
type_lpsi           :   type_bid
                        {$<tnoeud>$ = $1;}
                    |   type_nimage
                        {$<tnoeud>$ = $1;}
                    |   type_ftab
                        {$<tnoeud>$ = $1;}
                    |   type_mask
                        {$<tnoeud>$ =$1;}
                    |   lpsi_predef
                        {$<tnoeud>$ = $1;}
                    ;
lpsi_predef         :   DRAWING
                        {$<tnoeud>$ =$1;}
                    |   WINDOW
                        {$<tnoeud>$ =$1;}
                    |   POSITION
                        {$<tnoeud>$ = $1;}
                    ;
                    /*----------------------------*/
                    /* declaration des variables */
                    /*----------------------------*/
decl_var            :   VAR var
                        {$<tnoeud>$ = $2;}
                    |   {$<tnoeud>$ = listevide(LISTDVAR);}
                    ;
```

```
var             : var_pascal_lpsi ';'
                  {$<tnoeud>$ = creerliste(LISTOVAR,$1);}
                | var var_pascal_lpsi ';'
                  {$<tnoeud>$ = insererliste($1,$2);}
                ;
var_pascal_lpsi : suite_ident ':' type_pascal_lpsi
                  {$<tnoeud>$ = node2 (DCL_VTPL2,$1,$3);}
                | suite_ident ':' IDENT environnement
                  {$<tnoeud>$ = node3(DCL_VTPL3,$1,$3,$4);}
                | suite_ident ':' type_bid environnement
                  {$<tnoeud>$ = node3 (DCL_VTPL3,$1,$3,$4);}
                | suite_ident ':' type_nimage entrelacement
                  {$<tnoeud>$ = node3 (DCL_VTPL3,$1,$3,$4);}
                ;
type_bid        : type_image
                  {$<tnoeud>$ =$1;}
                | type_binaire
                  {$<tnoeud>$ = $1;}
                | type_region
                  {$<tnoeud>$ = $1;}
                ;
type_image      : IMAGE cadre type_taille
                  {$<tnoeud>$ = node2(TIMAGE,$2,$3);}
                ;
type_binaire    : BINARY cadre
                  {$<tnoeud>$ = node1(TBINAIRE,$2);}
                ;
type_region     : REGION cadre
                  {$<tnoeud>$ = node1(TREGION,$2);}
                ;
type_mask       : MASK cadre info_mask
                  {$<tnoeud>$ = node2(TMASQUE,$2,$3);}
                ;

type_ftab       : FTAB '[' type_indice ']' type_info
                  {$<tnoeud>$ = node2 (TFTAB3,$3,$5);}
                | FTAB type_info
                  {$<tnoeud>$ = node1(TFTAB2,$2);}
                ;
type_nimage     : NIMAGE cadre '(' nb_image ')' type_taille
                  {$<tnoeud>$ = node3(TNIMAGE,$2,$4,$6);}
                ;
type_taille     : OF type_info taille_info
                  {$<tnoeud>$ = node2 (TTT,$2,$3);}
                | {$<tnoeud>$ = listevide(TYPETAILLE);}
                ;
cadre           : '[' expression ',' expression ']'
```



```
                  {$<tnoeud>$ = node2(CADRE,$2,$4);}
                | {$<tnoeud>$ = atome(VIO_CAD);}
                ;
type_info       : GL
                  {$<tnoeud>$ = $1;}
                | INTEGER
                  {$<tnoeud>$ = $1;}
                | REAL
                  {$<tnoeud>$ = $1;}
                ;
info_mask       : OF BIN
                  {$<tnoeud>$ = $2;}
                | OF REAL
                  {$<tnoeud>$ = $2;}
                | OF INTEGER
                  {$<tnoeud>$ = $2;}
                | {$<tnoeud>$ = listevide(TINTEGER);}
                ;
entrelacement   : INTERLACED '(' type_entr ')'
                  {$<tnoeud>$ = node1(ENTREL,$3);}
                | {$<tnoeud>$ = listevide(TENTREL);}
                ;
taille_info     : '(' expression ')'
                  {$<tnoeud>$ = $2;}
                | {$<tnoeud>$ = listevide(TINTEGER);}
                ;
nb_image        : expression
                  {$<tnoeud>$ = $1;}
                ;
environnement   : RESIDENT
                  {$<tnoeud>$ = $1;}
                | IN ident_sup '(' nbm ')'
                  {$<tnoeud>$ = node2 (ENV,$2,$4);}
                ;
type_indice     : INTEGER
                  {$<tnoeud>$ = $1;}
                | BIN
                  {$<tnoeud>$ = $1;}
                | GL
                  {$<tnoeud>$ = $1;}
                ;
                /*-----------------------------*/
                /* declaration des supports    */
                /*-----------------------------*/
decl_supp       : SUPPORT support
                  {$<tnoeud>$ = $2;}
                | {$<tnoeud>$ = listevide(LISTSUP);}
                ;
```

```
support          :  decl_supp_lpsi ';'
                    {$<tnoeud>$ = creerliste(LISTSUP,$1);}
                 |  support  decl_supp_lpsi  ';'
                    {$<tnoeud>$ = insererliste($1,$2);}
                 ;
decl_supp_lpsi   :  IDENT '(' liste_rubriques ')'
                    {$<tnoeud>$ = node2(DCL_SUP2,$1,$3);}
                 |  IDENT '(' liste_rubriques',' nom_fichier')'
                    {$<tnoeud>$ = node3(DCL_SUP3,$1,$3,$5);}
                 ;

                  /*-------------------------*/
                 /* association d"attributs */
                /*-------------------------*/
assoc_att        :  ASSOCIATE association
                    {$<tnoeud>$ = $2;}
                 |  {$<tnoeud>$ = listevide(LISTASS);}
                 ;
association      :  decl_ass
                    {$<tnoeud>$ = creerliste(LISTASS,$1);}
                 |  association  decl_ass
                    {$<tnoeud>$ = insererliste($1,$2);}
                 ;
decl_ass         :  IDENT ':' ident_fct_lpsi '(' suite_ident ')' ';'
                    {$<tnoeud>$ = node3(DCL_ASS,$1,$3,$5);}
                 ;
                  /*--------------------*/
                 /* bloc instructions */
                /*--------------------*/
bloc_inst        :  BEGINL suite_inst  END
                    {$<tnoeud>$ = $2;}
                 ;
suite_inst       :  instruction
                    {$<tnoeud>$ = creerliste(LISTINS,$1);}
                 |  suite_inst ';' instruction
                    {$<tnoeud>$ =insererliste($1,$3);}
                 ;
instruction      :  inst_lpsi_simple
                    {$<tnoeud>$ = node2(INST,$1,$2);}
                 |  inst_lpsi_compose
                    {$<tnoeud>$ = node2(INST,$1,$2);}
                 |  {$<tnoeud>$ = atome(VID_INST);}
                 ;
```



```
inst_lpsi_compose :  BEGINL suite_inst END
                       {$<tnoeud>$ = $2;}
                  |  IFG expression THENG  instruction ELSEG
                     instruction
                     {$<tnoeud>$ =node3(IFTHENG,$2,$4,$6);}
                  |  IFG expression THENG  instruction
                     {$<tnoeud>$ =node2(IFG,$2,$4);}
                  |  FORALL exp_point IN expression DO instruction
                     {$<tnoeud>$ = node3(TFORALL,$2,$4,$6);}
                  |  FORSEQ exp_point IN expression DO instruction
                     {$<tnoeud>$ = node2(TFORSEQ,$2,$4,$6);}
                  ;

inst_lpsi_simple  :  variable AFF suite_expression
                    {$<tnoeud>$ =  node2(AFF1,$1,$3);}
                  |  IDENT liste_par_eff
                     {$<tnoeud>$ = node2(INST5,$1,$2);}
                  |  READG e_s
                     {$<tnoeud>$ = node1(TREADG,$2);}
                  |  WRITEG    e_s
                     {$<tnoeud>$ = node1(TWRITEG,$2);
                  ;
expression        :  expression '=' expression
                     {$<tnoeud>$ = node2(EGAL,$1,$3);}
                  |  expression '<' expression
                     {$<tnoeud>$ = node2(INF,$1,$3);}
                  |  expression '>' expression
                     {$<tnoeud>$ = node2 (SUP,$1,$3);}
                  |  expression IE   expression
                     {$<tnoeud>$ = node2 (TIE,$1,$3);}
                  |  expression SE   expression
                     {$<tnoeud>$ = node2(TSE,$1,$3);}
                  |  expression DIFF expression
                     {$<tnoeud>$ = node2(TDIFF,$1,$3);}
                  |  expression IN   expression
                     {$<tnoeud>$ = node2(TIN,$1,$3);}
                  |  expression '+' expression
                     {$<tnoeud>$ = node2(PLUS,$1,$3);}
                  |  expression '-' expression
                     {$<tnoeud>$ = node2(MOINS, $1,$3);}
                  |  expression OR  expression
                     {$<tnoeud>$ = node2 (TOR,$1,$3);}
                  |  expression AND expression
                    {$<tnoeud>$ = node2(TAND,$1,$3);}
                  |  expression '/' expression
                     {$<tnoeud>$ = node2(RDIV,$1,$3);}
```

```
                | expression '*' expression
                  {$<tnoeud>$ = node2(MULT,$1,$3);}
                | expression DIV expression
                  {$<tnoeud>$ = node2(TDIV,$1,$3);}
                | expression MOD expression
                  {$<tnoeud>$ = node2(TMOD,$1,$3);}
                | '(' expression ')'
                  {$<tnoeud>$ = $2;}
                | NOT expression
                  {$<tnoeud>$ = node1(TNOT,$2);}
                | '-' expression %prec UMOINS
                  {$<tnoeud>$ = node1(TUMOINS,$2);}
                | exp_simple
                  {$<tnoeud>$ = $1;}
                ;
exp_simple      : variable1
                  {$<tnoeud>$ = $1;}
                | constante
                  {$<tnoeud>$ = $1;}
                | val_efrmt % expression region, masque, drawing
                          % fenetre
                  {$<tnoeud>$ = $1;}
                | elt_filtre % expression filtre
                  {$<tnoeud>$ = $1;}
                ;
elt_filtre      : L1 % l$
                  {$<tnoeud>$ = $1;}
                | C1 % c$
                  {$<tnoeud>$ = $1;}
                | V1 % v$
                  {$<tnoeud>$ = $1;}
                | P1 % v$
                  {$<tnoeud>$ = $1;}
                | intervalle ','
                  {$<tnoeud>$ = node1(TINT1,$1);}
                | intervalle ',' intervalle
                  {$<tnoeud>$ = node2(TINT,$1,$3);}
                | ',' intervalle
                  {$<tnoeud>$ = node1(TINT2,$2);}
                ;
exp_point       : position
                  {$<tnoeud>$ = $1;}
                | L1 % l$
```



```
                  {$<tnoeud>$ = $1;}
                | C1 % c$
                  {$<tnoeud>$ = $1;}
                | V1 % v$
                  {$<tnoeud>$ = $1;}
                ;
variable        : IDENT
                  {$<tnoeud>$ = $1;}
                | variable1
                  {$<tnoeud>$ = $1;}
                ;
variable1       : IDENT suite_designation
                  {$<tnoeud>$ = node2(TVAR1,$1,$2);}
                ;
val_efrmt       : '[' suit_elt']'
                  {$<tnoeud>$ = node1(TVAL,$2);}
                | '['']'
                  {$<tnoeud>$ = atome(VID_VAL);}
                | '(' suit_elt ')'
                  {$<tnoeud>$ = node1(TELT,$2);}
                ;
suit_elt        : expression BIP expression
                  {$<tnoeud>$ = node2(TBIP,$1,$3);}
                | expression ','
                  {$<tnoeud>$ = node1(TEXP1,$1);}
                | ',' expression
                  {$<tnoeud>$ = node1(TEXP2,$2);}
                | suite_expression
                  {$<tnoeud>$ = $1;}
                ;
suite_var       : variable
                  {$<tnoeud>$ = creerliste(LISTVAR,$1);}
                | suite_var ',' variable
                  {$<tnoeud>$ = insererliste($1,$3);}
                ;

suite_expression : expression
                  {$<tnoeud>$ = creerliste(LISTEXP,$1);}
                | suite_expression ',' expression
                  {$<tnoeud>$ = insererliste($1,$3);}
                ;
suite_constante : constante
                  {$<tnoeud>$ = creerliste(LISTCONST,$1);}
                | suite_constante ',' constante
                  {$<tnoeud>$ = insererliste($1,$3);}
                ;
```

```
suite_designation   :  designation
                       {$<tnoeud>$ = creerliste(LISTDES,$1);}
                    |  suite_designation designation
                       {$<tnoeud>$ = insererliste($1,$2);}
                    ;
designation         :  '[' suite_expression ']'
                       {$<tnoeud>$ = node1(DES1,$2);}
                    |  '.' IDENT % attribut
                       {$<tnoeud>$ = node1(DES2,$2);}
                    |  '<' position'>'
                       {$<tnoeud>$ = node1(DES3,$2);}
                    |  '(' suite_expression ')'
                       {$<tnoeud>$ = node1(DES4,$2);}
                    ;

e_s                 :  '('suite_ident ')' suite_var
                       {$<tnoeud>$ = node2(ES,$2,$4);}
                    ;

constante           :  IDENT
                       {$<tnoeud>$ = $1;}
                    |  constante1
                       {$<tnoeud>$ = $1;}
                    ;
constante1          :  ENTIER
                       {$<tnoeud>$ = $1;}

                    |  REEL
                       {$<tnoeud>$ = $1;}
                    |  NIL
                       {$<tnoeud>$ = $1;}
                    ;

liste_rubriques     :  listrub1 ',' listrub2 ',' listrub3
                       {$<tnoeud>$ = node3(LISTRUB,$1,$3,$5);}
                    ;
listrub1            :  no_sup ',' flag ',' nb_oct
                       {$<tnoeud>$ = node3(LISTRUB1 ,$1,$3,$5);}
                    ;
listrub2            :  nb_can ',' no_can ',' type_entr
                       {$<tnoeud>$ = node3(LISTRUB2,$1,$3,$5);}
                    ;
listrub3            :  ijmax ',' jo ',' io
                       {$<tnoeud>$ = node3(LISTRUB3,$1,$3,$5);}
                    ;
```



```
ijmax               : imax ',' jmax
                      {$<tnoeud>$ = node2(IJMAX,$1,$3);}
                    ;
ident_sup           :  IDENT
                       {$<tnoeud>$ = $1;}
                    ;
nom_fichier         :  IDENT
                       {$<tnoeud>$ = $1;}
                    ;
ident_fct_lpsi      :  IDENT
                       {$<tnoeud>$ = $1;}
                    ;
nbm                 :  expression
                       {$<tnoeud>$ = $1;}
                    ;
no_sup              :  expression
                       {$<tnoeud>$ = $1;}
                    ;
flag                :  expression
                       {$<tnoeud>$ = $1;}
                    ;
nb_oct              :  expression
                       {$<tnoeud>$ = $1;}
                    ;
nb_can              :  expression
                       {$<tnoeud>$ = $1;}
                    ;
no_can              :  expression
                       {$<tnoeud>$ = $1;}
                    ;
type_entr           :  expression
                       {$<tnoeud>$ = $1;}
                    ;
jmax                :  expression
                       {$<tnoeud>$ = $1;}
                    ;
imax                :  expression
                       {$<tnoeud>$ = $1;}
                    ;
jo                  :  expression
                       {$<tnoeud>$ = $1;}
                    ;
io                  :  expression
                       {$<tnoeud>$ = $1;}
                    ;
position            :  IDENT
                       {$<tnoeud>$ = $1;}
                    |  '['IDENT ',' IDENT ']'
                       {$<tnoeud>$ = node2(POS,$2,$4);}
                    ;
%%
```

BIBLIOGRAPHIE

Bibliographie

[ABD*82] M. Abdelmeged, A. Belaid, C. Draman, B. Keit, and P.L. Wendel. Projet icotech : système pour la conception assistée par ordinateur et pour le traitement numérique des images. In *Journées ETCA-AFCET*, Arcueil, 1982.

[ASU85] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers : Principles, Techniques and Tools*. Addition-Wesley Publishing Company, 1985.

[AU77] A.V. Aho and J.D. Ullman. *Principles of Compiler Design*. Addison-Wesley Publishing Company, 1977.

[BB87] A. Belaid and Z. Boufriche. Hierarchical specification of image data types and its use in a high level language. In *IV International Conference of Image Analysis and Processing*, Italy, 1987.

[Bel83] A. Belaid. Projet sapin: système d'aide à la programmation du traitement d'images numérisées. rapport interne, CRIN 83-R-032, 1983.

[Bel85a] A. Belaid. Un langage de traitement d'images fondé sur des concepts de type abstraits. *Technique et Science Informatique*, 4(3):309–315, 1985.

[Bel85b] F. Bellegarde. Utilisation des systèmes de réécriture d'expressions fonctionnelles comme outils de transformation de programmes itératifs. thèse de doctorat d'état en Informatique, Université de Nancy 1, 1985.

[Bel87] A. Belaid. Méthodes et outils de programmation de systèmes de traitement d'images : le projet sapin. thèse de doctorat d'état en Informatique, Institut National Polytechnique de Lorraine, 1987.

[Bou84] Z. Boufriche. Lpsi : un langage de traitement d'images. Rapport de DEA, CRIN, 1984.

[Bou86] Z. Boufriche. Lex et yacc, 2 générateurs d'analyseurs lexical et syntaxique. rapport interne, CRIN 86-R-078, 1986.

[Cas83] D. Cassignac. Sapin-ni : un système interactif pour le traitement des images numérisées. Rapport de DEA, CRIN 83-R-061, 1983.

[Cas85] S. Castan. Architectures adaptées au traitement d'images. *Technique et Science Informatique*, 4(5):431–441, 1985.

[CBL86] S. Castan, J.L. Basille, and J.Y. Latil. Structures parallèles de traitement d'images. In *premier colloque image*, pages 437–442, Biarritz, 1986.

[CGV80] P.Y. Cunin, M. Griffiths, and J. Voiron. *Comprendre la compilation.* Springer Verlag, 1980.

[Dan86] P. Danielsson. Algorithms and architecture for image processing. In *premier colloque image*, pages p461–472, Biarritz, 1986.

[Dax83] P. Dax. *Langage C.* Eyrolles, 1983.

[DL81] M.J.B Duff and S. Levialdi. *Languages and Architectures for Image Processing.* Academic Press, London, 1981.

[Duc84] A. Ducrin. *Du problème à l'algorithme.* Dunod, Paris, 1984.

[DW83] K. Dittmar and U. Weng-Beckmann. Pascal 68000 user's guide. 1983.

[DZW86] C. Draman, K. Zampieri, and P.L. Wendel. Poste de traitement d'images configurables : icotech. In *Semaine Internationale de l'Image Electronique*, pages 437–441, Nice, 1986.

[Eri82] C. Eric. Les traitements d'images. *Electronique industrielle*, 51–57, 1982.

[FG82] H. Freeman and E. Guerrieri. Chap, a transportable line-drawing software package. In *IEEE of ICPR*, pages 79–82, Munich, 1982.

[Fou81] T.J. Fountain. Clip4 : a progress report. In M.J.B Duff and S. Levialdi, editors, *Languages and Architectures for Image Processing*, pages 283–291, Academic Press, 1981.

[Fou86] T.J. Fountain. Array architectures for iconic and symbolic image processing. In *International Congress of Pattern Recognition*, pages 24–33, Paris, 1986.

[GMT85] A. Gagalowitz, S. D. Ma, and C. Tournier-Lasserve. Nouveaux modèles pour des textures homogènes et inhomogènes. In *congrès AFCET-INRIA-ADI (RFIA)*, pages 667–678, Grenoble, 1985.

[Gud81] B. Gudmundsson. Overview of the high-level languages for picap. In M.J.B Duff and S. Levialdi, editors, *Languages and Architectures for Image Processing*, pages 147–156, Academic Press, 1981.

[Gut77] J.V. Guttag. Abstract data types and the development of data structures. *ACM*, 20(6):396–409, 1977.

[HM78] R.M. Haralick and G. Minden. Kandidats : an interactive image processing system. *Computer Graphics and Image Processing*, 1–15, 1978.

[Joh70] E. G. Johnston. The pax processing system. In Academic Press, editor, *Picture processing and psychopictories*, B. S. Lipkin and A. Rosenfeld, 1970.

[Joh81] S.C. Johnson. Yacc: yet another compiler compiler. Bell Laboratories Murray Hill, 1981.

[JW80] K. Jensen and N. Wirth. *PASCAL : manuel de l'utilisateur.* Eyrolles, 1980.

[Lev82] S. Levialdi. Languages for image processing. *Computer Vision, Lecture notes n. 1*, 2:13–32, 1982.

[Lic85] A. Lichnewsky. Super-calculateurs et calcul scientifique. In *Congrès AFCET-Informatique (matériels et logiciels pour la 5ème génération)*, pages 11–17, Paris, 1985.

[LS75] M.E. Lesk and E. Schmidt. Lex: a lexical analyser generator. Bell Laboratories, 1975.

[MB86] R. Mohr and A. Belaid. Software tools for image processing. In *Semaine Internationale de l'Image Electronique*, Nice, 1986.

[Mel83] B. Melese. Mentor : l'environnement pascal. In *Les éditeurs dirigés par la syntaxe*, pages 205–236, Aussois (Savoie), 1983.

[Mer83] A. Merigot. Une architecture pyramidale d'un multi-processeur cellulaire pour le traitement d'images. thèse de 3ème cycle en électronique, Université de Paris-sud, 1983.

[Mon85] C. Mongenet. Une méthode de conception d'algorithmes systoliques, résultats théoriques et réalisation. thèse de l' Institut National Polytechnique de Lorraine en Informatique, 1985.

[MW87] O. Monga and B. Wrobel. Segmentation d'images : vers une méthodologie. *soumis à la revue Traitement du signal (TS)*, 1987.

[MZD84] A. Merigot, B. Zavidovique, and F. Devos. Pyramidal algorithms for image processing. In *Proceedings of 7th International Conference of Pattern Recognition*, 1984.

[PH74] T. Pavlidis and S.L Horowitz. Picture segmentation by a directed split-and-merge procedure. In *Proceedings 2nd International Joint Conference On Pattern Recognition*, pages 424–433, 1974.

[PR80] K. Preston-Jr and A. Rosenfeld. Image manipulative languages, a preliminary survey. *Pattern Recognition in Practice*, 1980.

[Pre81] K. Preston-Jr. Languages for parallel processing of images. In K. Preston Jr M. Onoe and A. Rosenfeld, editors, *Real-Time/Parallel Computing Image Analysis*, pages 145–155, Plenum Press, 1981.

[Ree84] A. P. Reeves. Parallel pascal : an extended pascal for parallel computers. *parallel and distributed computing*, 2(1):64–80, 1984.

[SB84] V. Serfaty-Dutron and L. Bol-Bittoum. Conception et normalisation d'un logiciel de traitement d'images. ETCA 85-R-007. 1984.

[Ser82] J. Serra. *Image Analysis and Mathematical Morphology.* Academic Press, London, 1982.

[Sha79] L.G. Shapiro. Data structures for picture processing : a survey. *Computer Graphics and Image Processing*, 162–184, 1979.

[TCW86] H. Tedjini-Bailiche, P. Colin, and P.L. Wendel. Gestion intégrée d'un système multiprocesseur pour le traitement numérique d'images. In *Semaine Internationale de l'Image Electronique*, pages 846–848, Nice, 1986.

[Tom83] K. Tombre. Essai de classification des algorithmes en traitement d'images. Rapport de DEA, CRIN 83-R-57, 1983.

[TTS*82] H. Tamura, F. Tomita, S. Sakane, N. Yokoya, K. Sakaue, and M. Kaneko. A transportable image processing software package. In *IEEE of ICPR*, pages 75–78, Munich, 1982.

[Uhr81] L. Uhr. A language for parallel processing of arrays, embedded in pascal. In M.J.B Duff and S. Levialdi, editors, *Languages and Architectures for Image Processing*, pages 53–67, Academic Press, 1981.

[Vog84] V. Vogley. Image 7. rapport interne, Ecole Nationale Supérieure de Physique de Strasbourg, 1984.

[Woo81] A. Wood. The interaction between hardware, software and algorithms. In M.J.B Duff and S. Levialdi, editors, *Languages and Architectures for Image Processing*, pages 1–11. Academic Press, 1981.

[Zub84] L. Zuber. La foire aux langages. *L'Ordinateur individuel*, (65):122–123, 1984.

institut
national
polytechnique
de lorraine

Le Président,

AUTORISATION SOUTENANCE DE THESE

—

DOCTORAT DE 3EME CYCLE

—o—

Vu le rapport établi par :

Monsieur le Professeur MOHR Roger, Professeur INPL.

Le Président de l'INSTITUT NATIONAL POLYTECHNIQUE DE LORRAINE,

autorise : Mademoiselle BOUFRICHE Zizette

à soutenir devant l'INSTITUT NATIONAL POLYTECHNIQUE DE LORRAINE, une thèse intitulée :

"Etude et implantation d'un langage de traitement d'images. Optimisation pour des architectures spécialisées"

en vue de l'obtention du titre de DOCTEUR 3EME CYCLE

SPECIALITE : "Informatique"

Fait à Vandoeuvre, le 3 Septembre 1987

Le Président de l'I.N.P.L.

INSTITUT NATIONAL POLYTECHNIQUE DE LORRAINE — NANCY

M.GANTOIS

2, avenue de la Forêt-de-Haye - Vandœuvre
B.P. 3 - 54501 Vandœuvre Cedex - Téléphone 83.57.48.48 (lignes groupées)

Résumé

A partir d'un travail de spécification formelle de types images, nous étudions dans cette thèse les aspects liés à la définition d'un langage adapté au traitement d'images, à son implantation et à l'optimisation du code généré.

- La phase de définition consiste à étendre le langage PASCAL par des entités syntaxiques. Ces entités permettent de déclarer des types image courants et de leur associer des attributs liés à l'environnement. Elles introduisent par ailleurs des structures de contrôle réalisant facilement les parcours séquentiel et parallèle dans les objets image.
- La phase d'implantation consiste à établir des schémas de traduction pour les entités syntaxiques étudiées dans la première étape. Une priorité absolue a été accordée à l'étude du passage des paramètres afin de pouvoir travailler sur des images de taille variable. Cette phase a permis d'entreprendre la réalisation du pré-compilateur du langage défini.
- Enfin la phase d'optimisation consiste à transformer le code du langage en un code plus facilement accessible par un interface de la machine spécialisée cible. L'optimisation a été étudiée pour deux types de machines pour lesquelles le parallélisme a des propriétés différentes.

Mots-clés

traitement d'images - système Sapin - extension de langage de haut niveau - détection du parallélisme - transformation de programmes - architectures spécialisées.