HAL
open science

Structures de représentation des données en ordinateur, application aux traitements graphiques

Jean-Michel Cagnat

To cite this version:

Jean-Michel Cagnat. Structures de représentation des données en ordinateur, application aux traitements graphiques. Modélisation et simulation. Université Joseph-Fourier - Grenoble I, 1971. Français. NNT : . tel-00282889

HAL Id: tel-00282889

https://theses.hal.science/tel-00282889v1

Submitted on 28 May 2008

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

N° D'ordre

THESE

Présentée à

LA FACULTE DES SCIENCES DE L'UNIVERSITE DE GRENOBLE

pour obtenir

LE GRADE DE DOCTEUR DE TROISIEME CYCLE

"Informatique"

par

Jean-Michel CAGNAT

Structures de représentation des données en ordinateur
Application aux traitements graphiques

Thèse soutenue le 19 Février 1971 devant la commission d'examen :

MM.	J. KUNTZMANN	Président
	L. BOLLIET	Examinateur
	Y. SIRET	Examinateur

LISTE DES PROFESSEURS

Doyen Honoraire : Monsieur M. MORET
Doyen : Monsieur E. BONNIER

PROFESSEURS TITULAIRES

MM.	NEEL Louis	Physique Expérimentale
	KRAVTCHENKO Julien	Mécanique Rationnelle
	CHABAUTY Claude	Calcul différentiel et intégral
	BENOIT Jean	Radioélectricité
	CHENE Marcel	Chimie Papetière
	FELICI Noël	Electrostatique
	KUNTZMANN Jean	Mathématiques Appliquées
	BARBIER Reynold	Géologie Appliquée
	SANTON Lucien	Mécanique des Fluides
	OZENDA Paul	Botanique
	FALLOT Maurice	Physique Industrielle
	KOSZUL Jean-Louis	Mathématiques
	GALVANI Octave	Mathématiques
	MOUSSA André	Chimie Nucléaire
	TRAYNARD Philippe	Chimie Générale
	SOUTIF Michel	Physique Générale
	CRAYA Antoine	Hydrodynamique
	REULOS René	Théorie des Champs
	BESSON Jean	Chimie Minérale
	AYANT Yves	Physique Approfondie
	GALLISSOT François	Mathématiques
Mlle	LUTZ Elisabeth	Mathématiques
MM.	BLANBERT Maurice	Mathématiques
	BOUCHEZ Robert	Physique Nucléaire
	LLIBOUTRY	Géophysique
	MICHEL Robert	Minéralogie et pétrographie
	BONNIER Etienne	Electrochimie et Electrométallurgie
	DESSAUX Georges	Physiologie Animale
	PILLET Emile	Physique Industrielle-Electrotechnique
	YOCCOZ Jean	Physique Nucléaire théorique
	DEBELMAS Jacques	Géologie Générale
	BERBER Robert	Mathématiques
	PAUTENET René	Electrotechnique
	MALGRANGE Bernard	Mathématiques Pures
	VAUQUOIS Bernard	Calcul Electronique
	BARJON Robert	Physique Nucléaire
	BARBIER Jean-Claude	Physique
	SILBERT Robert	Mécanique des Fluides
	BUYLE-BODIN Maurice	Electronique
	DREYFUS Bernard	Termodynamique

MM.	KLEIN Joseph	Mathématiques
	VAILLANT François	Zoologie et Hydrobiologie
	ARNAUD Paul	Chimie
	SENGEL Philippe	Zoologie
	BARNOUD Fernand	Biosynthèse de la Cellulose
	BRISSONNEAU Pierre	Physique
	GAGNAIRE Didier	Chimie Physique
Mme	KOFLER Lucie	Botanique
MM.	DEGRANGE Charles	Zoologie
	PEBAY-PEROULA Jean Claude	Physique
	RASSAT André	Chimie Systématique
	DUCROS Pierre	Cristallographie Physique
	DODU Jacques	Mécanique Appliquée I.U.T.
	ANGLES D'AURIAC Paul	Mécanique des Fluides
	LACAZE Albert	Thermodynamique
	GASTINEL Noël	Analyse Numérique
	GIRAUD Pierre	Géologie
	PERRET René	Servo-mécanisme
	PAYAN Jean Jacques	Mathématiques Pures

PROFESSEURS SANS CHAIRE

MM.	GIDON Paul	Géologie
Mme	BARBIER Marie-Jeanne	Electrochimie
Mme	SOUTIF Jeanne	Physique
MM.	COHEN Joseph	Electrotechnique
	DEPASSEL R.	Mécanique des Fluides
	GLENAT René	Chimie
	BARRA Jean	Mathématiques Appliquées
	COUMES André	Electronique
	PERRIAUX Jacques	Géologie et Minéralogie
	ROBERT André	Chimie Papetière
	BIARREZ Jean	Mécanique Physique
	BONNET Georges	Electronique
	CAUQUIS Georges	Chimie Générale
	BONNETAIN Lucien	Chimie Minérale
	DEPOMMIER Pierre	Physique Nucléaire-Génie Atomique
	HACQUES Gérard	Calcul Numérique
	POLOUJADOFF Michel	Electrotechnique
Mme	KAHANE Josette	Physique
Mme	BONNIER Jane	Chimie
MM.	VALENTIN Jacques	Physique
	REBECQ Jacques	Biologie
	DEPORTES Charles	Chimie
	SARROT-REYNAULD Jean	Géologie
	BERTRANDIAS Jean Paul	Mathématiques
	AUBERT Guy	Physique

PROFESSEURS ASSOCIES

MM.	RODRIGUES Alexandre	Mathématiques Pures
	MORITA Susumu	Physique Nucléaire
	RADHAKRISHNA	Thermodynamique

MAITRES DE CONFERENCES

MM.	LANCIA Roland	Physique Atomique
Mme	BOUCHE Liane	Mathématiques
MM.	KAHANE André	Physique Générale
	DOLIQUE Jean Michel	Electronique
	BRIERE Georges	Pyysique
	DESRE Georges	Chimie
	LAJZEHOWICZ Joseph	Physique
	LAURENT Pierre	Mathématiques Appliquées
Mme	BERTRANDIAS Françoise	Mathématiques Pures
MM.	LONGEQUEUE Jean Pierre	Physique
	SOHM Jean Claude	Electrochimie
	ZADWORNY François	Electronique
	DURAND Françis	Chimie Physique
	CARLIER Georges	Biologie Végétale
	PFISTER Jean Claude	Physique
	CHIBON Pierre	Biologie Animale
	IDELMAN Simon	Physiologie animale
	BLOCH Daniel	Electrotechnique I.P.
	MARTIN-BOUYER Michel	Chimie (C.S.U.Chambery)
	SIBILLE Robert	Construction mécanique (I.U.T.)
	BRUGEL Lucien	Energétique I.U.T.
	BOUVARD Maurice	Hydrologie
	RICHARD Lucien	Botanique
	PELMONT Jean	Physiologie animale
	BOUSSARD Jean Claude	Mathématiques Appliquées(I.P.G.)
	MOREAU René	Hydraulique I.P.G.
	ARMAND Yves	Chimie I.U.T.
	BOLLIET Louis	Informatique I.U.T.
	KUHN Gérard	Energétique I.U.T.
	PEFFEN René	Chimie I.U.T.
	GERMAIN Jean Pierre	Mécanique
	JOLY Jean René	Mathématiques Pures

Mlle	PIERY Yvette	Biologie Animale
MM.	BERNARD Alain	Mathématiques Pures
	MOHSEN Tahsin	Biologie (C.S.U.Chambery)
	CONTE René	Mesures Physiques I.U.T.
	LE JUNTER Noël	Génie Electrique Electronique I.U.T.
	LE ROY Philippe	Génie Mécanique I.U.T.
	ROMIER Guy	Technique Statistiques Quantitatives I.U.T
	VIALON Pierre	Géologie
	BENZAKEN Claude	Mathématiques Appliquées
	MAYNARD Roger	Physique
	DUSSAUD René	Mathématiques (C.S.U.Chambery)
	BELORIZKY Elie	Physique (C.S.U Chambery)
Mme	LAJZEROWICZ Jeanine	Physique (C.S.U Chambery)
M.	JULLIEN Pierre	Mathématiques Pures
Mme	RINAUDO Marguerite	Chimie
MM.	BLIMAN Samuel	E.I.E.
	BEGUIN Claude	Chimie Organique
	NEGRE Robert	I.U.T.

MAITRES DE CONFERENCES ASSOCIES

MM.	YAMADA Osamu	Physique du Solide
	NAGAO Makoto	Mathématiques Appliquées
	MAREZIO Massimo	Physique du Solide
	CHEECKE John	Thermodynamique
	BOUDOURIS Georges	Radioélectricité
	ROZMARIN Georges	Chimie Papetière

Je tiens à remercier :

Monsieur le Professeur Jean KUNTZMANN, Directeur du Service de Mathématiques Appliquées, qui a bien voulu me faire l'honneur de présider le Jury de Thèse.

Monsieur Louis BOLLIET, Professeur à l'Institut Universitaire de Technologie de GRENOBLE, qui m'a orienté vers la recherche en programmation et m'a encouragé à poursuivre ce travail.

Monsieur Yvon SIRET, Docteur Es Sciences, qui a toujours porté un intérêt sympathique à cette recherche.

Je souhaite également remercier Monsieur Olivier LECARME qui a accepté la lourde tâche de me guider tout au long de mes recherches; ce qu'il a fait avec une compétence et une patience dont je lui suis reconnaissant.

Que tous mes collègues de Laboratoire, en particulier MM. CLEEMANN et LUCAS, trouvent ici l'expression de ma reconnaissance pour les nombreux conseils qu'ils m'ont donnés.

Je dois enfin remercier les services de dactylographie et de tirage à qui je dois la réalisation matérielle de cet ouvrage.

P L A N

INTRODUCTION

Ière PARTIE : STRUCTURES DE REPRESENTATIONS DES DONNEES.

A. GENERATION D'UNE IMAGE

B. UTILISATIONS EN "TRACEURS DE COURBES".

- B.1 Ecriture directe
- B.2 Production d'une liste d'affichage par un algorithme.

C. UTILISATION EN "DESSIN ASSISTE".

- C.1 Caractéristiques principales.
- C.2 Notion de Plex
- C.3 Le système SKETCHPAD.

D. UTILISATIONS TOPOLOGIQUES.

E. INFLUENCE DU MATERIEL UTILISE.

- E.1 Terminal sans mémoire d'entretien
- E.2 Terminal avec mémoire d'entretien
- E.3 Terminal avec mémoire d'entretien et appel de sous-programmes.

2ième PARTIE : CONSTRUCTION ET MANIPULATION DES STRUCTURES.

A. ETUDE COMPARATIVE DES FACILITES OFFERTES PAR DIVERS LANGAGES DE PROGRAMMATION RELATIVEMENT AUX STRUCTURES.

A.1 Langage machine et d'assemblage.

A.2 FORTRAN, ALGOL 60 et COBOL.

A.3 Langages permettant d'utiliser des structures.

a) IPL-V

b) LISP 1.5

c) SLIP

A.4 Langages permettant de définir des structures dans un cadre pré-établi :

a) CORAL

b) ASP

c) APL

d) LEAP

A.5 Langages sans structure pré-établie :

a) SNOBOL4

b) PL/1

c) AED-0

d) L^6, *1, DSPS.

B. REALISATION ET PRESENTATION D'UN SYSTEME DE MACRO-INSTRUCTIONS.

B.1 Définition des éléments de base.

B.2 Modifications des éléments de base.

B.3 Demande dynamique d'espace en mémoire.

B.4 Référence mémoire. Notion de séquence.

B.5 Opérations élémentaires.

B.6 Opérations non élémentaires.

B.7 Exemples.

CONCLUSION

3ième PARTIE : APPLICATIONS AUX TRAITEMENTS GRAPHIQUES.

A. REPRESENTATION EN GRAPHE.

A.1 Relations entre une image et un graphe.

A.2 Caractéristiques physiques d'une figure.

A.3 Signification d'une figure.

B. SYSTEME PERMETTANT LA CONSTRUCTION SIMULTANEE D'UNE IMAGE ET DE SA REPRESENTATION INTERNE.

B.1 Description externe

a) Etat "image". Commandes.

b) Etat "structure". Commandes.

B.2 Description interne.

a) Représentation en Plex

b) Forme interne des blocs représentatifs.

c) Structure du programme.

B.3 Développements ultérieurs.

CONCLUSION

BIBLIOGRAPHIE

INTRODUCTION

Depuis quelques années, les ordinateurs pénètrent de plus en plus profondément dans les milieux les plus divers. Tout d'abord accessibles seulement à quelques spécialistes qui acceptaient de se plier à toutes leurs exigences et qui n'avaient pas de problèmes financiers, ces ordinateurs restaient d'un emploi très délicat.

Afin de pouvoir les commercialiser, il fallait rendre leur abord plus facile; d'où le développement d'une part des langages de programmation et d'autre part de systèmes d'exploitation.

Ces langages sont artificiels et présentent surtout la caractéristique de transiter par un support linéaire. Ainsi il est longtemps resté impossible de communiquer par l'intermédiaire du dessin, moyen pourtant très naturel mais qui nécessite un support à deux dimensions.

Aussi a-t-on vu avec un certain optimisme le développement de ces nouveaux dispositifs d'entrée et sortie, appelés consoles de visualisation. Pour la première fois, on disposait d'un support à deux dimensions, susceptible de permettre des communications plus poussées avec la machine. Comme de plus ces terminaux étaient silencieux et rapides, tout concourait à encourager leur emploi. Toutefois, leur prix élevé limitait leur diffusion et seuls quelques centres de recherche purent s'offrir cet équipement.

Les difficultés rencontrées ont été nombreuses et proviennent en particulier du fait qu'un dessin ne peut être conservé tel quel dans la mémoire de l'ordinateur et doit donc subir une certaine transformation. Cette transformation doit être complète, c'est-à-dire que toute l'information contenue dans le dessin doit être conservée; elle doit être réversible afin de permettre la sortie du dessin; enfin, elle doit être suffisamment simple afin que sa fabrication soit rapide, qu'elle ne nécessite pas une place en mémoire trop

importante et surtout qu'elle permette de travailler sur le codage obtenu.

D'autres problèmes se posent également liés aux nécessités de conversation, de rapidité, etc....

C'est cependant aux problèmes de représentation du dessin ou encore de structures d'information que nous allons nous intéresser dans les pages qui suivent.

Nous commencerons par une étude de diverses méthodes de représentation des données qui ont été utilisées dans des systèmes existants. Nous dégagerons ainsi quelques critères généraux.

Dans une deuxième partie, nous nous intéresserons aux outils dont on peut disposer pour construire des représentations complexes. Nous verrons se dégager de l'étude de divers langages certaines règles qui nous serviront à proposer un outil de base.

Dans la dernière partie, nous décrirons un système expérimental que nous avons développé afin de permettre d'une part de prendre conscience des problèmes posés, et d'autre part d'étudier l'intérêt de certains outils originaux.

P R E M I E R E P A R T I E

STRUCTURES

DE

REPRESENTATION DES DONNEES

PREMIERE PARTIE

STRUCTURES DE REPRESENTATIONS DES DONNEES

Cette étude sera limitée aux principales méthodes employées pour résoudre les problèmes posés par l'emploi des consoles de visualisation. Il ne s'agit cependant pas d'une limitation car ces consoles sont l'objet d'applications dans des domaines très divers.

Ces applications sont décrites plus ou moins complètement dans des revues spécialisées ou dans des rapports techniques qu'il est parfois difficile de se procurer. De plus, ces descriptions insistent en général sur le côté spectaculaire, c'est-à-dire sur l'utilisation que l'on peut en faire, et sur le profit que l'on peut en retirer. L'aspect technique est rapidement survolé et seuls quelques points de détails sont commentés.

Il nous a donc paru nécessaire de présenter les structures de représentations des données les plus fondamentales et les plus courantes. Nous les classerons par ordre de complexité croissante, en donnant les avantages et les inconvénients de chaque type.

Nous présenterons ainsi successivement les utilisations en "traceurs de courbes", les utilisations en "dessin assisté" et enfin les utilisations "topologiques". Nous montrerons également en quoi le mode d'utilisation d'une console influe sur la complexité de la structure choisie.

A. GENERATION D'UNE IMAGE.

Commençons par quelques rappels sur le fonctionnement élémentaire d'un terminal graphique.

Un faisceau d'électrons frappe la surface d'un tube cathodique. Un spot lumineux est produit par l'impact de ce faisceau sur la substance à base de phosphore qui recouvre la surface du tube. La direction du faisceau, et donc la position du spot, et son intensité sont gouvernées par quelques ordres simples, tels que :

POINTA	X,Y	allumer le point x,y.
POSA	X,Y	aller à la position x,y sans éclairer.
VECTA	X,Y	aller de la position précédente jusqu'au point x,y en éclairant le chemin parcouru.

Le A qui termine chacun des codes opérations précédents indique que les coordonnées qui suivent sont absolues. Des ordres utilisant des coordonnées relatives, ou incrémentales, sont également disponibles :

POINTR	X,Y
POSR	X,Y
VECTR	X,Y

Ce sont donc essentiellement des ordres de déplacement et des indications d'intensité : nous les appellerons ordres graphiques.

Les autres possibilités, telles que tracé de caractères ou de cercles, euvent se réduire à des combinaisons de ces ordres élémentaires.

Les coordonnées indiquées correspondent aux points d'une grille (de dimension 1024*1024 ou 4096*4096 en général). Seuls les points de cette grille sont affichables.

Au bout d'un certain temps, appelé rémanence de l'écran, un point affiché disparaît. Il est donc nécessaire de régénérer une image pour la rendre stable. Les vitesses de régénération couramment employées se situent entre

30 et 60 fois par seconde. Ce phénomène de régénération est d'une importance capitale par les exigences qu'il impose. Nous aurons l'occasion de revenir de nombreuses fois sur ce point.

La production d'une image se ramène toujours à la production ultime d'une séquence d'ordres graphiques, séquence qui est envoyée au terminal soit élément par élément, soit globalement selon la puissance de ce terminal. On appellera liste d'affichage la séquence des ordres graphiques qui déterminent l'image affichée.

Nous allons introduire progressivement le concept de structures de représentation des données en étudiant divers types d'utilisation des terminaux graphiques, classés par ordre de complexité croissante.

Nous mettrons dans une première catégorie toutes les utilisations dans lesquelles le dessin est une forme de sortie des résultats. Nous appellerons ceci des utilisations "en traceurs de courbes" d'un terminal graphique.

Nous passerons ensuite à l'étude de systèmes plus complexes où l'image est utilisée comme forme d'entrée d'information et où l'aspect intéractif est plus développé.

B. UTILISATIONS EN TRACEURS DE COURBES.

Il s'agit ici de toutes les utilisations qui exploitent le fait que le support de sortie est à deux dimensions au lieu d'une . Ainsi au lieu de sortir des suites de chiffres représentant les coordonnées des points d'une courbe, il est plus agréable de sortir directement la courbe, ce qui en facilite grandement l'utilisation et la compréhension. Remarquons seulement qu'un traceur de courbes est mieux adapté à des emplois de ce genre, puisqu'il permet d'obtenir un dessin sur papier qui peut être conservé.

Cette gamme de travaux nous intéresse seulement dans la mesure où un certain nombre de mécanismes simples sont apparents et ainsi faciles à étudier.

La liste d'affichage qui produira le dessin définitif peut être obtenue de deux façons :

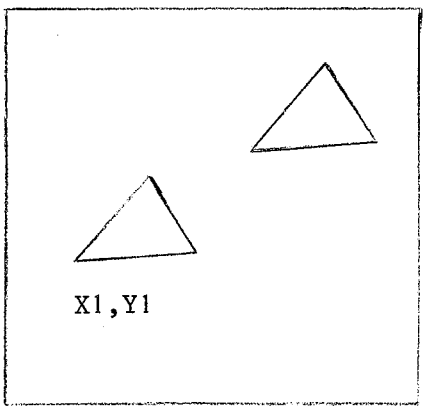
- par écriture directe d'un programme de tracé
- par écriture d'un algorithme qui génère une liste d'affichage à partir d'un certain nombre d'informations.

B.1 Ecriture directe.

Le programme est ici directement constitué par la séquence des ordres graphiques, et le dessin produit sera immuable à quelques paramètres près (abscisse, ordonnée etc....).

Exemple :

```
POSA   0,0
POINTR X1,Y1
VECTR  X2,Y2
VECTR  X3,Y3
VECTR  X4,Y4
POINTR X5,Y5
VECTR  X2,Y2
VECTR  X3,Y3
VECTR  X4,Y4
```



Ce programme très élémentaire peut être considéré comme un tableau à deux dimensions :

POSA	0	0
PONTR	X1	Y1
VECTR	X2	Y2
. . .	. . .	. . .
VECTR	X4	Y4

Ce tableau définit complètement notre dessin. Nous dirons qu'il constitue la <u>représentation matricielle</u> du dessin.
Remarquons que toute l'information définissant le dessin est contenue dans cette représentation.

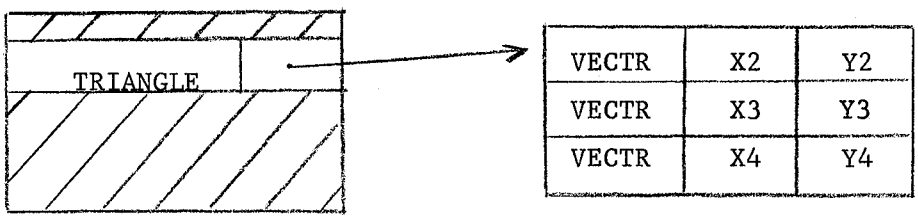
La présence d'ordres utilisant des coordonnées relatives va permettre de ne pas réécrire la séquence correspondant à une figure lorsqu'on veut tracer cette figure en des emplacements différents. Si nous ajoutons l'ordre élémentaire :

AFFICHER nom-de-figure

au répertoire de notre dispositif de tracé, le programme précédent devient :

```
POSA        0,0
POINTR      X1,Y1
AFFICHER     TRIANGLE
POINTR      X5,Y5
AFFICHER     TRIANGLE
```

Une simple table permet d'associer les noms de figures à leurs représentations matricielles.



Il est évident que le mécanisme réel est un peu plus compliqué puisqu'il faut pouvoir revenir au programme principal. Il sera donc nécessaire

de conserver une adresse de retour qui permettra de reprendre la séquence principale après l'exécution d'un tel sous-programme. Ces problèmes sont bien connus en programmation classique et nous les laissons de côté pour le moment.

Remarquons que cette utilisation des coordonnées relatives permet seulement de tracer des figures ayant subi des translations par rapport à la figure d'origine. Il n'est pas question de pouvoir effectuer des changements d'échelle ou des rotations avec cette méthode.

B.2 Production d'une liste d'affichage par un algorithme.

Il est vite fastidieux et parfois impossible d'écrire complètement un programme de tracé et on veut maintenant écrire un algorithme qui génère la liste d'affichage à partir de certaines informations.

Prenons l'exemple d'un tracé de courbes.

Soit une fonction

$$Y = F(X)$$

Un algorithme peut calculer les valeurs y1, y2, y3,...,yn de la fonction correspondant aux valeurs x1, x2,...,xn de la variable. Supposons que ces couples de valeurs (xi, yi) soient rangés dans un tableau à deux dimensions.

x1	y1
x2	y2
⋮	⋮
xn	yn

Il est alors extrêmement simple de générer la liste d'affichage (ou la représentation matricielle) correspondant à la courbe.
Il suffit pour cela de rajouter devant chaque couple (xi, yi) un code de tracé de point (ou de vecteur selon la nature du tracé de courbe voulue).

Prenons une fonction plus complexe : z = f(x,y)
De la même façon un algorithme obtiendra des triplets (xi, yi, zi) si on lui fournit les valeurs xi et yi. Le tableau des résultats devient le suivant :

x1	y1	z1
x2	y2	z2
.	.	.
.	.	.
.	.	.
xn	yn	zn

Le passage de ce tableau à la liste d'affichage n'est cependant pas tout à fait immédiat. Le dispositif de tracé ne permettant qu'un dessin à deux dimensions, il faut appliquer un certain algorithme de projection au tableau afin d'obtenir une liste d'affichage correcte.

x1	y1	z1
x2	y2	z2
.	.	.
.	.	.
.	.	.
xn	yn	zn

$\xrightarrow[\text{projection}]{\text{algorithme de}}$

POINTA	A1	B1
POINTA	A2	B2
.	.	.
.	.	.
.	.	.
POINTA	An	Bn

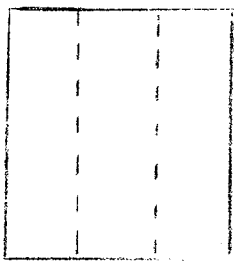
Considérons maintenant une utilisation un peu différente mais qui se classe toujours dans cette même catégorie: la fabrication de dessins animés.

Il s'agit essentiellement de faire subir à des figures simples (c'est-à-dire définies par des segments et des points) des transformations géométriques telles que translation, rotation, homothétie etc....ou des transformations mathématiques (par exemple $x \rightarrow x^2$, $y \rightarrow y^2$),

Les figures sont définies par l'écriture d'un programme qui permet d'obtenir leurs représentations matricielles.

POINTR	X1	Y1
VECTR	X2	Y2
.	.	.
.	.	.
.	.	.
VECTR	Xn	Yn

transformation



liste d'affichage ou représentation matricielle intermédiaire.

Il est alors très simple d'appliquer des transformations : il suffit d'explorer la matrice, de tester le code opération et d'effectuer (ou non) un certain calcul pour obtenir de nouvelles coordonnées à partir des anciennes.

Prenons comme exemple de transformation une homothétie de rapport K, par rapport au point x0, y0.
Les calculs sont donnés par les formules suivantes :

(1) si les coordonnées sont absolues

$$x \longrightarrow x' = K(x-x_o) + x_o$$

$$y \longrightarrow y' = K(y-y_o) + y_o$$

(2) si les coordonnées sont relatives

$$\delta x \longrightarrow (\delta x)' = K.\ \delta x$$

$$\delta y \longrightarrow (\delta y)' = K.\ \delta y$$

Les exemples que nous venons de voir peuvent être étendus à l'espace à trois dimensions. Il suffit d'ajouter une troisième coordonnée dans la représentation matricielle.
Notons ici l'intérêt des coordonnées homogènes qui permettent de manipuler facilement des figures à trois dimensions. Les transformations s'expriment alors par des opérations matricielles qui sont commodes à réaliser avec cette structure.

Résumons en quelques mots le principe des coordonnées homogènes :

Le point de départ est l'introduction d'une quatrième composante pour représenter un point dans l'espace. Ainsi, le point (x,y,z) sera représenté par un vecteur ligne de dimension :

$$\begin{vmatrix} X & Y & Z & W \end{vmatrix}, \text{ avec } x = \frac{X}{W},\ y = \frac{Y}{W} \text{ et } z = \frac{Z}{W}.$$

De même un plan sera représenté par un vecteur colonne de dimension 4.

L'introduction de cette quatrième composante, assimilable à un facteur d'échelle, permet, entre autres possibilités, de représenter des points à l'infini avec des coordonnées finies (on prend W = 0).

Les opérations de rotation, de translation, de changement de perspective suivant un axe et de changement d'échelle se ramènent alors à une seule multiplication d'un vecteur de dimension 4 par une matrice carrée d'ordre 4, qui est constituée de la façon suivante :

$$\left|\begin{array}{c|c} R & T \\ \hline P & E \end{array}\right|$$

R:	matrice	3×3	matrice de rotation
T:	"	3×1	vecteur de translation
P:	"	1×3	vecteur de changement de perspective
E:	"	1×1	facteur d'échelle.

Une description plus complète de cette méthode et de ses applications peut être trouvée dans les ouvrages référencés 1-ADI et 1-AHC.

C. UTILISATION EN DESSIN ASSISTE

C.1 Caractéristiques principales.

La présentation de résultats sous forme de dessins est incontestablement un point intéressant, mais il paraît encore plus intéressant d'utiliser l'écran d'une console comme support d'entrée à deux dimensions.

C'est le but de certains systèmes destinés aux utilisations "à dessin assisté", dans lesquelles l'utilisateur veut construire un dessin directement sur l'écran, en bénéficiant, du fait de la présence de l'ordinateur, d'une certaine correction géométrique du tracé (comme, par exemple, des segments vraiment rectilignes, des cercles vraiment ronds etc...),et, parfois, de corrections plus complexes dénommées contraintes (telles que parallélisme de deux segments etc...)

Un exemple bien connu et qui constitue un classique du genre est le système SKETCHPAD, sur lequel nous reviendrons plus loin.

Le dessin devient ici une forme d'entrée d'informations et l'aspect intéractif du processus de travail devient un élément essentiel.

En effet, l'utilisateur travaille par adjonctions et modifications successives et le système doit interpréter correctement les actions de cet utilisateur et conserver les résultats correspondants. Les actions demandées peuvent être très diverses :

(1) ajouter un nouvel élément (point, segment etc....)

(2) supprimer un élément existant,

(3) déplacer des éléments, changer leur échelle etc....,

(4) classer un groupe d'éléments comme une figure.

Cette figure peut à son tour être soumise à diverses opérations de déplacement, de changement d'échelle, de suppression etc.... Ces opérations doivent naturellement se répercuter sur les composants de la figure.

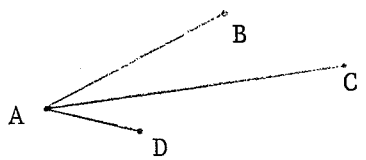
Bien entendu, la liste d'affichage doit être construite et modifiée

selon les actions de l'utilisateur. La nécessité de prévoir de telles possibilités a des conséquences importantes qui vont nous amener à nous éloigner de la représentation matricielle.

a/ Le processus de construction d'une image est dynamique. Ainsi, les ordres graphiques définissant l'image doivent varier en nombre et en qualité pour suivre les modifications. Or il est assez malaisé de rajouter, d'intercaler et de supprimer des éléments dans une matrice (qui a une dimension fixe et ne peut être coupée en morceaux).

De plus la présence de coordonnées relatives entraîne une translation de certains éléments si on modifie ceux qui précèdent.

b/ Il faut conserver les relations entre les éléments du dessin. Considérons l'exemple suivant :



Nous avons ici quatre points A, B, C, D et trois segments AB, AC et AD.

La commande "Suppression du point A" doit entraîner, en plus de la disparition du point A lui-même, celle des trois segments AB, AC et AD qui ont été définis à partir du point A.

De même, un déplacement du point A devra entraîner une modification des trois segments.

Ceci revient à dire qu'il faut conserver explicitement les relations entre les éléments, ainsi que les hiérarchies qui existent entre eux.

La représentation matricielle ne satisfait pas à cette condition: les relations entre les éléments ne sont absolument pas explicites et sont à peu près impossible à déterminer. Ces insuffisances ont entraîné le développement de diverses solutions que nous allons voir maintenant.

C.2 Notion de plex.terminologie utilisée.

Toutes ces solutions s'appuieront sur l'utilisation de "mémoires chaînées"; nous désignerons ainsi un système dans lequel le successeur d'un

élément d'information n'est pas simplement l'élément qui lui est adjacent, mais un élément situé en un emplacement quelconque de la mémoire. Chaque élément doit alors désigner lui-même son successeur et pour cela contenir, par exemple, l'adresse de l'élément successeur (d'où le terme de "chaînage").

Nous supposerons connues les techniques de base de manipulation des mémoires chaînées. (On pourra en trouver un excellent exposé dans le livre de D.E. KNUH [3-KNU] .)

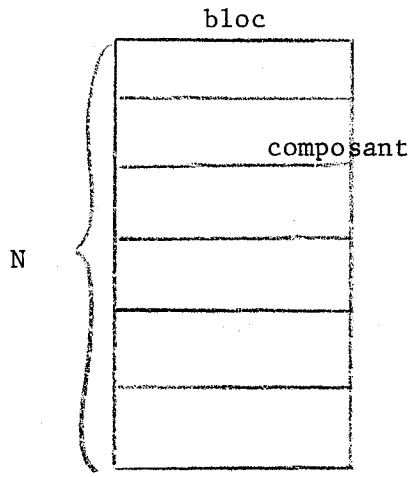
Nous allons cependant définir quelques termes que nous emploierons largement dans la suite.

Notion de Plex

Le mot PLEX a été introduit vers 1960 par D.T.ROSS [3-ROS1] pour désigner une méthode très générale d'utilisation de mémoires non consécutives. Il ne faut pas voir dans ce terme une solution magique à tous les problèmes. Ce mot recouvre un ensemble très vaste de techniques diverses qu'il importe d'adapter aux besoins de chacun.

Précisons un peu ce que ce terme recouvre.

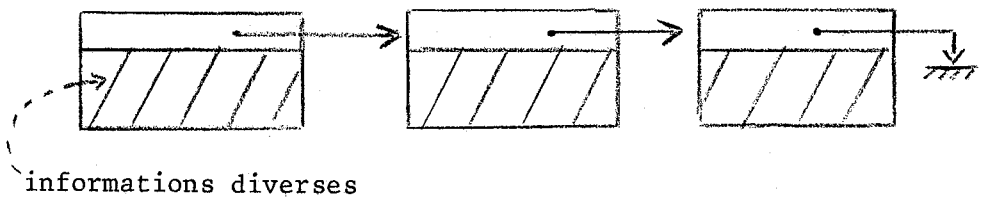
Les éléments de base sont des blocs, un bloc étant un groupe de N mémoires consécutives qui sont appelées composants du bloc.



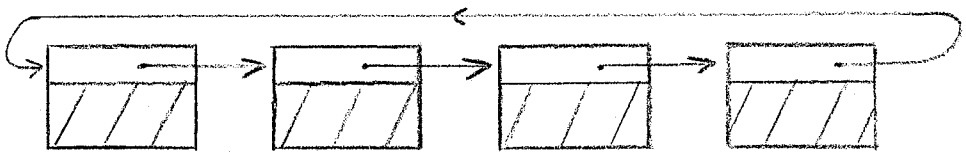
Dans les composants d'un bloc, on peut mettre des valeurs(nombres, caractères etc...) ou des adresses.
Ces adresses permettent de relier un bloc à un ou plusieurs autres. On peut ainsi créer des réseaux interconnectés de blocs, que l'on désigne d'une manière plus concise par le terme PLEX.

Ces réseaux sont alors le support d'information utilisé par des algorithmes spécialement adaptés.

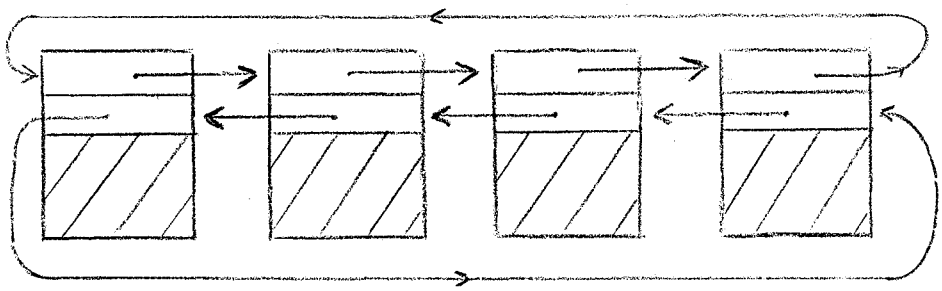
Voici quelques exemples partiels qui peuvent être pris comme des exemples de plex :



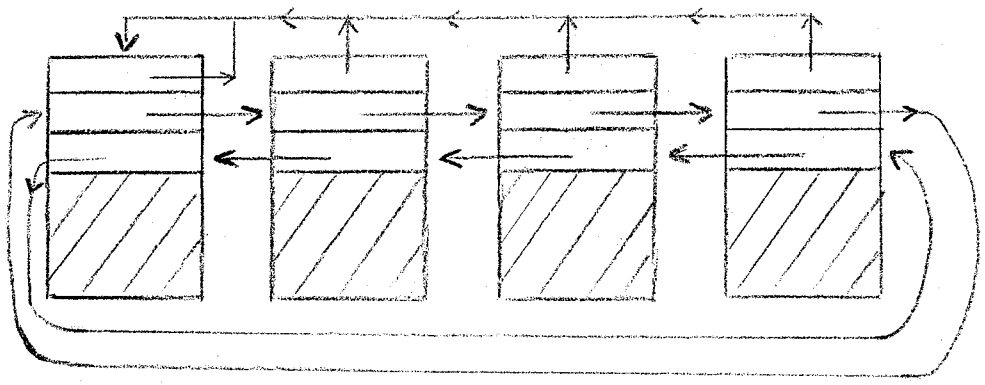
Chaîne (simple)



Anneau (simple)



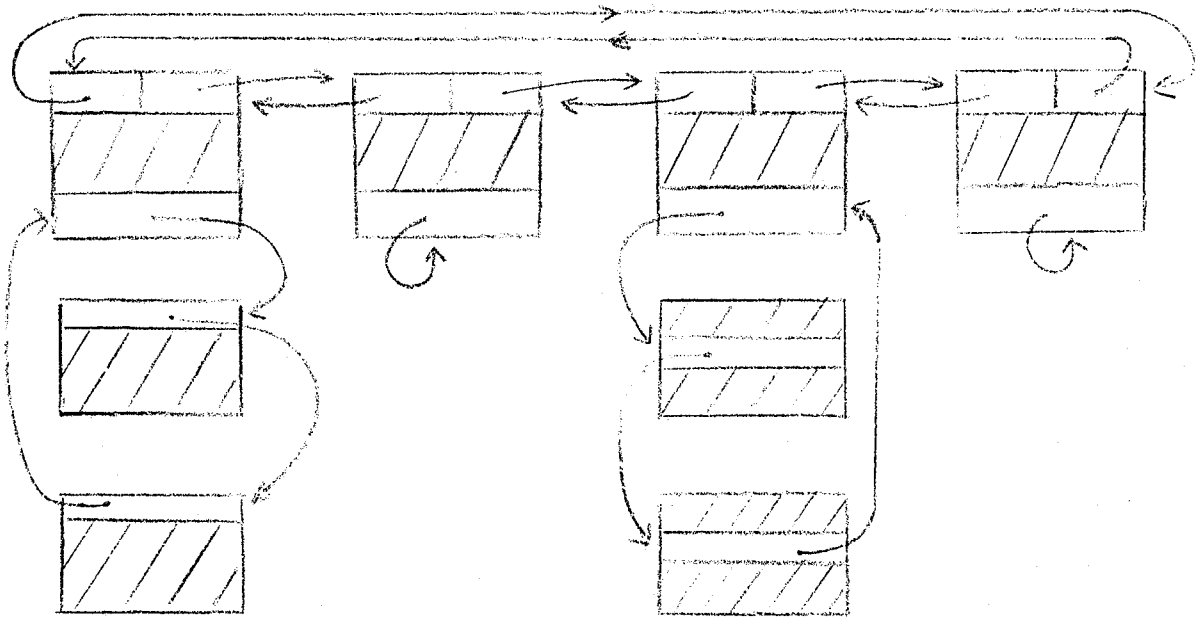
Anneau double



Anneau complet

Bien entendu, l'ordre et la dimension des composants contenant l'information ne sont pas rigoureux et toutes les combinaisons possibles sont permises dans le cadre de cette terminologie extrêmement simple.

Voici un autre exemple déjà un peu plus compliqué :



C.3 Le système SKETCHPAD (1963). [2-CAG1 , 2-SUT1]

Avant d'aborder la partie "représentation interne" qui nous intéresse plus particulièrement, rappelons rapidement le fonctionnement de ce système.

Il s'agit d'un système typique de dessin assisté dans lequel un utilisateur, assis devant la console, peut définir un dessin à partir d'éléments de base tels que points, vecteurs, cercles etc.... La communication se fait par l'intermédiaire du pointeur optique et des touches de fonctions. En plus de la génération de ces éléments de base, le système peut assurer une certaine correction géométrique du tracé, et offre, par exemple, des fonctions de parallélisme, de perpendicularité etc....

Le plus grand mérite de ce système a été de faire découvrir

l'intérêt des terminaux graphiques, et de faire prendre conscience des difficultés qui se présentent, tout en donnant un certain nombre de solutions qui ont servi de base aux travaux ultérieurs.

Le point le plus complexe de ce système est la structure d'information choisie, que nous allons étudier maintenant en détail.

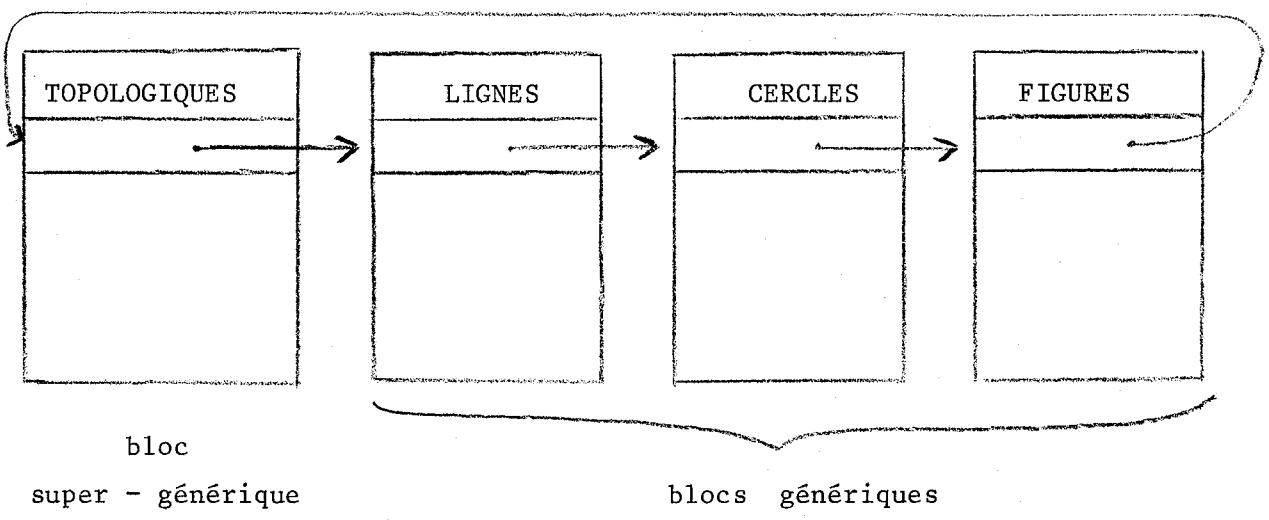
Les éléments disponibles sont classés en plusieurs types. On trouve par exemple :

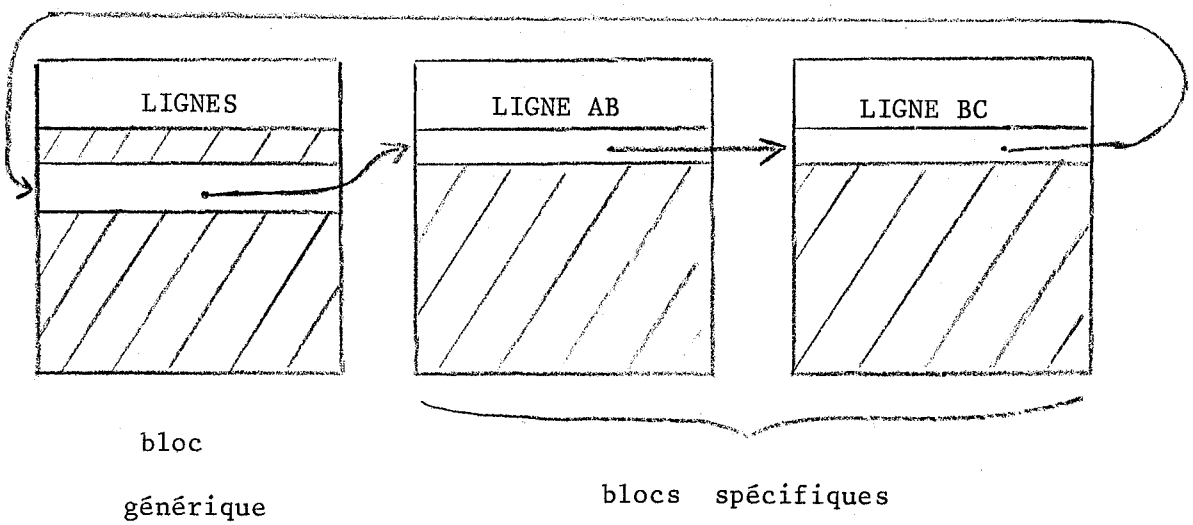
éléments VARIABLES tels que points, scalaires, textes, etc....
éléments TOPOLOGIQUES tels que lignes, cercles, figures etc....
CONTRAINTES telles que horizontalité, perpendicalarité etc....
éléments UTILITAIRES tels que éléments libres, éléments en déplacements etc....

Les types les plus généraux (VARIABLES, TOPOLOGIQUES etc...) sont appelés types "super-génériques". Les types plus précis (points, lignes, figures etc....) sont des types "génériques".

A chacun de ces types correspond un anneau sur lequel sont regroupés tous les éléments de ce type, chaque élément étant représenté par un bloc.

Exemple :

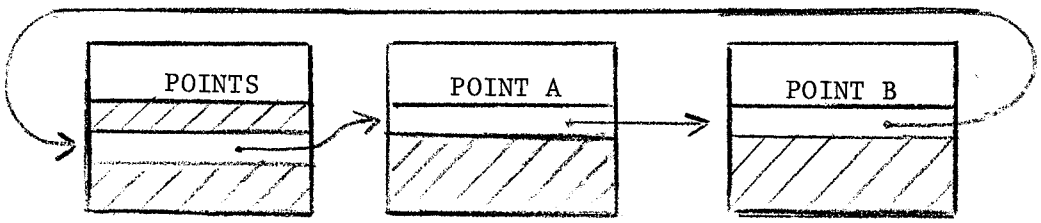




Ceci établit une hiérarchie entre les divers éléments. Ainsi partant du bloc "TOPOLOGIQUES", on trouvera, en particulier, le bloc générique LIGNES. A partir de ce dernier, on pourra alors trouver les blocs correspondant à toutes les lignes précédemment définies.

Les blocs génériques contiennent de plus toutes les informations propres au type générique qu'ils représentent. Ainsi, le bloc générique "CERCLES" pourra contenir l'adresse du sous-programme de tracé de cercles, adresse qu'il est évidemment inutile de répéter dans chaque bloc spécifique cercle.

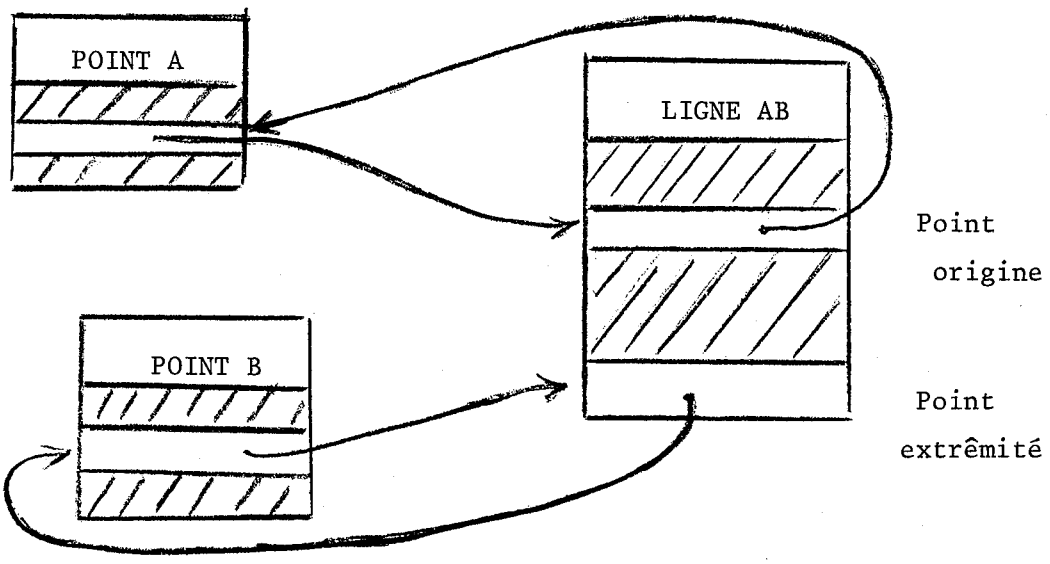
En début de travail, avant toute création, il n'existe qu'un embryon de structure , formé par l'ensemble des blocs super-génériques et génériques. Ce modèle interne est alors complété par adjonctions de blocs spécifiques en fonction des actions de l'utilisateur. Ainsi, après l'introduction des points A et B, on aura :



Ceci permet de suivre l'évolution dynamique du dessin et remédie donc à l'une des insuffisances constatées de la représentation matricielle.

Passons maintenant aux relations entre les éléments. Celles-ci vont être exprimées par le regroupement sur des anneaux fonctionnels d'éléments ayant des liens fonctionnels entre eux.

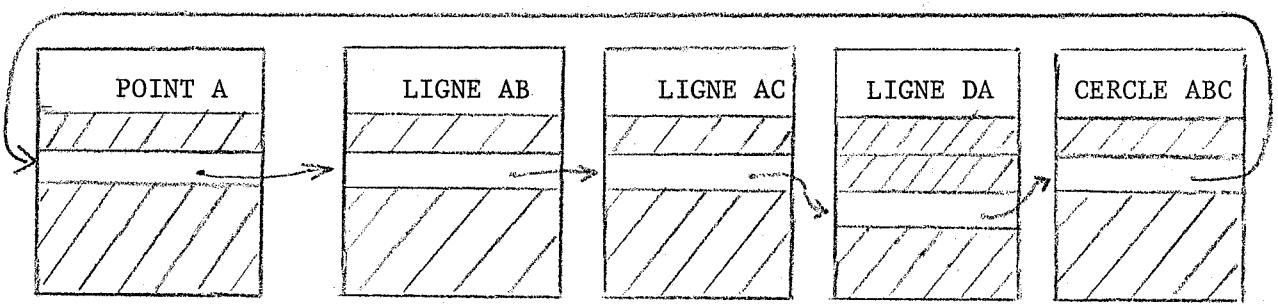
Considérons les points A et B et le segment AB : un anneau partant du bloc "point A" regroupera tous les éléments topologiques (lignes, cercles..) dont la définition utilise ce point. Il en sera de même pour le point B.



Le processus de tracé du segment AB est alors le suivant : accéder au bloc spécifique représentant AB; explorer ensuite l'anneau passant par le composant "point origine" de ce bloc; on atteint ainsi le bloc "point A" dans lequel se trouvent les coordonnées de ce point. On procède de la même façon, à partir du composant "point extrémité" pour trouver les coordonnées du point extrémité. Le sous-programme "tracé de vecteurs", dont l'adresse est dans le bloc générique "LIGNES" est alors appelé, avec les coordonnées trouvées comme paramètres.

Examinons maintenant une situation un peu plus compliquée : nous avons le point A, les segments AB, AC et DA, et le cercle ABC.

L'anneau d'utilisation du point A est alors le suivant :



Pour trouver le point A à partir du bloc "ligne AB", il faudrait parcourir tout l'anneau.

Ce mécanisme répété de nombreuses fois (le même phénomène se retrouvant évidemment sur tous les anneaux) diminuerait le temps de réponse du système. Aussi, les liaisons utilisées sont-elles un peu plus élaborées : dans un anneau chaque élément possède, en plus d'un pointeur vers l'élément suivant, un pointeur vers l'élément de tête de l'anneau. Partant d'un bloc ligne, on peut alors trouver immédiatement les blocs des points origine et extrêmité.

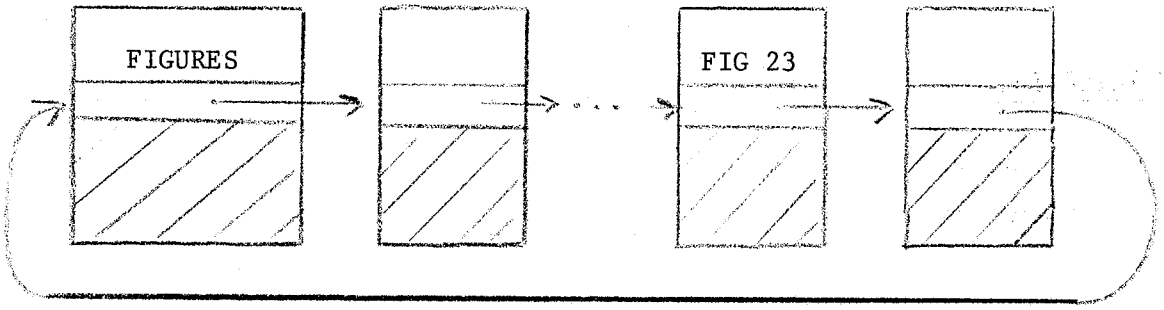
De plus, afin de faciliter diverses opérations sur les anneaux, chaque élément pointe aussi sur l'élément qui le précède.

Enfin, signalons l'existence d'un quatrième pointeur qui désigne la première mémoire du bloc dans lequel on se trouve. L'intérêt de cette liaison est évident sur l'exemple précédent : en suivant l'anneau d'utilisation du bloc A, on peut atteindre un bloc ligne en deux endroits différents selon que le point est origine ou extrêmité de cette ligne.

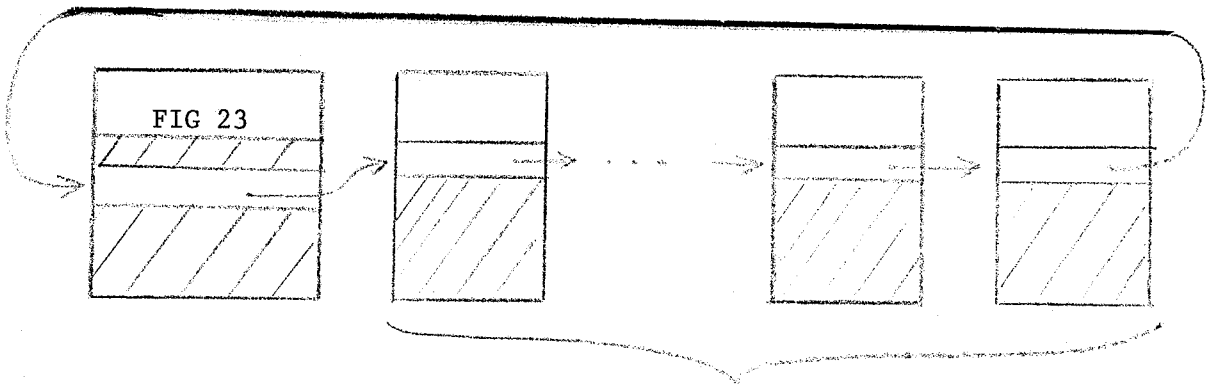
En résumé, chaque liaison utilisée est donc quadruple. Nous ne représentons cependant que la liaison vers l'élément suivant, afin de donner plus de clarté aux schémas.

Etudions maintenant l'organisation des figures.

Un bloc générique "FIGURES" regroupe l'ensemble des figures, chaque figure étant représentée par un bloc spécifique particulier.



Un anneau partant du bloc spécifique d'une figure réunit tous les éléments constituant cette figure.



Blocs spécifiques représentant les éléments de la figure.

Enfin l'ensemble des figures qui forment l'image affichée se trouve relié par un anneau partant d'un bloc générique spécial.

Toutes les autres opérations telles que contraintes, déplacements, etc....se ramènent également à des manipulations de blocs et de pointeurs, en liaison avec des algorithmes d'exploration d'anneaux.

Ces diverses opérations sont cependant très lourdes du fait du grand nombre de pointeurs à modifier pour changer une liaison, et de la complexité de ces liaisons. De plus, le système nécessite des blocs de dimension importante puisque la place nécessaire pour toutes les liaisons possibles doit être prévue dès le départ. Ceci conduit à un gaspillage de place en mémoire puisqu'on prend un bloc de taille maximum dès la création de l'élément sans savoir si cet élément sera largement utilisé ou non.

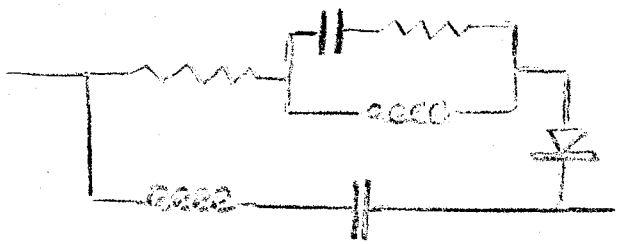
D. UTILISATIONS TOPOLOGIQUES

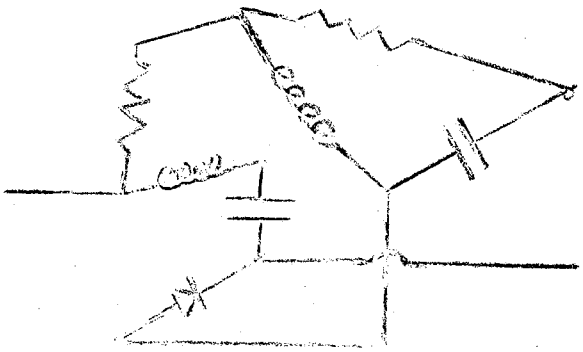
Dans cette dernière catégorie d'utilisation des consoles de visualisation, nous regroupons les systèmes dans lesquels l'image est un moyen d'entrée d'information et un moyen de sortie de résultats mais ne constitue pas l'essentiel.

Dans cette gamme entrent les problèmes "topologiques". Ces derniers utilisent des éléments généralement prédéfinis, dont l'utilisateur indique la position sur l'écran. Les connections entre ces éléments sont ensuite placées selon les désirs de cet utilisateur.

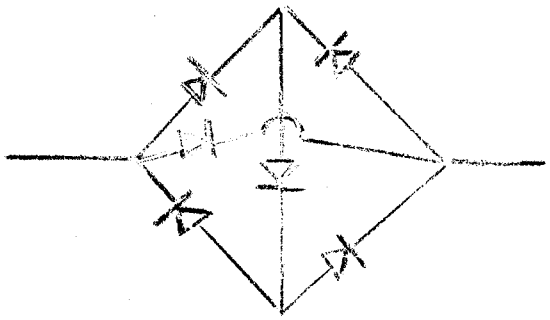
La position géométrique précise des divers éléments n'a pas d'importance et seules comptent les liaisons entre éléments et la nature de ces éléments.

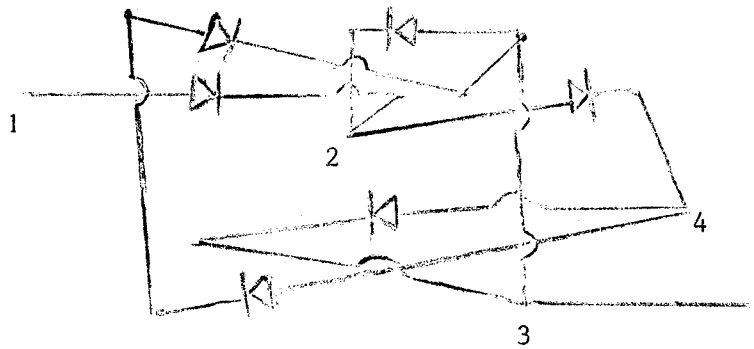
Prenons pour exemple un circuit électrique : il peut avoir des apparences très différentes et pourtant la même valeur du point de vue électrique. Ainsi les deux circuits suivants sont équivalents :





Il en est de même pour les deux suivants :





On distingue bien sur ces deux exemples la différence entre <u>l'aspect syntaxique</u> (ou dessin) et <u>l'aspect sémantique</u> (ou signification du dessin).

Il y a ici superposition de deux problèmes : Le problème sémantique, ou problème <u>réel</u>, posé par la signification réelle du dessin et qui constitue le problème de fond: par exemple, déterminer l'impédance du circuit parcouru par un courant alternatif de fréquence donnée; le problème syntaxique posé par la forme géométrique de l'image, qui est un problème secondaire, mais qui ne peut être totalement négligé. L'utilisateur serait sans doute assez déconcerté de voir réapparaître une forme équivalente, mais différente, de son circuit, après interprétation par le système.

Dans les paragraphes précédents, nous avons étudié le problème de l'image et nous avons vu des exemples de représentations internes qui peuvent être utilisées.

Il est important de noter que la définition du problème sémantique présente les mêmes caractéristiques que la construction du dessin et que ce

ne sont en fait que deux interprétations différentes d'un même processus.

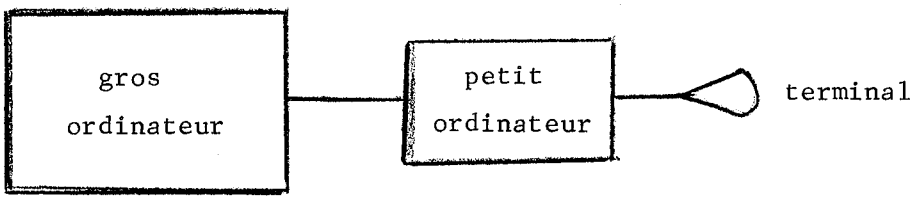
Les contraintes imposées par l'évolution dynamique de l'image subsistent donc et il est logique de penser qu'une représentation en plex est la solution encore une fois la plus adaptée. Il serait évidemment très agréable que les algorithmes de résolution des deux problèmes en présence puissent utiliser la même représentation des informations. Cependant deux objections importantes peuvent être faites.

La première, c'est que la résolution du problème réel est indépendante de l'image. Ce problème réel peut avoir été l'objet d'études en fonction desquelles des algorithmes et une certaine représentation des données, adaptés à la nature du problème ont été développés. Cette adaptation reste évidemment nécessaire dans la mesure où on veut assurer des performances satisfaisantes pour les algorithmes choisis.

Il n'y a aucune raison pour que la représentation adaptée aux problèmes précis posés par l'image soit également adaptée aux problèmes totalement différents posés par la signification que l'on accorde à cette image.

La deuxième objection est liée à des considérations sur le matériel. Les terminaux graphiques étant des périphériques très rapides peuvent arriver à "occuper" presque complètement un petit ordinateur. (Nous reviendrons au paragraphe suivant sur la notion de mémoire d'entretien).

Aussi, on peut trouver des systèmes constitués de la manière suivante [2-CHP] :



Le petit ordinateur s'occupe alors de l'image, tandis que le gros fait les calculs correspondants entre autres choses. La présence du petit calculateur assure une réponse plus rapide aux modifications apportées au dessin. Le gros ordinateur se contente d'une version simplifiée de la représentation correspondant à l'image elle-même et contient surtout la représentation adaptée au problème sémantique. Ainsi dans ce type de configuration, il est plus commode d'avoir deux représentations distinctes, chacune d'entre-elles pouvant alors être optimisée.

E. INFLUENCE DU MATERIEL UTILISE.

Nous n'avons pas précisé jusqu'ici à quel type de consoles de visualisation nous nous intéressions. Nous excluons les terminaux alphanumétiques, qui posent des problèmes qui se rapprochent plus de ceux posés par la manipulation de textes. Les terminaux graphiques présentent toutefois des caractéristiques assez diverses selon leur complexité.

E.1 Terminal graphique sans mémoire d'entretien.

Le matériel le plus simple est constitué par l'unité d'affichage, comportant l'écran à tube cathodique, et un décodeur d'ordres, le tout étant relié à un ordinateur de dimension réduite.

Il importe assez peu que ce terminal dispose ou non d'un générateurs de vecteurs ou de caractères. Les performances en sont bien sûr améliorées, mais le fonctionnement reste le même.

Les ordres composant la liste d'affichage doivent être envoyés l'un après l'autre au terminal par le calculateur. Ce dernier doit naturellement assurer de plus la permanence de l'image, c'est-à-dire qu'il lui faut répéter l'envoi de toute la liste d'affichage environ trente à quarante fois par seconde.

Ceci constitue une charge très lourde pour le calculateur qui dispose alors de peu de temps pour faire autre chose. Il importe donc de ne pas compliquer le processus d'obtention de la liste d'affichage. En particulier, il faut éviter d'utiliser des représentations internes trop complexes dont l'exploration prendrait trop de temps.

Ainsi, avec un ordinateur limité, on se trouve contraint d'utiliser des représentations très proches de la représentation matricielle. Cette limitation disparait si on dispose d'un ordinateur puissant, réservé complètement pour la gestion du terminal graphique; l'exemple du système SKETCHPAD, utilisant le célèbre TX-2, est là pour le prouver.

L'association terminal-petit ordinateur est une forme d'utilisation qui se révèle bien adaptée à certaines applications telle la production de dessins animés. [1-LUC]

Il est en effet facile de modifier légèrement la liste d'affichage entre chaque régénération; les régénérations étant ici quasi-permanentes, ces changements permettent la production d'images variant progressivement et suffisamment rapidement pour que l'oeil ait une impression de continuité.

La puissance limitée de l'ordinateur reste un facteur qui limite la gamme des animations possibles.

E.2 Terminal graphique avec mémoire d'entretien.

A un stade plus évolué, on trouve associé à l'unité d'affichage et au décodeur d'ordres, une mémoire propre à l'unité. La liste d'affichage est placée dans cette mémoire par le calculateur, qui n'a plus besoin d'intervenir ensuite, sauf pour modifier cette liste. L'image correspondante est produite par exploration automatique de cette mémoire et la régénération de l'image est assurée sans intervention du calculateur. D'où le nom de mémoire "d'entretien" donné à cette mémoire.

Ce dispositif soulage considérablement le calculateur connecté dont l'intervention n'est nécessaire que pour modifier la liste d'affichage et pour gérer les interruptions. Il devient possible de préparer en mémoire centrale la liste d'affichage en explorant une structure interne même très compliquée et ceci sans contraintes de temps abusives. Cette liste est ensuite envoyée d'un seul coup à la mémoire d'entretien et l'image correspondante est obtenue immédiatement.

Les ordres composant la liste d'affichage sont encore très élémentaires. On y trouve toutefois un ordre de transfert, qui permet de ne pas suivre séquentiellement la liste d'affichage si on le désire. En l'absence de possibilités d'appel de sous-programmes, il faut remarquer que l'on est obligé de répéter une séquence d'ordres pour produire une même figure, en deux emplacements différents. Il n'est pas possible dans ce cas d'exploiter utilement l'existence de coordonnées relatives.

E.3 Terminal avec mémoire d'entretien et appel de sous-programmes.

On arrive ici à un matériel représentant les mêmes caractéristiques

physiques que le précédent, mais qui dispose en plus, dans son répertoire d'ordres, d'un appel de sous-programme simple au niveau de la mémoire d'entretien.

On peut alors restreindre la liste d'affichage, si plusieurs figures analogues et déduites les unes des autres par translation apparaissent dans l'image. Un seul exemplaire de la séquence d'ordres constituant la figure sera placé dans la mémoire d'entretien, cette séquence utilisant les coordonnées relatives. Des appels de cette séquence seront mis à tous les endroits où la figure est désirée. Pendant l'exécution, la figure est générée à chaque fois en partant de la position du spot au moment de l'appel, ce qui a pour effet d'assurer la translation.

D E U X I E M E P A R T I E

CONSTRUCTION ET MANIPULATION DES STRUCTURES

DEUXIEME PARTIE

CONSTRUCTION ET MANIPULATION DES STRUCTURES

Le concept de structure d'information étant introduit, nous allons maintenant nous intéresser à l'étape suivante : de quels outils dispose-t-on pour construire une structure d'information et pour définir les algorithmes qui s'y rapportent ?

Ces outils sont représentés essentiellement par des langages de programmation. Le choix de l'un d'entre eux est toutefois rendu difficile par leur nombre et par la diversité de leurs aptitudes.

Afin de faciliter ce choix, nous commencerons par une étude d'une quinzaine de ces langages, choisis parmi les plus significatifs, en les classant selon les commodités qu'ils offrent dans le domaine de la construction et de la manipulation des structures.

En nous appuyant sur les notions dégagées par cette étude, nous décrirons ensuite un système de macro-instructions que nous avons développé afin de découvrir par l'intérieur les raisons de certaines limitations rencontrées fréquemment dans les langages précédents.

A. ETUDE COMPARATIVE DE LANGAGES DE PROGRAMMATION RELATIVEMENT AUX STRUCTURES D'INFORMATION.

1. LANGAGE MACHINE (OU LANGAGE D'ASSEMBLAGE) [4-IBM4]

Si aucun des langages existants ne satisfait un utilisateur particulier, celui-ci pourra toujours avoir recours au langage machine et définir les opérations qui lui sont nécessaires. La présence d'un macro-assembleur évolué est alors une aide très appréciable.

2. FORTRAN, ALGOL 60, COBOL.

Ces langages sont bien connus et il paraît inutile de revenir sur leurs formes et leurs objectifs. Ils n'ont évidemment pas été prévus pour manipuler des structures.

L'utilisation de représentation matricielle et de tableaux est pos sible, mais le manque de possibilités d'allocation dynamique de mémoires impose de prévoir la dimension des tableaux avant leur utilisation.

En ALGOL 60, la présence de blocs permet d'obtenir plus facilement un tableau supplémentaire pour ranger des résultats intermédiaires. Toutefois, à la sortie du bloc, cet espace sera perdu et ces résultats ne pourront être conservés.

Avec le qualificatif "rémanent", le tableau sera conservé, mais ses dimensions seront alors figées.

La définition d'une structure en forme de plex ne peut se faire directement, en l'absence de gestion dynamique de l'espace mémoire et en l'absence de variables pouvant contenir des adresses. Il est alors nécessaire de simuler ceci en utilisant des tableaux et en conservant comme adresse un indice dans un tableau.

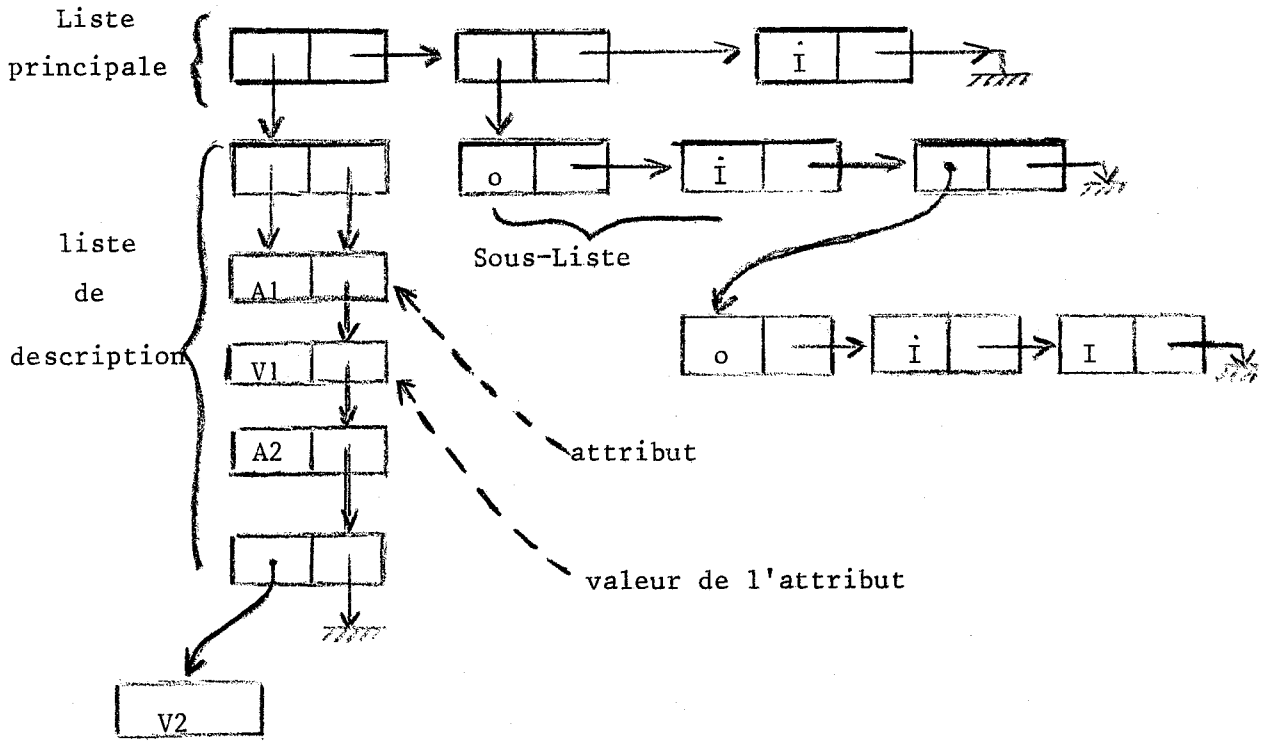
Seul COBOL permet de définir des éléments composés et aussi des tableaux d'éléments composés, ce qui donne la possibilité d'optimiser la place occupée et de travailler sur une information plus élaborée.

3. LANGAGES PERMETTANT D'UTILISER DES STRUCTURES : IPL-V,LISP et SLIP.

L'objectif de ces langages n'est pas de permettre à l'utilisateur de définir une structure quelconque, mais "simplement" de lui offrir des facilités pour ranger ses données de manière dynamique et pour exprimer certaines relations entre ses informations. Ils sont tous trois prévus pour travailler avec des listes.

a) IPL-V (Information processing language). [4-NET]

La structure utilisée pour représenter des listes est la suivante :



L'élément de base est ici le mot; chaque mot a une partie "information" et une partie "pointeur" utilisée pour désigner l'élément suivant de la liste. Comme un mot ne peut contenir qu'une quantité limitée d'information, il est utile de définir un mécanisme permettant d'associer des informations supplémentaires à un élément. La partie information de l'élément contient soit

la valeur de cet élément, soit un pointeur vers une liste de description. Le premier mot représente en fait la liste. Les mots suivants sont associés par couples : le premier représentant la nature d'un attribut. On peut ajouter un nombre quelconque d'attributs dans la liste de description d'un élément, ce qui permet de donner une valeur complexe à un élément particulier.

IPL-V forme un langage complet, mais très proche d'un langage d'assemblage. Sa syntaxe est rigide et peu agréable, comme le montre l'exemple suivant :

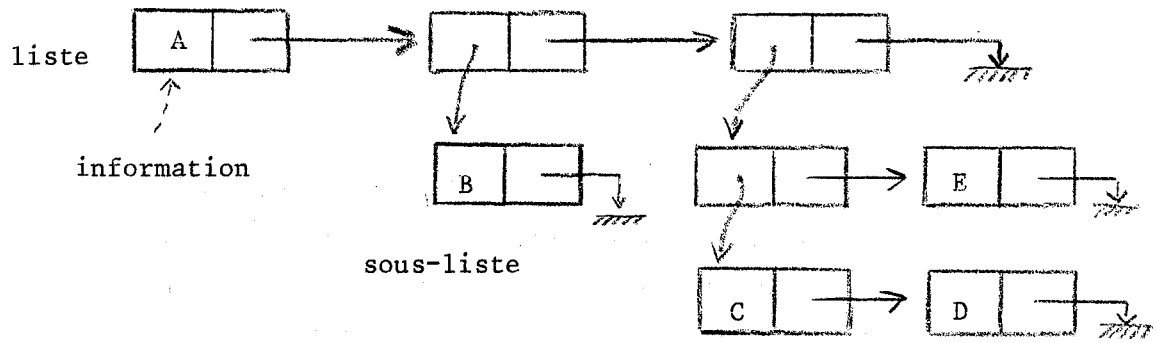
```
Eo              J50
        10      9-10
                J102
                J5          J30

9-10    70                  J8
        52      HO
        11      WO
                J2          J5
```

(Cette portion de programme recherche un élément dans une structure en arbre. J102, J5 etc.... sont des appels de sous-programmes.)

On voit que seul un initié a une chance de comprendre un tel programme!

b) LISP [4-MCC]

Représentation des listes utilisée :



Ce schéma représente la liste : (A(B)((C D)E))

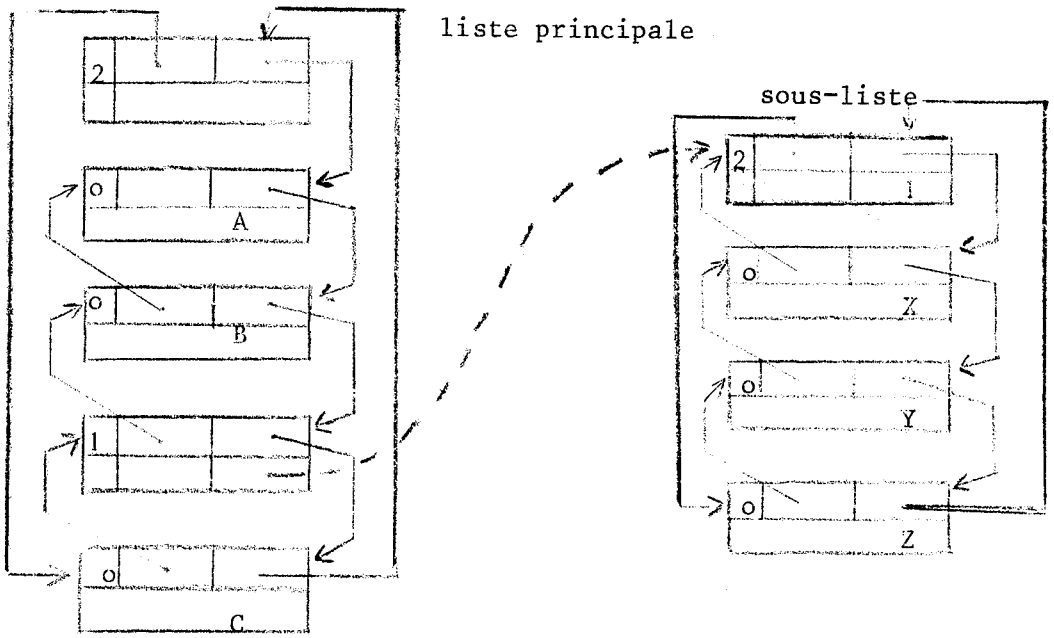
Ce langage est suffisamment répandu pour qu'il soit inutile d'insister sur sa syntaxe (écriture parenthésée) et sur ses applications (telles que traitement général de données symboliques pouvant faire appel à des processus récursifs).

c) SLIP (Symmetric LIST Processor) [4-WEI]

La représentation adoptée est un peu plus élaborée, car les liaisons sont à double sens.

Exemple : Représentation de

(A, B, (X, Y, Z), C)



SLIP est constitué par un ensemble de sous-programmes utilisables en FORTRAN. Remarquons les noms peu évocateurs :

NXTLFT, NXTRGT,

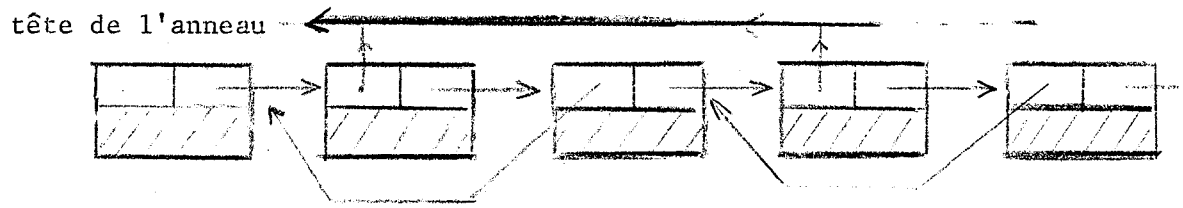
ADVSWR, ADVSER, ADVSWL, ADVSNL etc....

Ceux-ci ne facilitent pas la lecture des programmes.

4. LANGAGES PERMETTANT DE DEFINIR ET DE MANIPULER DES STRUCTURES DANS UN CADRE PREETABLI.

a) CORAL. [4-SUT]

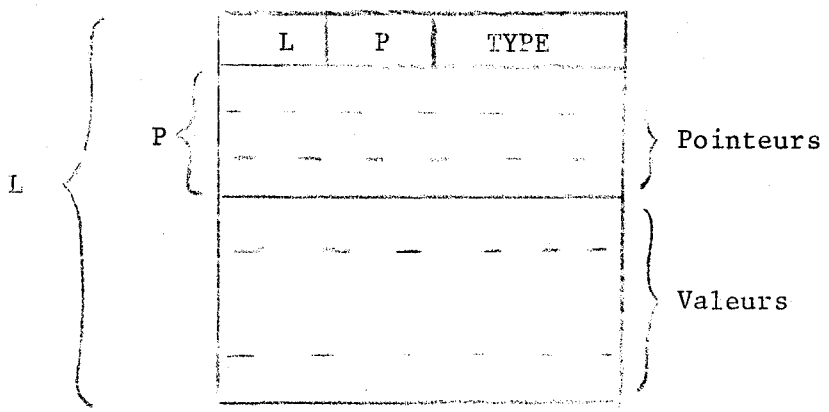
Ce langage, développé au MIT, a pour but de permettre une manipulation plus facile de structures en anneaux. La seule forme d'anneau utilisable est la suivante :



Cette méthode permet d'avoir les avantages d'un anneau complet (c'est-à-dire avec liaisons dans les deux sens), tout en réduisant la place occupée par les pointeurs.

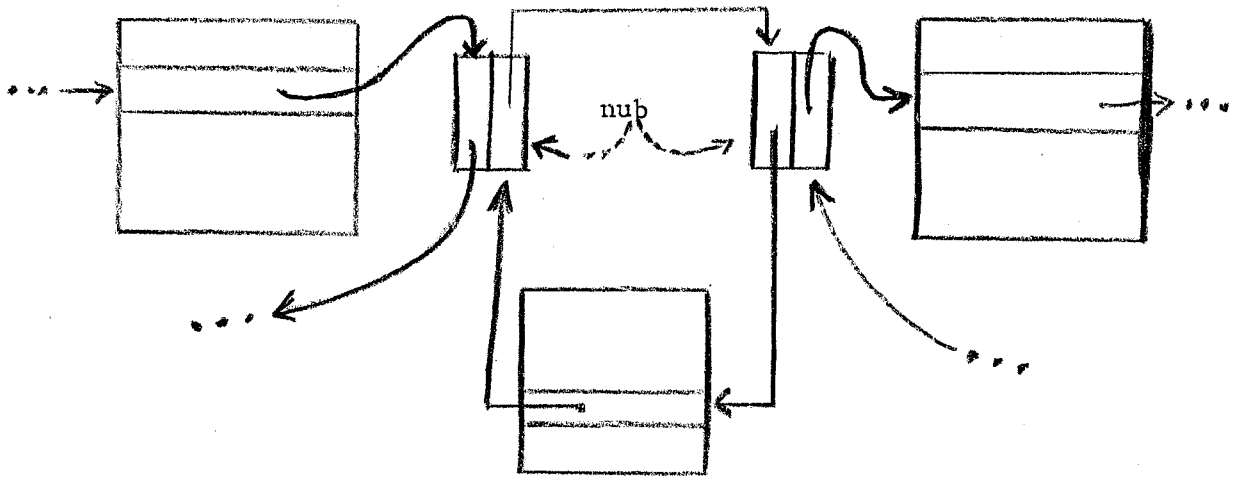
Notons cependant que l'insertion d'un nouvel élément sur un anneau impose de modifier un grand nombre de pointeurs, puisque ceux-ci dépendent de l'ordre d'apparition de l'élément sur l'anneau. Seule une insertion en queue d'un anneau peut être faite rapidement. Notons également que la machine utilisée (le TX-2) permet de mettre deux pointeurs dans un seul mot machine, ce qui réduit évidemment la place occupée.

Dans les blocs utilisés, les pointeurs sont placés en tête et les valeurs en fin de bloc.



Les blocs sont regroupés en un certain nombre de types et ils sont de longueur fixe, ceci facilitant la gestion de la mémoire libre.

Enfin une dernière chose à noter : la possibilité de faire passer plusieurs anneaux dans le même mot d'un bloc, par l'intermédiaire d'un double mot supplémentaire appelé nub.



Le langage CORAL lui-même est uniquement destiné à travailler sur des structures utilisant ce type d'anneaux. Sa forme est assez curieuse du fait de l'utilisation de nombreux symboles particuliers qui représentent les opérations à effectuer.

Exemples : ☞ R saut à l'adresse R

P ⇏ N → S ☞ R remonter jusqu'à l'élément de départ de l'anneau sur lequel se trouve le bloc désigné par P et aller vers la droite jusqu'au $N^{\text{ième}}$ élément dont l'adresse sera mise dans S. S'il n'y a pas de $N^{\text{ième}}$ élément, alors aller à l'adresse R.

(R) TP∧L → S fabriquer un bloc de type TP, de longueur L et mettre son adresse dans S.

P (>) Q → S Insérer le bloc désigné par P à droite de l'élément Q.

De nombreuses opérations de déplacement sur un anneau et dans un bloc, ainsi que des opérations pour créer des blocs et des anneaux sont fournies. On dispose également de tests (sur le type d'un bloc, etc...) et d'opérations de rangement d'informations dans les blocs. Remarquons toutefois le manque de rigueur de ces opérations qui manipulent aussi bien des blocs que des anneaux sans que rien ne les distingue.

Cet ensemble permet de travailler correctement sur ce genre de structures. La gestion de la mémoire libre est assurée par le système.

On peut reprocher à ce langage le trop grand nombre de symboles nouveaux utilisés qui permettent d'écrire des programmes concis, mais peu compréhensibles.

b) ASP (Associative Structure Package) [4-LAG]

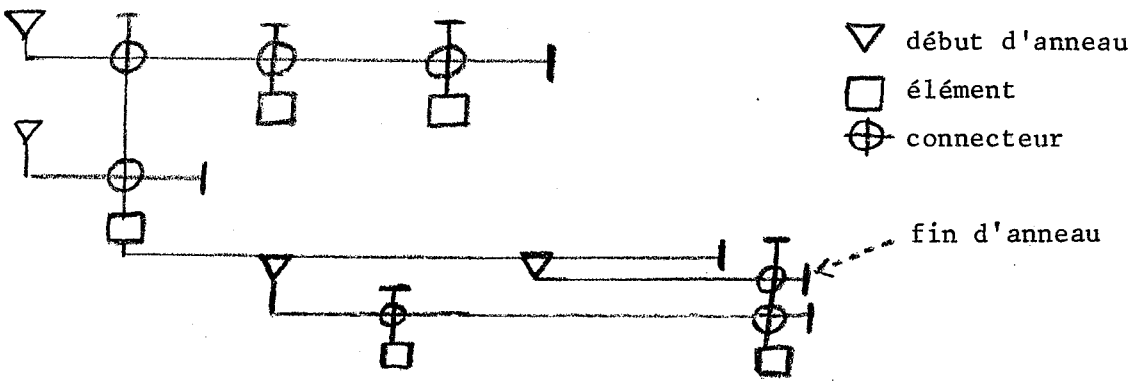
Toujours fondé sur l'utilisation d'anneaux, ce langage se veut toutefois très général.

Les entités de base sont de trois types : Element , début d'anneau et connecteur. Elles sont toutes représentées par des blocs.

Il existe deux types d'anneaux selon que leurs éléments sont des "débuts d'anneau" ou des "connecteurs". Un anneau a toujours pour élément de tête un "début d'anneau".

Chaque élément contient des données, un "début d'anneau" supérieur et un "début d'anneau"inférieur. L'anneau supérieur ne contient que des connecteurs et permet de mettre un élément sur divers anneaux. L'anneau inférieur ne contient que des débuts d'anneau et permet de trouver tous les anneaux appartenants à cet élément.

Il faut signaler une intéressante méthode pour représenter de telles structures en anneaux :



La barre verticale indique la fin de l'anneau. Les débuts d'anneau inférieur et supérieur de chaque élément ne sont pas représentés.

Le langage lui-même consiste en un ensemble de fonctions assez élémentaires que l'on peut appeler suivant ses besoins. Chaque fonction a un nom composé de 3 ou 4 lettres seulement ce qui ne facilite pas le travail de l'utilisateur.

Le niveau des opérations est élémentaire et, si le programmeur n'a pas à manipuler des pointeurs pour créer les liaisons, il lui faut tout de même manipuler élément, début d'anneau et connecteur par leurs adresses pour les regrouper en une structure particulière. Les créations et suppressions des éléments de base sont naturellement dynamiques.

Exemples : EC(L) crée un élément de longueur L mots et a pour valeur l'adresse de cet élément .

CSE(E,n,t) crée un début d'anneau de nom n et de type t sur l'anneau inférieur de l'élément E, et a pour valeur l'adresse de ce début d'anneau.

Des fonctions permettent de se déplacer sur les anneaux de un ou plusieurs blocs, de remonter à la tête d'un anneau, d'obtenir le nombre de blocs sur un anneau.....

Exemples: LRN(N,E,L) donne l'adresse du $N^{ième}$ début d'anneau sur l'anneau inférieur de l'élément dont l'adresse est E. Il y a saut à L s'il n'y a pas N débuts d'anneau.

URL(E) donne le nombre de connecteurs sur l'anneau supérieur de l'élément dont l'adresse est E.

Certaines fonctions un peu plus puissantes peuvent être utilisées essentiellement comme outils de mise au point. Ces fonctions permettent d'exprimer le contenu de tous les constituants d'un anneau.

Exemples : PRUR(E) imprime tous les connecteurs situés sur l'anneau supérieur de E.

PRER(S) imprime tous les éléments connectés à l'anneau dont le début est S.

L'intérêt essentiel de ce langage est l'aspect général et très dynamique de la structure utilisée.

La place réservée dans un bloc d'élément pour des liaisons ultérieures est réduite à un minimum (qui est constant), ce qui présente un gros avantage par rapport à ce qui est fait dans le langage APL(voir plus loin) et dans le système SKETCHPAD, par exemple.

L'un des objectifs de ce langage est de permettre des études sur les structures en anneaux.

c)APL (Associative Programming Language). [4-DOD]

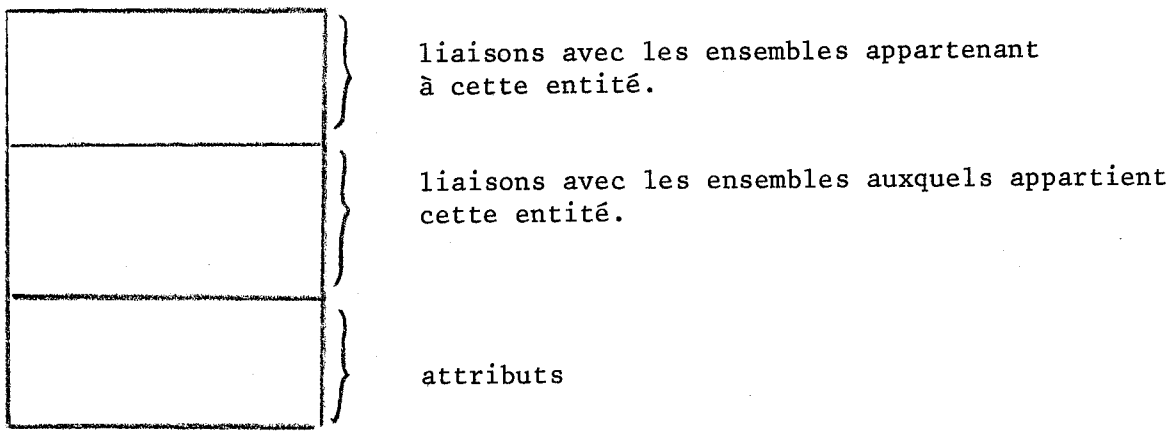
Ce langage, défini par G.DODD de la Compagnie General Motors et qu'il ne faut pas confondre avec celui de Iverson, a encore pour objectif la définition et la manipulation de structures utilisant des blocs et des anneaux. Les opérations fournies sont toutefois d'un niveau plus élevé que dans les langages précédents.

L'élément de base est l'Entité qui sera représentée par un bloc. Une entité est décrite par des attributs (plus précisément par la valeur de ces attributs).

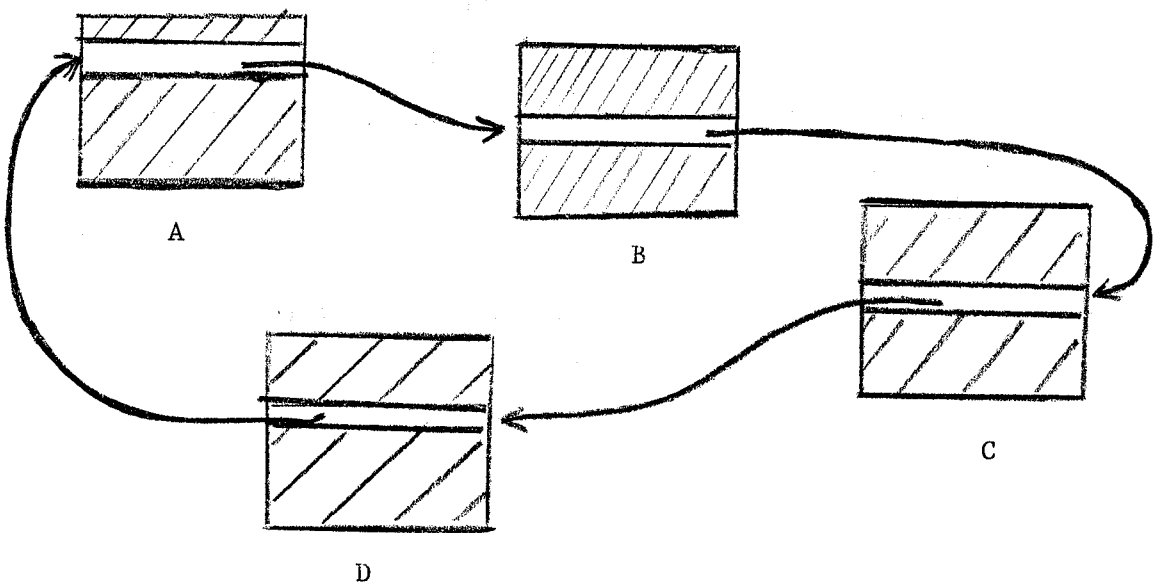
Ces entités pourront être regroupées dans des ensembles.

Un ensemble peut être la propriété exclusive d'une entité ou appartenir à plusieurs entités.

La forme d'un bloc représentant une entité est la suivante :



Exemple : les entités B,C,D sont mises dans un ensemble appartenant à l'entité A.



Ce langage est prévu pour être inséré dans le langage PL/1. Il utilise donc les facilités de ce dernier et ajoute des opérations de niveau élevé qui permettent de définir dynamiquement des entités et d'établir des relations entre elles.

c-1) La déclaration d'un type d'entité, et en particulier de ses attributs, se fait en utilisant essentiellement les déclarations de "structure" (ou d'éléments composés) qui existent en PL/1, simplement complétées par les mots ENTITY et ATTRIBUTE.
La création dynamique du bloc représentant une entité se fait par :
CREATE type d'entité CALLED nom-de-variable;

c-2) Constitution d'ensembles.
Insérer une entité dans un ensemble :
INSERT variable-entité IN ensemble;
Enlever une entité d'un ensemble :
REMOVE variable-entité FROM ensemble ;
Enlever une entité de tous les ensembles où elle figure :
REMOVE variable-entité FROM ALL;

c-3) La suppression définitive (espace occupé rendu à la mémoire libre) d'une entité ou d'un ensemble se fait par :

```
DELETE { variable-entité } ;
       {     ensemble     }
```

Cette opération est déjà complexe lorsqu'il s'agit de supprimer un ensemble. Il faut en effet dans ce cas, supprimer également tous les ensembles appartenant à des entités qui se trouvent sur cet ensemble,etc..

c-4) Il est possible de rechercher une entité satisfaisant à certaines conditions, avec l'instruction FIND :

```
FIND var-entité = spécification-d'entité
     [, WITH condition-1]
     [, UNTIL condition-2]
     [, ELSE instruction ] ;
```

c-5) Il est également possible d'effectuer un traitement sur toutes les entités qui satisfont à certaines conditions :

```
FOREACH   var-entité=spécification-d'entité
     [,WITH cond-1] [,UNTIL cond-2] ;
instructions
END;
```

On voit que le programmeur n'a pas à se soucier de la manipulation de pointeurs, ni de la gestion de la mémoire libre.
Remarquons toutefois qu'il n'est possible de mettre une entité dans un ensemble que si une zone de liaison a été prévue dans la déclaration du type de cette entité. Il en est de même pour les ensembles que peut posséder une entité. Il est donc indispensable de prévoir à l'avance le nombre des liaisons dans une entité, ce qui est une limitation importante dans les applications dynamiques et ce qui n'optimise pas la place occupée en mémoire.

d) LEAP [4-ROF]

Ce langage, de développement récent, présente un grand nombre de points originaux.

Les éléments de base sont de trois types :

(1) Elément (de type algébrique réel, entier, tableau etc...)

(2) Triplet : exprime une relation entre trois éléments.

(3) Ensemble : collection non ordonnée de triplets.

Ce langage est de niveau élevé et décharge le programmeur de tout souci quant à l'implémentation des éléments et des liaisons entre eux.

d-1) Ensemble.

a/ On peut définir un ensemble d'éléments par une déclaration :

SET nom-de-l'ensemble

Exemple : SET FILS

b/ On peut ensuite :

mettre un élément dans un ensemble

PUT ALFRED IN FILS

enlever un élément d'un ensemble

REMOVE ALFRED FROM FILS

c/ Opérations sur les ensembles

- Opérations booléennes (réponses : vrai ou faux)

Appartenance d'un élément à un ensemble :

<élément> $\in$ <ensemble>

Un ensemble est-il contenu dans un autre ensemble :

<ensemble> $\subset$ <ensemble>

Egalité de deux ensembles :

<ensemble> $=$ < ensemble >

- Opérations diverses.

Le nombre d'éléments dans un ensemble est donné par :

$\| \text{ensemble}$

etc...

- Affectations.

  On peut affecter un ensemble à un autre ensemble :

  Ex : FILS $\cup$ FRERES $\longrightarrow$ FAMILLE

  (union) (affectation)

- Action sur chaque élément d'un ensemble :

  FOREACH X IN FILS DO instruction
  (utilisant X)

  Remarque : Il n'y a pas de relation d'ordre explicite dans un ensemble.

d-2) Triplet.

Un triplet est une collection ordonnée de trois éléments, et a la forme suivante :

Attribut.objet $\equiv$ Valeur

(Attribut, objet et Valeur sont des éléments).

Création d'un triplet :

MAKE PERE.JULES $\equiv$ JIM

Suppression d'un triplet :

ERASE PERE.JULES $\equiv$ JIM

Il ne s'agit pas ici de créer ou de supprimer des éléments mais de créer ou de supprimer des relations entre ces éléments.

d-3) Liaisons entre triplet et ensemble.

A.B désigne l'ensemble des X tels que A.B $\equiv$ X

A'B " " " " A.X $\equiv$ B

A*B désigne l'ensemble A.B $\cup$ A'B

L'instruction FOREACH permet de définir un traitement à appliquer à tous les éléments qui satisfont certaines relations :

FOREACH FAMILLE.X $\equiv$ JIM AND SEXE.X $\equiv$ MALE DO instruction

d-4) Utilisation du "Hash-coding".

En pratique, mais ceci ne concerne pas l'utilisateur, ce système fait une large utilisation du "Hash-coding" pour ranger les éléments et exprimer les relations entre eux.
Il devient alors possible de répondre rapidement à des questions, parfois complexes, telles que :

A(O) = ? , A(?) = V , ?(O) = V , A(?) = ? etc...

Trois représentations internes sont en fait conservées pour chaque triplet, afin de réduire le temps de réponse à de telles questions.

Ainsi, soit la question A(?)= V, on cherche ici tous les objets O dont l'attribut A a la valeur V.
On prend alors la table des objets O, et on utilise A et V comme des indices (Hash-coding) pour accéder à l'élément de la table qui représente l'objet O cherché.
Il existe en fait une table des Attributs, une table des Objets et une table des Valeurs.

Lorsque des collisions se produisent (du fait de l'utilisation du Hash-coding), les éléments sont rangés sur un anneau.
Ce langage permet de travailler sur plusieurs niveaux de mémoire (mémoire centrale, supports externes), mais le programmeur n'en est pas conscient et ne peut contrôler le processus.

En résumé, il faut surtout apprécier le très haut niveau des opérations fournies par le langage LEAP.

5. LANGAGES SANS STRUCTURES PREETABLIES.

Cette dernière catégorie rassemble des langages qui laissent toute liberté au programmeur pour définir ce dont il a besoin.

Nous nous intéresserons dans cet ordre à

SNOBOL4, PL/1, AED-O,L^6, *1 et DSPS.

Nous verrons ainsi apparaître progressivement des notions de plus en plus précises et des possibilités de plus en plus raffinées, mais d'emploi de plus en plus délicat.

a) SNOBOL4 [4-GPP, 4-GRI]

Bien que plus spécialement créé pour les manipulations de caractères, ce langage présente tout de même quelques possibilités intéressantes dans le domaine des structures.

a-1) Définition d'éléments composés.

La fonction DATA permet de définir de nouveaux types d'éléments (au même titre que les types "entier", "réel", "tableau", etc...).

Exemple : DATA('CLIENT(NOM, ADRESSE, DATE)')

On a ici défini le type "CLIENT". Tout élément de ce type aura trois composants, appelés NOM, ADRESSE, et DATE.

Une fonction de création et des fonctions de référence sont créées implicitement par cette fonction DATA.

Ainsi I = CLIENT('DUPONT', 'IMAG', '051269') a pour effet de créer un élément appelé I, de type CLIENT, et dont les trois composants sont initialisés avec les valeurs 'DUPONT', 'IMAG', et '051269'. La partie droite de l'affectation est une utilisation de la fonction de création.

Les fonctions de référence sont, dans notre exemple, NOM(I), ADRESSE(I) et DATE(I). Elles permettent d'accéder aux contenus des divers composants de l'élément..

a-2) Organisation en structures.

Supposons que nous désirions travailler sur des éléments reliés ensemble de la manière suivante : chaque élément se décompose en trois parties, VALEUR, PRECEDE et SUIT. PRECEDE contient l'adresse de l'élément précédent (et une valeur vide s'il s'agit du premier élément). SUIT contient l'adresse de l'élément suivant (et une valeur vide pour le dernier élément).

La variable PREMIER contient l'adresse du premier élément.
On définit à titre d'exemple la fonction TETE(I), qui ajoute l'élément I en tête de la séquence d'éléments, et la fonction INSERE (I1,I2) , qui insère l'élément I1 juste après l'élément I2 dans la séquence.

```
          DATA ('ELEMENT(VALEUR,PRECEDE,SUIT)')
          DEFINE('TETE(I)')
          DEFINE('INSERER(I1,I2)')
          .
          .
          .
TETE      PRECEDE(I)=
          SUIT(I)=PREMIER
          PRECEDE(PREMIER)=I
          PREMIER=I    :(RETURN)

INSERER   SUIT(I1)=SUIT(I2)
          PRECEDE(I1)=I2
          SUIT(I2)=I1
          PRECEDE(SUIT(I1))=I1     :(RETURN)
```

Notons que le programmeur utilise le nom de l'élément comme s'il s'agissait de l'adresse de cet élément.
Ces quelques possibilités, bien que très élémentaires, sont tout de même intéressantes car d'utilisation aisée et sans soucis quant à l'implémentation réelle. La revers de la médaille est évidemment qu'il est impossible de contrôler la place réellement occupée en mémoire.

b) PL/1. [4-CAV, 4-IBM3]

On trouve en PL/1 quelques caractéristiques intéressantes dans le domaine des structures regroupées sous le nom de "List Processing". Ce nom laisse supposer qu'il s'agit de possibilités de niveau élevé, alors que, comme nous allons le voir, il s'agit en fait d'outils très élémentaires.

L'instruction ALLOCATE permet d'obtenir dynamiquement de la place en mémoire, l'instruction FREE servant à la libérer.

Un attribut particulier, POINTER, décrit une variable de type "adresse" ou pointeur , variable qui peut contenir l'adresse réelle en mémoire d'autres variables (dont la place a été obtenue par ALLOCATE en particulier).

La définition d'un élément composé est un peu plus élaborée que dans SNOBOL, car on peut préciser le type exact de chacun des composants ce qui permet de contrôler la place occupée en mémoire par un tel élément.

Exemple :

```
DECLARE 1 CLIENT, 2 NOM CHAR(8),2 ADRESSE CHAR(25),
                  2 COMMANDE,
                     3 DATE CHAR(8),
                     3 PRIX PICTURE '9999V99' ;
```

CLIENT est ici une variable ordinaire d'un type particulier. Une allocation de mémoire lui est faite lors de l'entrée dans le bloc où se trouve cette déclaration.

On peut également faire de CLIENT un "modèle" qui ne désignera pas une zone unique de mémoire :

Exemple :

```
DECLARE 1 CLIENT BASED(POINTEUR)
   2 NOM CHAR(8), 2 ADRESSE CHAR(25),
   2 COMMANDE, 3 DATE CHAR(8), 3 PRIX PIC'9999V99';
```

Si les informaions relatives à un client sont dans une zone mémoire, dont l'adresse se trouve dans le pointeur P, on peut y faire référence par :

```
     P -> CLIENT
ou   P -> PRIX
```

Cette notation est d'une lecture agréable et est suffisamment explicite.

Pour illustrer ces quelques caractéristiques, reprenons l'exemple développé en SNOBOL dans le paragraphe précédent.

```
            DECLARE PREMIER PØINTER;

            DECLARE  1 ELEMENT BASED(P),
                     2  VALEUR PIC '9999V99',
                     2  PRECEDE PØINTER,
                     2  SUIT PØINTER;

TETE: PROCEDURE(I); DECLARE I POINTER;
  I -> PRECEDE = NULL;
  I -> SUIT = PREMIER;
  IF PREMIER ¬= NULL  THEN PREMIER-> PRECEDE = I;
  PREMIER = I ;
END   TETE;

INSERER: PROCEDURE(I1, I2); DECLARE(I1, I2, I3)PØINTER;
   /* LE BLOC I1 EST MIS APRES LE BLOC I2. */
   I1 -> SUIT = I2-> SUIT;
   I1 -> PRECEDE = I2;
   I2 -> SUIT = I1;
         I3 = I1 -> SUIT;
   I3 -> PRECEDE = I1;
END INSERER;
```

Signalons également une facilité que PL/1 est seul à posséder: on peut définir des "zones" en mémoire, de dimension relativement importante, et demander des allocations de mémoire dans une zone précise indiquée dans l'instruction.

Ceci évite une trop grande dispersion des blocs en mémoire et facilite une éventuelle sortie sur un support externe.

De plus, et c'est là le point original, on peut utiliser des adresses <u>relatives</u> au début d'une zone, ce qui diminue la place occupée en mémoire par les pointeurs.

c) <u>AED-0</u> [4-ROS2]

Sous une syntaxe un peu plus lourde qu'en PL/1, on retrouve les

possibilités précédentes (à l'exception des notions de zone et de variable relative).

Les différents types d'éléments sont plus limités : réel et entier seulement.

Exemple :

```
INTEGER  ELEMENT  §,                          } définition du type
REAL COMPONENT VALEUR §,                      } et du nom de chaque
INTEGER COMPONENT PRECEDE, SUIT §,            } composant;
```

(Le type INTEGER s'applique également à des variables adresses)

```
VALEUR §=§0  §,
PRECEDE §=§ 1 §,
SUIT §=§ 2 §,
```

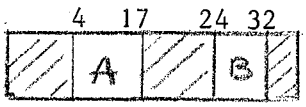
(Ces trois instructions indiquent les positions relatives, en mots, des divers composants dans ELEMENT.)

Afin d'optimiser la place occupée en mémoire et pour compenser le petit nombre de types élémentaires disponibles, on peut définir des portions de mot machine comme éléments.

La définition est assez laborieuse, mais doit être améliorée dans des versions ultérieures du langage.

L'instruction PACK est spécialement adaptée au mot de 36 bits de l'IBM 7044, sur lequel est implémentée la version considérée.

Exemple :



INTEGER COMPONENT A,B §,

Définition de A :

PACK 77777C18, 18, DECREMENT COMPONENTS A §,

masque en octal — décalage nécessaire pour cadrer l'élément à droite — optimisation

Définition de B :

PACK 7770C3,3, SPECIAL COMPONENTS A §,

Il reste encore à indiquer la position relative de A et B dans un groupe de mots :

A §=§ B §=§0 §, (tous les deux dans le premier mot)

Cette instruction originale permet donc de définir une partie quelconque d'un mot et de lui donner un nom. Le système assurera automatiquement les cadrages, décalages et conversions nécessaires lors de l'utilisation de ces composants.

On peut évidemment demander et rendre dynamiquement des blocs en mémoire : Ainsi,

P = FREE(4)§,

demande quatre mots consécutifs et met l'adresse du premier dans P.

FRET(N, P)§ , a pour effet de libérer un bloc de N mots consécutifs, dont l'adresse est dans P.

Reprenons encore une fois l'exemple développé précédemment.

```
INTEGER ELEMENT §,
REAL COMPONENT VALEUR §,
INTEGER COMPONENT PRECEDE, SUIT §,
VALEUR §=§0§, PRECEDE§=§1  §, SUIT§=§2§,

DEFINE PROCEDURE TETE(I) WHERE INTEGER I TOBE
BEGIN   PRECEDE(I) = 0§,
        SUIT(I) = PREMIER §,
        PRECEDE(PREMIER) = I §,
        PREMIER = I§,
END §,
DEFINE PROCEDURE INSERER(I1,I2) WHERE INTEGER I1, I2 TOBE
BEGIN    SUIT(I1)=SUIT(I2) §,
         PRECEDE(I1)=I2 §,
         SUIT(I2) = I1 §,
         PRECEDE(SUIT(I1)) = I1 §,
END §,
```

d) L^6, *1, DSPS.

Ces trois langages sont très proches les uns des autres, et on peut en fait les considérer comme des versions successives d'un même langage, tout au moins en ce qui concerne les idées qu'ils expriment. Avec eux nous atteignons le niveau le plus élémentaire en particulier avec *1 qui est constitué par un ensemble de macro-instructions utilisables dans le langage d'assemblage de l'IBM 360.

Aucun d'eux ne fait de supposition sur l'organisation des divers éléments et ils laissent le programmeur absolument libre de définir une structure en plex adaptée à ses besoins propres.

d-1) L^6. (Bell Telephone Laboratories Low Level Linked List Language.) [4-KNO2]

L^6 se présente comme un langage complet dont la syntaxe est peu explicite, mais suffisamment dense. Les diverses opérations sont codées de la manière suivante :

(opérande-1, opération, opérande-2 [,opérande-3])

La partie opération est constituée de une ou deux lettres ce qui rend la lecture d'un programme assez difficile.

a/ Notion de champ.

Un champ est une portion d'un mot machine quelconque. On le définit par - un nom
- un n° de mot (n° relatif au début d'un groupe de mots).
- un n° de bit gauche, un n° de bit droit.

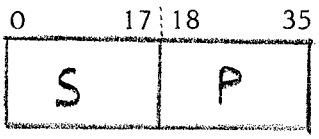
Un champ peut recevoir toute séquence de bits ayant la dimension voulue (des cadrages sont effectués si nécessaires).

Modèle de définition d'un champ :

(n° de mot, Dnom-du-champ, n° de bit gauche, n° de bit droit)

Exemples :

(0, DP, 18, 35)
(0, DS, 0, 17)



Les valeurs définissant un champ peuvent être désignées comme étant le contenu d'un autre champ. La définition de champ est

alors dynamique et peut varier pendant l'exécution.
Un nom de champ peut être soit une lettre A,B,...,Z soit un chiffre 1,..., 9.

b/ Gestion de la mémoire.

L'unité de gestion est le bloc, qui est un groupe de mots mémoire consécutifs. On peut obtenir des blocs de longueur 1,2,4,... jusqu'à 128 mots.

Demande d'un bloc :

(champ, GT, longueur)
demande un bloc de la "longueur" indiquée, dont l'adresse sera mise dans le "champ" indiqué. Cette "longueur" peut être une valeur numérique ou le contenu d'un autre champ.

Libération d'un bloc :

(champ, FR, $\left\{\begin{matrix} 0 \\ \text{champ-1} \end{matrix}\right\}$)

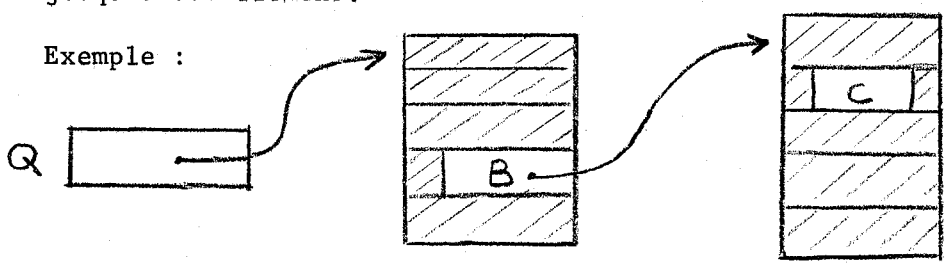
Le bloc dont l'adresse est dans " champ " est libéré; "champ" reçoit ensuite la valeur indiquée en troisième paramètre, ou reste inchangé si 0 est codé.

La gestion de la mémoire est assurée par le système et le programmeur n'a pas à s'en soucier.

c/ Référence à un élément.

Des mots fixes et prédéfinis sont utilisables sans autre précaution, simplement en indiquant leur nom. Pour atteindre un champ qui se trouve dans un certain bloc, le mécanisme est un peu plus compliqué et il faut indiquer le chemin à suivre pour arriver jusqu'à cet élément.

Exemple :



Le champ C est désigné par QBC.
Q est un élément fixe. Son contenu donne l'adresse d'un bloc.
Le champ B de ce bloc contient l'adresse du bloc où C est défini.
Puisque le nom d'un champ comporte au plus un seul symbole, il suffit de concaténer les noms des divers champs à utiliser en partant d'un champ fixe, pour former sans ambiguité une référence à un élément.
Remarquons que les notations parenthésées, rencontrées en SNOBOL et en AED-O par exemple, correspondent à un cheminement inverse (notation fonctionnelle).
Une telle référence peut toujours être utilisée comme une opérande de type "champ" dans les diverses opérations du langage.

d/ Opérations simples.

On trouve des opérations d'affectation, arithmétiques (addition, soustraction, multiplication et division), des opérations de décalages, logiques (intersection, union, etc...), toutes applicables entre deux champs ou entre un champ et une valeur décimale.

Exemple :

addition : (champ,A, $\left\{\begin{array}{l}\text{champ}\\ \text{valeur décimale}\end{array}\right\}$)

affectation : (WB, E, 32767)
mettre dans le champ B du bloc dont l'adresse est dans le champ fixe W, la valeur 32767.

e/ Opérations d'entrée-sortie.

Ces opérations dépendent évidemment de la machine sur laquelle est implémenté le langage, c'est-à-dire une IBM 7094. On précise le nombre de caractères (de 6 bits) à écrire ou à lire.

Exemple : lire 6 caractères et les ranger dans le champ fixe X:

(X, IN, 6)

Diverses opérations de conversion sont également disponibles.

Exemple :

Affecter le contenu de X à WB après remplacement des blancs situés en tête par des zéros :

(WB,BZ,X)

Mettre le contenu de X converti de Décimal en Binaire dans le champ WB :

(WB,DB,X)

f/ Tests élémentaires.

Divers tests permettant de contrôler le déroulement d'un programme sont fournis.

Ex : Le champ X contient-il la valeur 0 ? (X,E,0)

Le contenu de A est-il supérieur au contenu de B? (A,G,B)

g/ Opération non élémentaire.

Signalons ici la présence d'une opération d'un niveau plus élevé que les précédentes : L'impression du contenu d'une séquence de blocs chainés entre eux.

(champ, PL, nom-de-champ)

ou

(champ, PL, nom-de-champ, nombre-maximum)

Le "champ" indiqué permet de trouver le premier bloc de la séquence. L'adresse du bloc suivant se trouve dans le champ désigné par "nom-de-champ".

Cette opération porte donc sur une séquence complète et non plus seulement sur un élément simple. Ce type d'opérations paraît très intéressant. C'est cependant la seule opération de ce genre que l'on peut trouver dans ce langage.

L^6 dispose également de deux opérations de manipulation de piles:

(S,FC,champ) a pour effet d'empiler le contenu du champ indiqué dans une pile de sauvegarde définie par le système.

L'opération inverse

(R,FC,champ)

met dans "champ" le contenu du sommet de la pile de sauvegarde. On peut empiler de la même façon le triplet définissant un champ, ce qui permet de modifier une définition, puis de retrouver la définition précédente.

h/ Ecriture d'un programme.

Une instruction a la forme suivante :

[étiquette] THEN opération-1 opération-2 ...

ou

[Etiquette] { IFANY / IFALL / IFNONE / IFNALL } test-1 test-2... étiquette

Exemple :

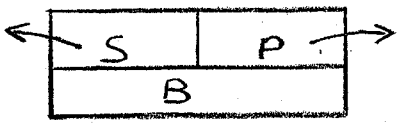
DEBUT THEN (W,GT,2) (WB, E, 32767)

TEST IFNONE (X,E,O) BOUCLE

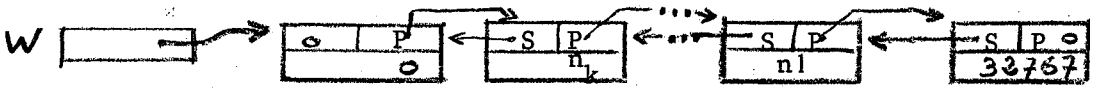
(si le test est satisfait, on va à l'étiquette BOUCLE)

Exemple :

Des nombres sont rangés sur cartes, chacun occupant 6 colonnes. Ils sont tous compris entre 0 et 32767. Chaque nombre sera lu, converti en binaire et rangé dans le champ B d'un bloc de 2 mots.



Les blocs seront reliés entre eux par un double chainage (S indique l'élément suivant et P le précédent). Le premier bloc contiendra le nombre 0 et le dernier 32767.



Le programme correspondant est le suivant ;

```
DEBUT      THEN  (W,GT,2)(WB,E,32767)(S,FC,X)
BOUCLE     THEN  (X,IN,6) (X,BZ,X)
           THEN (W,GT,2,WP)(WPS,P,W)(WB,DB,X)
           IFNONE (X,E,O)BOUCLE
           THEN(X,IN,73) (R,FC,X)FINI
```

d-2) *1 [4-NEH]

Ce langage s'inspire très fortement de L^6 et est constitué par un ensemble de macro-instructions utilisables dans le langage d'assemblage de l'IBM 360.

On retrouve la notion de champ, que l'on définit par :

FIELD nom, position -relative, (n° de bit gauche, n° de bit droit)

Seule "position relative" peut être un symbole, dont la valeur peut changer pendant l'exécution.

Un champ représente un sous-ensemble d'un mot. La notion de BLOCK est introduite afin de pemettre de désigner un groupe de plusieurs mots quelconques.

BLOCK nom,(position - relative, nombre-de-mots)

De même qu'un champ, un BLOCK n'existe que relativement à une adresse qui n'est connue que pendant l'exécution. Notons que la définition est dynamique, les deux quantités qui définissent le BLOCK pouvant être le contenu de symboles.

Des champs fixes qui sont en fait des constantes de type mot, peuvent être définies par :

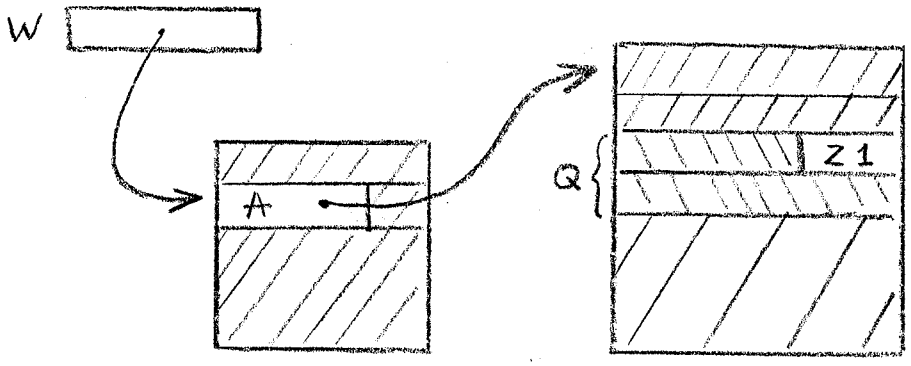
BFIELD nom [,adresse]

L'emplacement d'un tel champ est déterminé à l'assemblage et ne varie donc pas pendant l'exécution.

Le système de référence à un élément est identique à celui utilisé par L^6. On part d'une position fixe et on indique le chemin à parcourir pour arriver à l'élément désiré.

Exemple :

```
BFIELD    W
FIELD     A, 1,(0,23)
BLOCK     Q,(2,2)
FIELD     Z1, 0,(21,26)
```



L'élément Z1 est désigné par WAQZ1.
Afin d'éviter les ambiguités, un nom situé au milieu d'une telle séquence de référence sera écrit entre crochets s'il est composé de plusieurs lettres.

Ainsi avec la définition

```
FIELD     ABC,1,(0,23)
```

la référence précédente prend la forme

W [ABC] QZ1

Z1 est composé de deux symboles, mais il n'y a pas ambiguité car tout nom doit commencer par une lettre.

Les opérations se codent comme en L^6 :

(opérande-1, opération, opérande-2)

On trouve toutes les opérations classiques : affectation, décalages, opérations arithmétiques, logiques, conversions etc..... Des tests sont également disponibles.

Exemple :

```
ETIQ    IFANY   (ABF,=,ABE),THEN,(ABH,+,10),(GOTO, ETIQ2)
        DO      (X,+,3),(Y,*,Z),(ABCF,←,AG)
```

Des opérateurs imagés tels que ← pour l'affectation, ←o pour l'affectation d'adresse, →→ pour les décalages à droite sont théoriquement prévus, mais sont malheureusement remplacés dans l'implémentation par les symboles disponibles tels que _ (souligné), _ @ et # # .

On peut bien entendu demander dynamiquement de la mémoire :

ACQUIRE quantité, champ

et la libérer :

RELINQ quantité, champ

Il faut encore signaler que l'ensemble de ce langage fait l'objet d'une description syntaxique complète sous la forme de Backus, ce qui facilite bien sa compréhension. On peut regretter que l'emploi du langage d'assemblage devienne assez difficile, et en particulier l'emploi des registres. Des versions ultérieures doivent toutefois améliorer ce point.

Exemple :

Chercher le dernier élément d'une séquence :

```
          BFIELD  LISTE          } définition des constantes LISTE
          BFIELD  T              }                           et T
          FIELD L,2,(8,31)         définition du champ L

          DO (T,←o,LISTE)          contenu de LISTE mis dans T.
BOUCLE    IF (¬ ,TL),THEN,(GOTO,FINI) si TL = 0, on termine

          DO (T,←o,TL),(GOTO,BOUCLE) contenu du champ L du bloc T mis
                                     dans T, puis retour à BOUCLE.
FINI      ...
```

d-3) DSPS (Data Structure Programming System.) [4-EVA]

Conçu comme une extension des deux langages précédents, ce langage présente les mêmes possibilités que *1, auxquelles s'ajoute un système de pagination dont il faut dire quelques mots.

En plus des entités de base telles que champ et bloc, on trouve la notion de page. Ces pages sont les unités de mémoire qui pourront faire l'objet d'échanges entre la mémoire centrale et un support externe Plusieurs types de pages sont à la disposition du programmeur.

Une page permanente est une page désignée comme complète par le programmeur. Elle n'est plus modifiable. Des pages temporaires servent d'espace de travail et peuvent devenir des pages permanentes une fois complétées. La distinction entre pages temporaire et permanente réside donc dans la possibilité de subir ou non des modifications.

La présence d'un système de recherche d'information s'appuyant sur des combinaisons de clés conduit à deux autres types de pages. On distinguera des pages globales qui sont cataloguées dans le système de recherche et qui sont donc accessibles par des clés et des pages locales dont les clés ne figurent pas dans le système global et qui servent en fait de complément aux pages globales. Chaque page globale peut pointer sur une ou plusieurs pages locales. L'ensemble d'une page globale et de ses pages locales associées constitue alors une masse d'information formant un tout.

Le programmeur dispose donc avec DSPS d'un certain contrôle sur l'utilisation d'une mémoire à deux niveaux. Cette caractéristique est très originale et méritait d'être signalée.

La syntaxe de DSPS ne présentant que des différences infimes avec celle de *1, il ne paraît pas utile d'en donner un exemple.

B. REALISATION ET PRESENTATION D'UN SYSTEME DE MACRO-INSTRUCTIONS.

De l'étude cette gamme de langages, on peut dégager aisément l'existence de deux niveaux assez éloignés :

l'un élevé et plutôt "mathématique" où l'on manipule des entités comme des objets irréels, en leur faisant subir des opérations abstraites;

l'autre plus terre à terre et "pragmatique", où l'on se préoccupe d'abord de l'intendance, où le moindre détail doit être prévu, justifié et organisé et où une construction quelle qu'elle soit ne peut être que le fruit d'un dur labeur !

Bien sûr, il est plus satisfaisant de fournir un support de niveau élevé, mais même si tout le monde arrive à un accord sur les grands principes, chacun sait que leurs applications effectives sont l'objet de discussions sans fin.

Aussi, plutôt que de pénétrer à notre tour dans ces discussions, nous préférons nous placer au niveau le plus bas, tel qu'il s'est dégagé de l'étude précédente.

Partant de ce niveau, nous essayerons de voir jusqu'où on peut aller sans prendre d'options contraignantes.

Puisque nous voulons partir du niveau le plus élémentaire, l'utilisation du langage machine doit rester possible afin de laisser l'utilisateur libre de développer les outils supplémentaires qu'il jugera nécessaires.

Nous sommes donc conduits à fournir des facilités utilisables en langage machine. On peut envisager ceci de deux façons : soit une bibliothèque de sous-programmes, soit un ensemble de macro-instructions.

Un sous-programme présente l'inconvénient d'imposer une syntaxe rigide et peu significative. Il suffit pour s'en rendre compte d'examiner le support de programmation graphique, connu sous le nom de GSP. On y trouve de nombreux sous-programmes, ayant jusqu'à 17 paramètres dont certains sont facultatifs, mais dont l'ordre reste impératif.

Les macros instructions permettent une syntaxe plus explicite (par l'utilisation de paramètres à mots clefs) et plus souple (on peut omettre des paramètres, changer leur ordre...).

La nécessité d'avoir cependant un système dynamique, en particulier pour laisser les définitions évoluer pendant l'exécution, est satisfaite par la présence de sous-programmes. Mais ceux-ci sont appelés par les macro-instructions, et ne sont pas accessibles à l'utilisateur.

B.1 Définition des éléments de base.

Il y a deux éléments de base: le bloc et le champ.

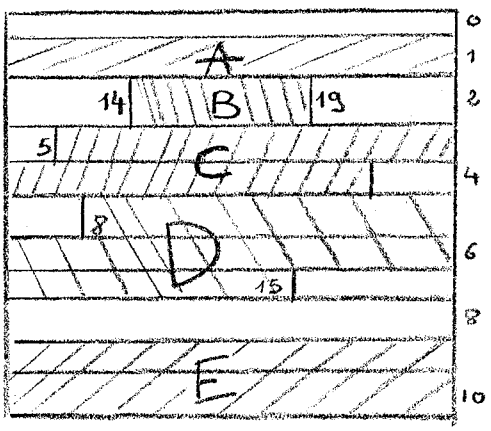
a) BLOC : groupe de N mots mémoire consécutifs.

BLOC nom, taille
(nombre de mots machine)

"nom" désigne non pas un bloc particulier, mais un type de bloc. Nous en verrons l'utilisation plus loin.

b) CHAMP : masque permettant de désigner une partie d'un bloc quelconque. Cette partie est une séquence continue de bits, c'est-à-dire qu'un champ ne peut être composé de deux parties disjointes.

Exemples de champs :



La définition d'un champ peut prendre les formes suivantes :

① Champ qui occupe tout un mot :

CHAMP nom, numéro-du-mot
(n° relatif dans le bloc, commence à 0)

Exemple : CHAMP A,1

② Champ qui occupe complètement plusieurs mots :

CHAMP nom, n°-du-premier-mot, nombre-de mots

Exemple : CHAMP E,9,2

③ Champ sous-ensemble d'un mot :

CHAMP nom, n°du-mot, bit-gauche, bit-droit
(compris entre 0 et 31)

Exemple : CHAMP B,2,14,19

④ Champ qui occupe partiellement plusieurs mots :

CHAMP nom, n°-du-premier-mot, bit-gauche, bit-droit, nombre-de-mots-occupés

Exemple : CHAMP D,5,8,15,3

(Dans ce dernier cas, si "nombre-de-mots" est > 2 on impose, pour des raisons d'efficacité, que le champ soit cadré sur des frontières d'octets, c'est-à-dire que bit-droit peut avoir l'une des valeurs suivantes : 7,15,23 ou 31 et bit-gauche l'une des valeurs : 0,8,16 ou 24.)

Nature des paramètres :

"nom" est un identificateur classique ayant au plus 8 caractères. Les autres peuvent être :

- des constantes décimales (ex: 12)
- des litéraux de type mot (ex: =F'12')
- des symboles de type mot (ex: DOUZE)

dans ce dernier cas, c'est la valeur du symbole au moment de l'exécution, qui sera utilisée.

La présence possible de symboles implique évidemment que la définition du champ ne peut être complétée que pendant l'exécution du programme, donc de manière dynamique.

B.2 Modification des élément de base.

La macro-instruction MODIFCHA permet de modifier la définition d'un champ pendant l'exécution. Les paramètres sont les mêmes que ceux de la macro CHAMP et ont mêmes significations.

Cependant MODIFCHA ne peut modifier qu'un champ défini préalablement par CHAMP.

Cette macro impose en fait une certaine discipline à l'utilisateur et évite que celui-ci ne donne par inattention un même nom à deux champs différents. La macro CHAMP ne peut être utilisée qu'une seule fois pour un nom de champ, et il est donc impératif d'utiliser MODIFCHA pour changer la définition d'un champ déjà connu.

B.3 Demande dynamique d'espace en mémoire.

Une demande de mémoire peut se faire par :

$$\text{ALLOUER} \quad \left\{ \begin{array}{l} \text{nom-de-bloc} \\ \text{nombre-d'octets} \end{array} \right\} , \text{ adresse}$$

"nom-de-bloc" est le nom d'un bloc défini par la macro BLOC. Un bloc joue en fait le rôle d'unité d'allocation de mémoire.

"adresse" est le nom d'un mot dans lequel sera rangée l'adresse de la zone obtenue.

Les zones inutiles peuvent ensuite être libérées par :

$$\text{LIBERER} \quad \left\{ \begin{array}{l} \text{nom-de-bloc} \\ \text{nombre-d'octets} \end{array} \right\} \quad , \text{ adresse}$$

Toute la gestion de la mémoire se fait, du point de vue utilisateur, par ces deux macro-instructions. C'est donc ici que se placerait la liaison avec un système complet de gestion de mémoire si l'importance du travail le nécessitait.

B.4 Référence mémoire. Notion de séquence.

Dans la définition d'un champ, on indique toujours la position relative de ce champ par rapport au premier mot d'un bloc, dont l'adresse est inconnue à ce moment.

Afin de pouvoir accéder à une information effective, il faut fixer le champ, c'est-à-dire donner l'adresse du bloc auquel on veut l'appliquer.

Nous appelerons séquence la suite d'éléments suivants:

$$(\text{élément-fixe}, \quad \left[\begin{array}{l} \text{nom-de-champ} \\ \text{intermédiaire,} \end{array} \right] \ldots \quad \text{nom-de-champ-terminal})$$

"élément-fixe" peut être soit un symbole, soit un numéro de registre. Le contenu de cet élément est interprété comme l'adresse d'un bloc qui servira alors de base pour le champ qui suit.

On peut ainsi accéder au contenu du premier champ intérmédiaire. Ce contenu est à nouveau interprété comme l'adresse d'un bloc, qui sert à son tour de support au champ suivant, et, ainsi de suite, jusqu'au champ terminal.

C'est ce dernier qui est l'élément intéressant de la séquence; on peut alors soit examiner le contenu de ce champ, soit y mettre une autre valeur.

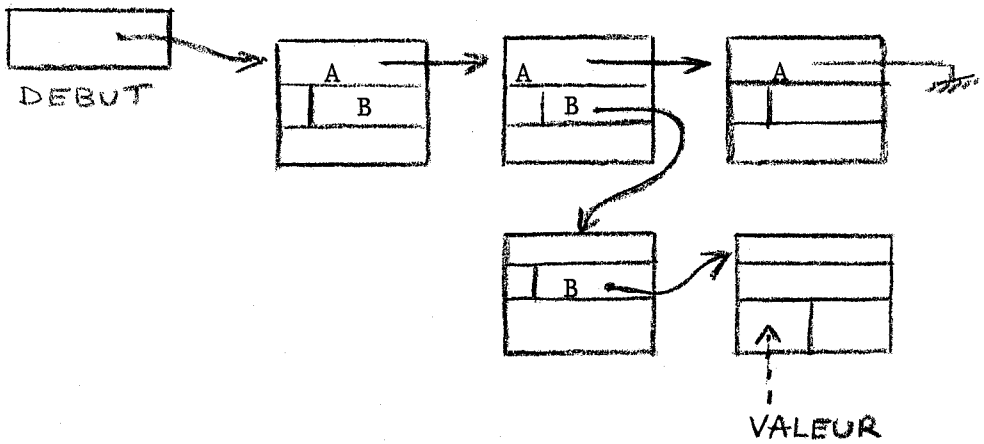
Exemple : Soient les définitions :

CHAMP A,0

CHAMP B,1,8,31

CHAMP VALEUR,2,0,17

Soient les blocs suivants :



La séquence suivante :

(DEBUT,A,B,B,VALEUR)

désigne le champ VALEUR indiqué.

Remarque : Une séquence doit comporter au moins deux éléments : un élément fixe et un nom de champ terminal.

B.5 Opérations élémentaires.

a/ Ranger.

RANGER valeur,séquence

Il s'agit ici de mettre la valeur indiquée dans le champ terminal de la séquence.

"valeur" peut être -un registre
- un symbole (de type mot)
- un litéral (de type mot)

Exemple :

```
RANGER  DOUZE,(DEBUT,A,B,B,VALEUR)
RANGER =F'12',(DEBUT,A,B,B,VALEUR)
RANGER  3,(DEBUT,A,B,B,VALEUR)
```

On range ici respectivement le contenu du symbole DOUZE, la constante 12 et le contenu du registre 3 dans le champ VALEUR désigné par la séquence.

b/ Charger.

CHARGER endroit, séquence

"endroit" peut être un symbole ou un registre. Ce symbole, ou ce registre, reçoit le contenu du champ terminal de la séquence.

c/ Cadrages.

Dans tous les cas des cadrages sont assurés afin de remplir correctement les diverses zones concernées.

Une quantité trop longue sera tronquée à gauche afin de tenir dans une zone plus courte. Une quantité courte sera cadrée à droite dans la zone réceptrice.

Ainsi, dans tous les cas, on cadre toujours à droite puis on tronque à gauche ou on allonge à gauche avec des zéros selon les cas.

d/ Opérations indexées.

Afin de faciliter des transferts d'informations plus volumineuses, les deux macros précédentes existent aussi dans une version indexée. Elles se distinguent des précédentes par le I, qui termine leurs noms.

RANGERI adresse,séquence
CHARGERI adresse,séquence
La valeur du symbole, ou du registre, indiqué comme "adresse"

est interprétée comme étant l'adresse d'une zone.

RANGERI prend ce qui se trouve dans cette zone et le met dans le champ terminal de la séquence. (on prend dans la zone juste ce qu'il faut pour remplir le champ).

CHANGERI prend le contenu du champ terminal et le range dans la zone indiquée, en le cadrant à gauche.

B.6. Opérations non élémentaires.

Les opérations que nous venons de décrire sont des opérations de base,qui, combinées avec les instructions usuelles du langage d'assemblage, donnent des possibilités simples mais pratiques.

C'est à ce point que, sauf rares exceptions, les langages sans structures préétablies s'arrêtent. En restant dans l'esprit d'une construction en plex, on peut cependant fournir des opérations de niveau plus élevé, capable en particulier de travailler sur des ensembles de blocs.

Nous dirons que des blocs, en nombre fini, forment un ensemble lorsqu'il est possible, à partir d'un bloc particulier que nous appelerons tête (ou nom), de retrouver tous les blocs, en utilisant toujours le même mécanisme pour passer d'un bloc au bloc suivant.

Ce mécanisme pourrait être, dans un exemple précis, l'exploitation du fait que le champ A de chaque bloc contient l'adresse du bloc successeur.

On ne demande pas de pouvoir passer d'un bloc quelconque à tous les autres blocs de l'ensemble. Un bloc a un, et un seul, successeur pour le mécanisme considéré. Nous distinguerons deux cas :

- il y a un bloc qui n'a pas de successeur. Nous dirons que l'ensemble forme une chaine.
- tout bloc a un successeur. L'ensemble sera alors appelé anneau.

Dans certains cas, l'ensemble pourra être constitué par la donnée de deux mécanismes, permettant d'accéder d'une part au successeur, d'autre part au prédécesseur d'un bloc donné.

Instruction INSERER.

La première opération que l'on peut imaginer, est l'insertion ou l'adjonction d'un bloc dans un ensemble existant.

Etant donnée l'adresse du premier bloc de cet ensemble, il nous faut encore le nom du champ qui, dans chaque bloc, contient l'adresse du successeur de ce bloc. Le type (CHAINE ou ANNEAU) de cette liaison doit également être précisé, car certaines manipulations en dépendent.

Si les blocs sont reliés entre eux par une seconde liaison, on précisera de même le nom du champ et le type qui définissent la liaison.(Cette liaison permet de trouver le prédécesseur d'un bloc donné;)

Enfin il faudra indiquer où doit se faire l'insertion : après ou avant un élément donné de l'ensemble, ou encore en TETE ou en QUEUE de cet ensemble. Cette position est relative à la première des deux liaisons indiquées.

Le modèle de cette instruction est le suivant :

$$\left[\text{étiquette}\right] \quad \text{INSERER} \quad \left\{\begin{array}{l}\text{mot}\\ \text{registre}\\ \text{(registre)}\end{array}\right\}, \ \text{DANS} = \left\{\begin{array}{l}\text{mot}\\ \text{registre}\\ \text{(registre)}\end{array}\right\},$$

$$\text{LIAISON} = (\text{nom-de-champ} \left[,\text{nom-de-champ}\right]),$$

$$\text{TYPE} = \left(\left\{\begin{array}{l}\text{ANNEAU}\\ \text{CHAINE}\end{array}\right\}\left[,\left\{\begin{array}{l}\text{ANNEAU}\\ \text{CHAINE}\end{array}\right\}\right]\right.,$$

$$\left\{\begin{array}{l}\text{PLACE} = \left\{\begin{array}{l}\text{TETE}\\ \text{QUEUE}\end{array}\right\}\\[2ex] \text{AVANT} = \left\{\begin{array}{l}\text{mot}\\ \text{registre}\\ \text{(registre)}\end{array}\right\}\\[2ex] \text{APRES} = \left\{\begin{array}{l}\text{mot}\\ \text{registre}\\ \text{(registre)}\end{array}\right\}\end{array}\right\}$$

Notons que tous les paramètres qui désignent un bloc le font par l'intermédiaire d'un registre ou d'un mot qui contient l'adresse du bloc considéré.

L'effet de cette macro-instruction est, bien entendu, de connecter un nouveau bloc à un ensemble de blocs préexistants. Si le paramètre DANS contient une valeur nulle au moment de l'exécution, ceci signifie que l'ensemble considéré est vide et que l'on y place le premier élément.

Instruction ENLEVER.

Complément naturel de l'opération précédente, l'instruction ENLEVER permet de déconnecter un bloc d'avec un ensemble d'autres blocs. Les paramètres ont la même signification que dans le cas précédent, l'indication de position ayant évidemment disparu.

Le modèle est donc le suivant :

$$[\text{étiquette}] \quad \text{ENLEVER} \left\{\begin{array}{l}\text{mot}\\ \text{registre}\\ \text{(registre)}\end{array}\right\} , \text{DE} = \left\{\begin{array}{l}\text{mot}\\ \text{registre}\\ \text{(registre)}\end{array}\right\} ,$$

$$\text{LIAISON} = (\text{nom-de-champ} \; [,\text{nom-de-champ}\;]),$$

$$\text{TYPE} = \left(\left\{\begin{array}{l}\text{ANNEAU}\\ \text{CHAINE}\end{array}\right\} \left[, \left\{\begin{array}{l}\text{ANNEAU}\\ \text{CHAINE}\end{array}\right\} \right] \right)$$

Instruction POUR.

Elle permet d'appliquer un certain sous-programme à tous les blocs d'une certaine séquence chainée.

modèle :

```
[étiquette]   POUR   mot-1,
                     INIT= { mot-2    } ,CHAMP=nom-de-champ
                           {   (reg)  }
                           { registre }
                     [,NOMBRE= { mot-3   } ]
                               {  (reg)  }
                               { décimal }
                      ,FAIRE= {symbole} [,RETOUR= { reg   } ]
                              {  (reg)  }          {  (reg) }
```

"Mot-1" est la variable d'itération. Elle contiendra l'adresse du bloc en cours de traitement.

"INIT" désigne l'adresse du premier bloc de la séquence.

"NOMBRE" indique le nombre maximum de blocs à considérer.

"champ" précise le nom du champ qui, dans chaque bloc, contient l'adresse du bloc suivant.

"FAIRE" désigne le sous-programme soit par le nom "symbole" de ce dernier, soit par l'adresse contenue dans le registre indiqué entre parenthèses. "RETOUR" spécifie le registre de retour utilisé par le sous-programme FAIRE. La valeur 14 est prise par défaut.

On arrête l'itération soit lorsqu'on a traité le "NOMBRE" de blocs indiqué , soit lorsque le contenu du "champ" indiqué est nul ou égal à la valeur initiale "INIT".

CONCLUSION

Le développement de ces macros-instructions nous a permis de voir de près quelques uns des problèmes pratiques que l'on rencontre dans ce genre de travail et nous a permis d'étudier les raisons de certaines limitations présentes dans les langages de programmation étudiés.

Tout d'abord le choix de déclarations dynamiques de champ ne nous permet pas d'optimiser les instructions générées au niveau de l'assemblage. Nous sommes obligés d'avoir recours à des sous-programmes qui traitent tous les cas possibles, au moment de l'exécution.

Ceci a des répercussions sur les autres macro-instructions, pour lesquelles il est également difficile d'optimiser le code généré.

Aussi, il faut bien voir que le fait de ranger des informations en mémoire en des emplacements sans rapport avec les frontières de mots et d'octets nécessite de nombreuses instructions afin d'assurer les cadrages qui permettent aux instructions classiques de fonctionner. Ce phénomène est lié à la technologie qui fait que, pour une machine donnée, la quantité minimale d'information manipulée n'est pas le bit, mais un certain groupe de bits (mot, octet...).Ce groupe de bits ne peut d'ailleurs être choisi arbitrairement et est soumis à certaines contraintes de frontières. Ainsi une quelconque séance de 8 bits ne constitue pas forcément un octet. Les instructions courantes traitent cette quantité minimale, et parfois même seulement un sur-multiple de cette quantité. Il s'ensuit que travailler au niveau du bit requiert de combiner plusieurs instructions, de manière parfois complexe.

Les opérations élémentaires sont assez faciles à choisir. Ce sont des extensions des opérations de base, permettant de s'affranchir au moins partiellement de ces contraintes de frontières.

Les opérations les plus élaborées sont naturellement les plus intéressantes, mais la difficulté réside dans leur choix. Nous voulons conserver le contrôle des blocs et des informations qu'ils contiennent, mais lorsque le niveau d'une opération augmente, le contrôle exercé par le programmeur tend à diminuer.

Les instructions INSERER, ENLEVER et POUR nous semblent être assez fondamentales et assez puissantes pour servir dans la plupart des cas.

EXEMPLE 1

```
STMT   SOURCE STATEMENT

   1           PRINT OFF
 625           SAVE  (14,12)
 628           BALR   12,0
 629           USING *,12
 630           ST     13,SAUVE+4
 631           LA    10,SAUVE
 632           ST    10,8(13)
 633           LR    13,10
 634 *
 635 *
 636 *
 637 *  INITIALISATION:
 638           INIT
 779 *
 780 *  DEFINITIONS DE CHAMPS:
 781           CHAMP   A,0
 796           CHAMP  B,1
 811           CHAMP  C,2,0,9
 842           CHAMP  D,2,10,20
 873           CHAMP  E,2,11,31
 904 *
 905 *  DEFINITION DE BLOC:
 906           BLOC  BIDULE,4
 909 *
 910 *
 911 *  CREATION DE DEUX BLOCS :
 912           ALLOUER BIDULE,ELEM1
 920           ALLOUER BIDULE,ELEM2
 928 *
 929 *
 930 *  INSERTION DES DEUX BLOCS
 931 *  DANS UN ANNEAU AVEC LIAISONS AVANT ET ARRIERE :
 932   INSERER ELEM1,DANS=ANNEAU,LIAISON=(A,B),TYPE=(A,A),PLACE=TETE
1011 *
1012   INSERER ELEM2,DANS=ANNEAU,LIAISON=(A,B),TYPE=(A,A),PLACE=QUEUE
1096 *
1097 *  EXECUTION DU SOUS-PROGRAMME DISPLAY
1098 *  SUR TOUS LES  ELEMENTS DE L'ANNEAU :
1099   POUR  P,INIT=ANNEAU,CHAMP=A,FAIRE=DISPLAY
1126 *
1127           ALLOUER BIDULE,ELEM3   CREER TROISIEME BLOC
1135 *
1136   INSERER ELEM3,DANS=ANNEAU,LIAISON=(A,B),TYPE=(A,A),APRES=ELEM1
1237 *
1238   POUR  P,INIT=ANNEAU,CHAMP=A,FAIRE=DISPLAY
1265 *
1266   ENLEVER ELEM3,DE=ANNEAU,LIAISON=(A,B),TYPE=(A,A)
1366 *
1367   POUR  P,INIT=ANNEAU,CHAMP=A,FAIRE=DISPLAY
1394 *
1395   INSERER ELEM3,DANS=ANNEAU,LIAISON=(A,B),TYPE=(A,A),AVANT=ELEM2
1475 *
1476   POUR  P,INIT=ANNEAU,CHAMP=A,FAIRE=DISPLAY
1503 *
1504 *  FIN DU PROGRAMME :
```

EXEMPLE 1

```
STMT   SOURCE STATEMENT

1505           L      13,SAUVE+4
1506           RETURN (14,12)
1509 *
1510 *  SOUS-PROGRAMME DISPLAY :
1511 *
1512 DISPLAY   DISPLAY  P,12,IND
1665           DISPLAY ANNEAU,20
1679           BR     14          SORTIE DU SOUS-PROGRAMME
1680 *   CONSTANTES :
1681 CHAINE    DC     F'0'
1682 ANNEAU    DC     F'0'
1683 ELEM1   DS    F
1684 ELEM2   DS   F
1685 ELEM3     DS     F
1686 P         DS     F
1687 SAUVE     DS       18F
1688           PRINT OFF
```

EXEMPLE 2.

```
STMT   SOURCE STATEMENT

  1            PRINT  OFF
430 *   CONVENTIONS DE LIAISONS:
431            SAVE  (14,12)
434            BALR  12,0
435            USING  *,12
436            ST    13,SAUVE+4
437            LR    10,13
438            LA    13,SAUVE
439            ST    13,8(10)
440 *
441 *
442            INIT
583            OPEN  (IMPR,OUTPUT)
589 *
590            CHAMP SUIVANT,ZERO,8,31
621 *   LE CHAMP LONG CONTIENDRA LA LONGUEUR DU TEXTE
622 *   EXPRIMEE EN MOTS (4 CARACTERES PAR MOT).
623            CHAMP LONG,ZERO,ZERO,7
654            CHAMP TEXTE,1,5
672 *
673 * DEMANDE UN BLOC:
674            ALLOUER 24,TETE
682            RANGER ZERO,(TETE,SUIVANT)
692            RANGER UN,(TETE,LONG)
702 * PRENDRE 4 CARACTERES DANS L'ALPHABET:
703            BAL   14,REMPLIR
704 * NOUVEAU BLOC:
705            ALLOUER 24,NOUVEAU
713 * LIAISON AVEC BLOC PRECEDENT:
714            RANGER TETE,(NOUVEAU,SUIVANT)
724            MVC   TETE(4),NOUVEAU
725            RANGER CINQ,(TETE,LONG)
735 * PRENDRE 20 CARACTERES DE L'ALPHABET:
736            BAL   14,REMPLIR
737 *
738            PUT   IMPR,PAGE
743 *
744 * SORTIR CONTENU DE TOUS LES BLOCS:
745            POUR  P,INIT=TETE,CHAMP=SUIVANT,FAIRE=SORTIR,RETOUR=9
772 * FIN:
773            CLOSE  (IMPR)
779            L     13,SAUVE+4
780            RETURN (14,12)
783 *
784 REMPLIR    CHARGER NOUVLONG,(TETE,LONG)
794            MODIFCHA TEXTE,1,NOUVLONG
811            RANGERI ALPHABET,(TETE,TEXTE)
825            BR    14   RETOUR AU PROGRAMME PRINCIPAL.
826 *
827 SORTIR     CHARGER NOUVLONG,(P,LONG)
837            MODIFCHA TEXTE,1,NOUVLONG
854 *   PREPARER ZONE DE SORTIE:
855            MVI   LIGNE+1,C' '
856            MVC   LIGNE+2(119),LIGNE+1
857 *
```

EXEMPLE 2

```
STMT    SOURCE STATEMENT

 858           CHARGER I ZONE,(P,TEXTE)
 872           PUT    IMPR,LIGNE
 877           BR     9
 878 ALPHABET  DC     A(ALPHA)
 879 ALPHA     DC     C'ABCDEFGHIJKLMNOPQRSTUVWXYZ'

 880 ZERO      DC     F'0'
 881 UN        DC     F'1'
 882 CINQ      DC     F'5'
 883 TETE      DS     F
 884 NOUVEAU   DS     F
 885 NOUVLONG  DS     F
 886 P         DS     F
 887 ZONE      DC     A(LIGNE+1)
 888           PRINT OFF
```

IIIe P A R T I E

APPLICATION AUX TRAITEMENTS GRAPHIQUES

3ième PARTIE

APPLICATION AUX TRAITEMENTS GRAPHIQUES

L'importance de la notion de représentation interne est, comme nous l'avons vu, très grande dans les problèmes graphiques. Ceci conduit tout naturellement au développement d'outils destinés à faciliter le travail de l'utilisateur dans ce domaine particulier.

Le langage spécialisé reste l'outil le plus courant, même si l'unanimité ne s'est pas faite sur un langage particulier. Le développement d'une représentation interne se fait alors indépendamment de l'image, et la relation entre cette dernière et sa structure interne reste difficile à percevoir. Aussi nous avons voulu essayer de faire apparaître à la fois l'image et sa représenta tion interne.

Il n'est naturellement pas question de montrer tous les détails (pointeurs, etc...) d'une représentation en Plex. Ceci conduirait à un schéma extrêmement touffu, et tout à fait incompréhensible. Nous voulons donc faire apparaître, non pas la représentation interne exacte, mais une interprétation de celle-ci.

Cette interprétation doit conduire à un schéma suffisamment simple pour que l'on puisse envisager de le compléter ou de le modifier dans le cadre d'un travail conversationnel. Il s'agit ici de maintenir le temps de réponse de l'utilisateur dans les limites raisonnables.

Cette interprétation doit cependant rester assez complète afin de donner une vue d'ensemble de la situation.

Une étude du système GRAPHIC-II [2-CHP] nous a suggéré l'utilisation d'une structure en forme de graphe, dont nous étudierons plus complètement les possibilités dans le chapitre suivant.

Nous présenterons ensuite le système qee nous avons développé pour construire simultanément une image et sa représentation interne. Après avoir décrit ses possibilités, nous suggèrerons quelques extensions et nous essayerons de voir ce que l'on peut attendre d'un tel système.

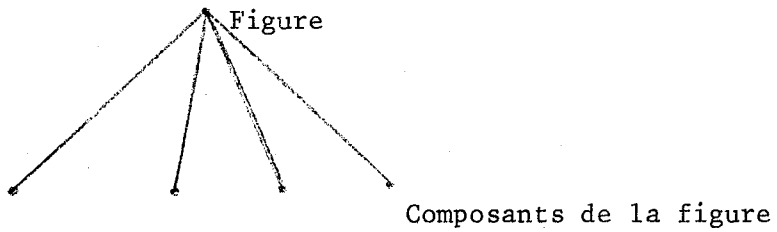
A. REPRESENTATION EN GRAPHE.

La notion de graphe est largement répandue et fait l'objet de théories assez développées. Ce ne sont toutefois pas ces théories qui nous intéressent, mais plutôt l'aspect "simple" d'un graphe. Dans les formes les plus simplifiées, telle que la forme arborescente par exemple, il semble qu'un graphe soit assez expressif et que sa signification puisse être perçue rapidement. Cette représentation peut donc s'adapter assez bien à un travail intéractif.

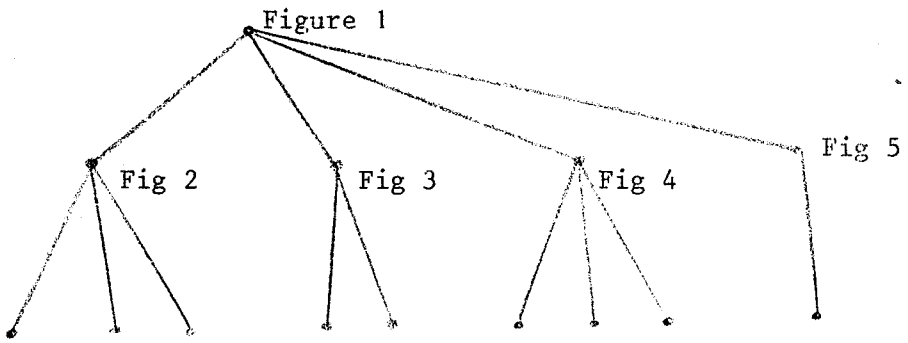
1. Relations entre une image et un graphe.

Essayons de voir dans quelle mesure une image est susceptible d'une représentation en forme de graphe, et quelles sont les conventions qu'il faut introduire.

Les relations entre une figure et les éléments composant cette figure s'expriment immédiatement sous une forme arborescente :



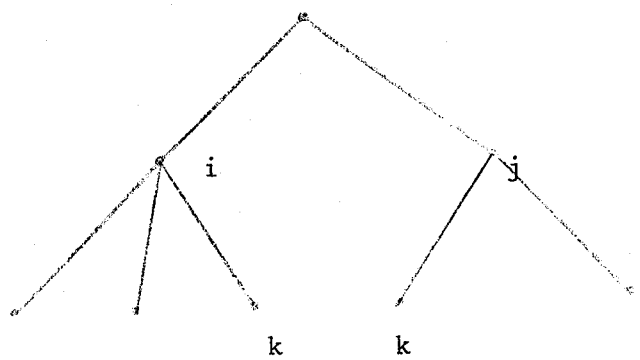
Ces composants peuvent être à leur tour des figures et la forme arborescente se ramifie encore :



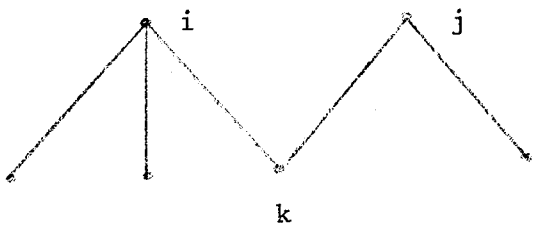
La hiérarchie existant entre les diverses figures et les composants de ces figures sont ainsi très bien exprimées.

On peut donner une autre interprétation qui est plus proche des réalités de la programmation : la branche reliant la figure i à la figure j correspond à un appel de la figure j (exactement comme on appelle un sous-programme). La figure i est ainsi constituée par plusieurs appels d'autres figures.

Il peut arriver que deux figures i et j appellent toutes deux une même figure k. Une première solution consisterait à mettre deux exemplaires de la figure k dans la représentation, de façon à respecter la forme arborescente :



Cependant, il s'agit de la même figure k et, en particulier, il n'existe qu'une représentation interne de cette figure. Aussi, nous préférons une deuxième solution dans laquelle la figure k sera représentée par un seul sommet, ce sommet étant le point extrêmité de plusieurs branches :



Nous reviendrons plus loin sur l'interprétation à donner à ce cas, en particulier avec l'introduction de paramètres.

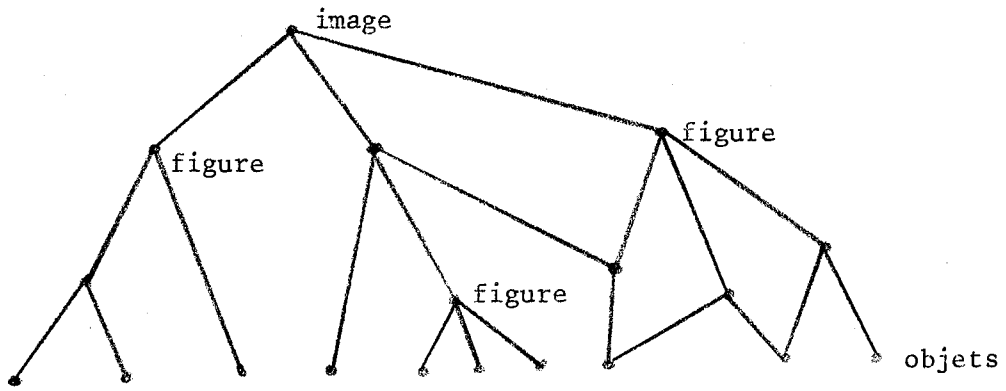
Il faut noter que ce partage de composants nous éloigne d'une forme arborescente et nous emploierons donc dans la suite le terme de graphe, sans

chercher à préciser plus complètement cette notion.

Une image est une collection de figures. Chaque figure est elle-même constituée de sous-figures, et ainsi de suite...

Aucun de ces éléments ne correspond, jusqu'ici, à un tracé effectif. Pourtant, à l'origine, nous ne disposons que d'éléments de tracé, tels que points, vecteurs etc...

Tous ces éléments, que nous appelerons objets graphiques, vont former les extrêmités de notre graphe :



Nous appelerons également objets, ou feuilles, ou extrêmités du graphe, tout ce qui correspond à un tracé effectif et qui n'est pas détaillé dans cette représentation.

2. Caractéristiques physiques d'une figure.

Nous avons souligné l'analogie d'un appel de figure et d'un appel de sous-programme. Ce dernier prévoit cependant le passage de paramètres. Un appel de figure peut également dépendre de paramètres, tels que :

- position d'affichage (translation)
- facteur d'échelle (homothétie)
- position angulaire (rotation)

La présence de ces paramètres est essentielle si l'on veut exploiter complètement le travail nécessité par la construction d'une figure à partir

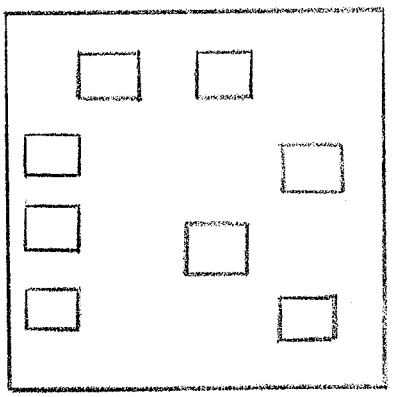
d'objets de base; il sera plus facile de réutiliser une figure pour en construire d'autres s'il est possible de transformer quelque peu la figure de base.

Dans notre graphe, les paramètres de l'appel sont en fait les valeurs de la branche qui représente un appel de figure.

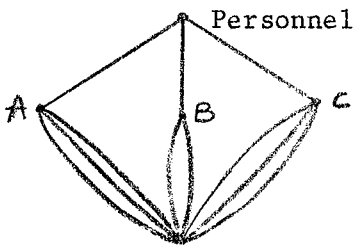
3. Signification d'une figure.

Dans notre graphe, une figure correspond, bien entendu, à un certain tracé, obtenu après exploration du graphe. Mais on peut attribuer une signification différente au niveau constitué par une figure. Il peut s'agir d'un niveau de signification ou de regroupement, qui permet d'exprimer des relations non apparentes sur l'image. La présence de ces relations cachées apparaîtra plus clairement dans l'étude de quelques exemples.

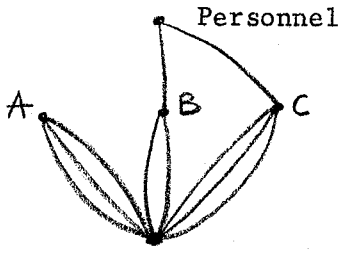
Premier exemple : Considérons le dessin suivant :



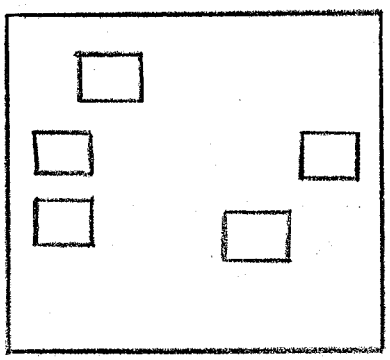
Plusieurs rectangles identiques sont affichés sur l'écran. Nous supposerons qu'ils constituent une représentation schématique des bureaux dans une entreprise. L'ensemble du personnel est reparti en trois équipes comme le montre le graphe correspondant :



Pour limiter l'affichage aux bureaux occupés par les équipes B et C, on supprime la branche reliant le noeud Personnel au noeud A.

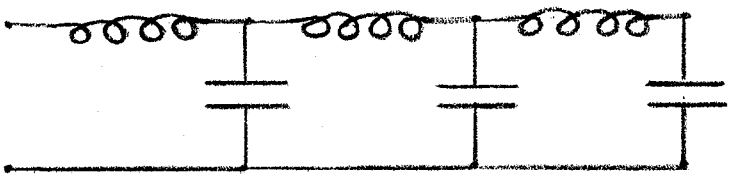


Le dessin correspondant au noeud Personnel devient alors :

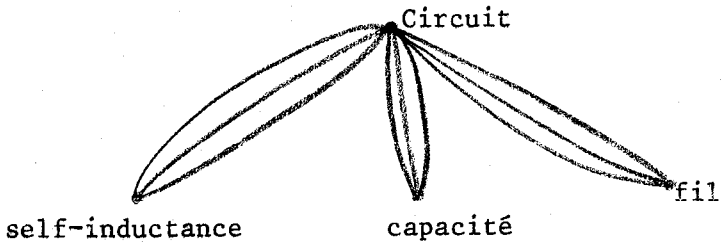


La position géographique des divers bureaux est une constante, tandis que les relations qui sont explicitées dans le graphe sont temporaires et susceptibles de subir de fréquentes modifications. D'autres relations correspondant à des interprétations différentes peuvent être établies et donner ainsi une signification tout à fait différente à un même dessin.

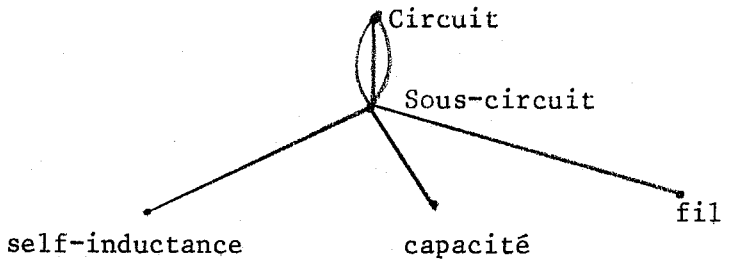
<u>Deuxième exemple</u> :Considérons cette fois un circuit électrique.



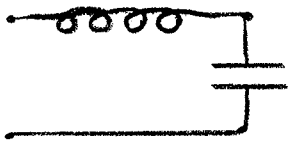
Des interprétations différentes seront représentées par des graphes différents :



Ici, le circuit est composé de trois self-inductances, trois capacités et trois fils. Ces éléments sont indépendants les uns des autres.

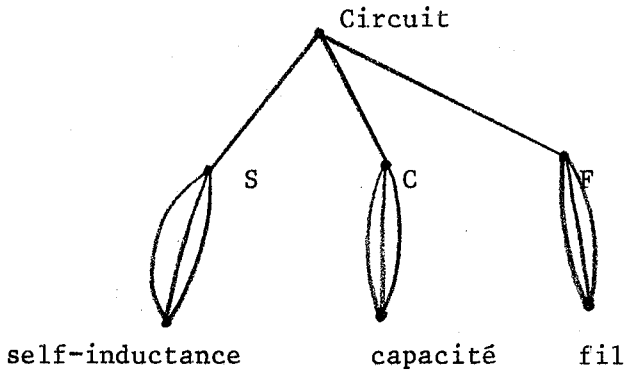


A partir d'un sous-circuit composé d'une self, d'une capacité et d'un fil :



On constitue le circuit principal en utilisant trois sous-circuits avec des paramètres de position judicieusement choisis.

Enfin, on peut définir des niveaux servant à regrouper des éléments de même type :



Les niveaux S, C et F ont peut d'intérêt du point de vue du dessin, mais on peut leur attribuer une signification sémantique importante. Ainsi, par exemple, la suppression de toutes les capacités apparaissant dans le circuit se réduit à la suppression de la branche liant le sommet "circuit" au sommet C.

Nous voici donc en possession d'un outil qui nous permet d'exprimer d'une part des relations hiérarchiques, d'autre part des regroupements entre divers objets et figures.

Ces informations, une fois codées en mémoire sous une certaine forme

pourront être exploitées par un algorithme spécialisé, dépendant de la nature du problème et des résultats désirés. Cet algorithme dispose alors en plus des informations géométriques (ou topologiques) de renseignements de niveau plus élevé fournis par les niveaux supérieurs du graphe. Notons que nous ne nous intéressons pas ici à ces algorithmes spécialisés, mais seulement à la fabrication des informations diverses contenues dans un dessin.

B. SYSTEME PERMETTANT LA CONSTRUCTION SIMULTANEE D'UNE IMAGE ET DE SA REPRESENTATION INTERNE.

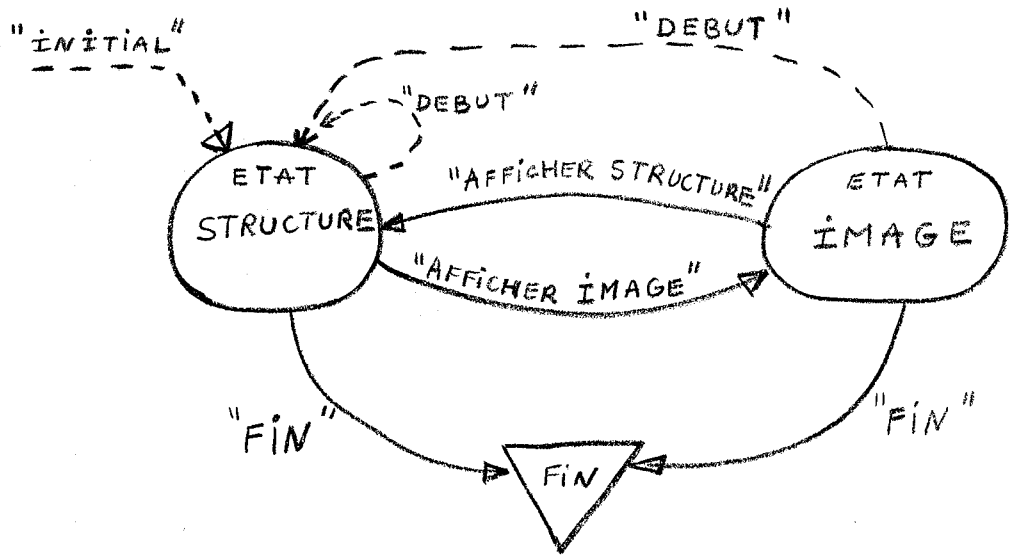
L'idée de ce système est de montrer à un utilisateur qui construit une figure sur l'écran d'un terminal graphique quelles sont les répercussions de ses actions en ce qui concerne la représentation interne de l'image. Pour ce faire, l'utilisateur peut à tout instant demander à "voir" la structure de son dessin, structure que nous avons choisi de présenter sous la forme d'un graphe pour les raisons développées précédemment.

A partir du moment où la structure interne devient accessible, il est logique de permettre d'agir directement sur celle-ci. Il devient alors possible de construire non plus seulement des objets mais également les relations qui existent entre ces objets.

B.1. Description externe.

Le programme développé pour expérimenter les idées précédentes se présente sous la forme d'un système intéractif utilisant la console IBM 2250. Les commandes que nous allons décrire seront lancées à l'aide du clavier de touches de fonction et du pointeur optique, associés à cette console.

Dans une première approche, on peut représenter ce programme sous la forme approximative d'un automate :



On distingue deux états principaux : l'état "structure" (ou état "graphe"), qui constitue également l'état initial, et l'état "image".

La touche "début" permet d'annuler tout le travail qui a été fait et réinitialise le programme.

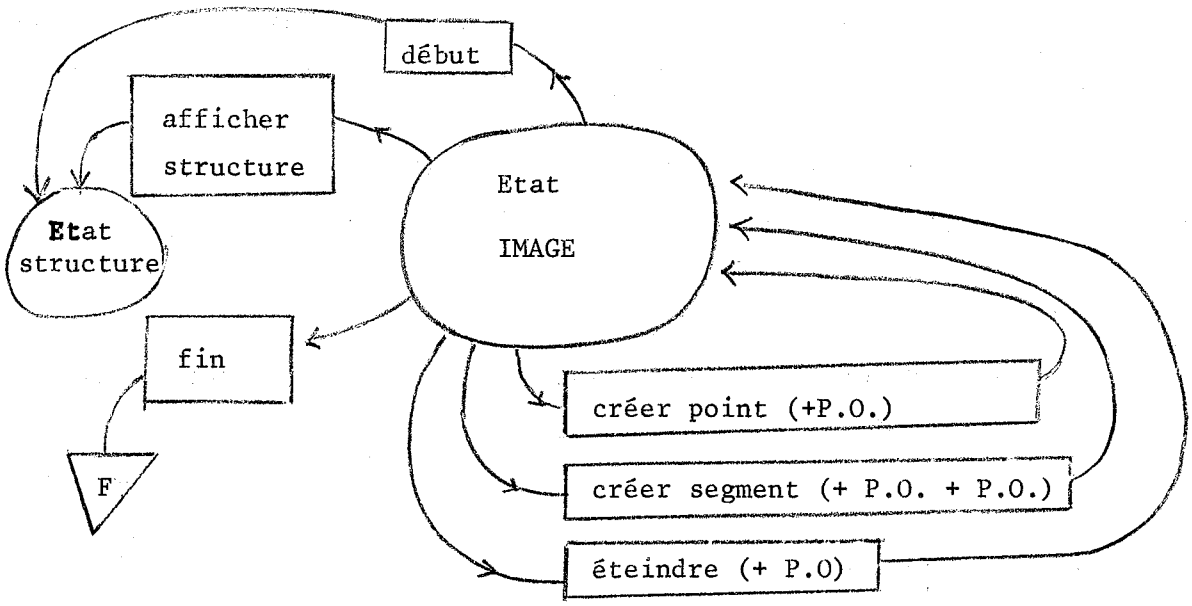
La commande "fin" entraîne la fin de l'exécution du programme.

Ces deux commandes peuvent être appliquées indifféremment dans les deux états.

a) Etat "image"

Cet état correspond à des solutions déjà éprouvées dans d'autres systèmes, aussi nous l'avons limité à un minimum.

Les commandes disponibles sont indiquées par le schéma suivant :



Créer Point

La touche "créer point" déclenche le rideau de caractères bien connu des utilisateurs de la console de visualisation IBM 2250 (voir note à la fin de ce paragraphe). Les coordonnées du caractère détecté par le pointeur optique (abrégé en P.O) donnent les coordonnées du point dans le système de

coordonnées du dessin. Le point apparaît immédiatement après la disparition du rideau de caractères.

Créer segment

Il s'agit ici, après avoir appuyé sur la touche correspondante, d'indiquer avec le pointeur optique deux points précédemment définis. Ces points sont pris comme extrêmités d'un segment qui est alors tracé.

Eteindre

Cette commande permet, en principe, de faire disparaître l'élément détecté avec le pointeur optique. Cependant, son emploi est très délicat, car cette commande n'agit pas au niveau de l'objet (point ou segment), mais au niveau de la figure. Un objet appartient le plus souvent à une figure, et c'est la figure entière qui va disparaître.

Une commande analogue mais agissant sur la représentation interne est beaucoup plus souple et plus commode à utiliser comme nous le verrons plus loin.

Remarquons que l'opération inverse est impossible puisque le pointeur optique ne peut détecter que des objets affichés.

Afficher structure

Cette dernière commande permet de passer dans l'état "structure" et entraîne la disparition du dessin.

NOTE : Le rideau de caractères.

Rappelons le principe de ce procédé. Le pointeur optique ne peut détecter que des objets qui apparaissent effectivement sur l'écran. Pour désigner un point de l'écran qui ne correspond pas à un objet, il est nécessaire d'afficher quelque chose en ce point, mais celui-ci est encore inconnu du système. Des techniques de poursuite du pointeur optique sont souvent employées dans ce cas [1-LUC] . Le procédé utilisé pour la console IBM 2250 consiste en l'affichage d'un nombre maximum

d'exemplaires d'un certain caractère, de façon à allumer un nombre maximum de points de l'écran. Il suffit alors de détecter l'un de ces caractères au pointeur optique. Les coordonnées d'affichage du coin inférieur gauche de ce caractère sont alors retenues comme les coordonnées du point désiré. Puis le rideau de caractères disparaît.

Notons que ce système ne donne que des coordonnées très approchées, et a l'inconvénient de faire disparaître l'image. Il ne convient donc pas à un travail précis.

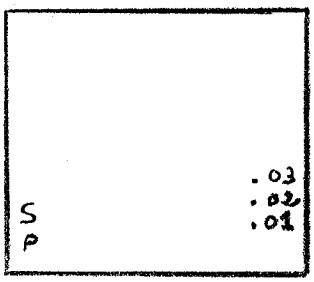
b) Etat "structure".

Dans cet état, on travaille sur le graphe représentant le dessin, ce graphe étant affiché sur l'écran à la place de l'image. Des commandes pour ajouter ou supprimer des éléments, pour indiquer des paramètres, et pour préciser des options sont alors disponibles.

Le graphe aura la forme définie précédemment: on y trouvera des noeuds représentant des niveaux de figures, des branches représentant des appels de figures et enfin des feuilles (ou extrémités).

Les feuilles seront de deux types: objets ou sous-programmes. Les objets en nombre volontairement limité sont les points et les segments. Afin de compenser ce nombre réduit, il est possible d'introduire, comme nouveaux objets, des sous-programmes préalablements écrits, donnant ainsi accès à des objets d'une quelconque complexité.

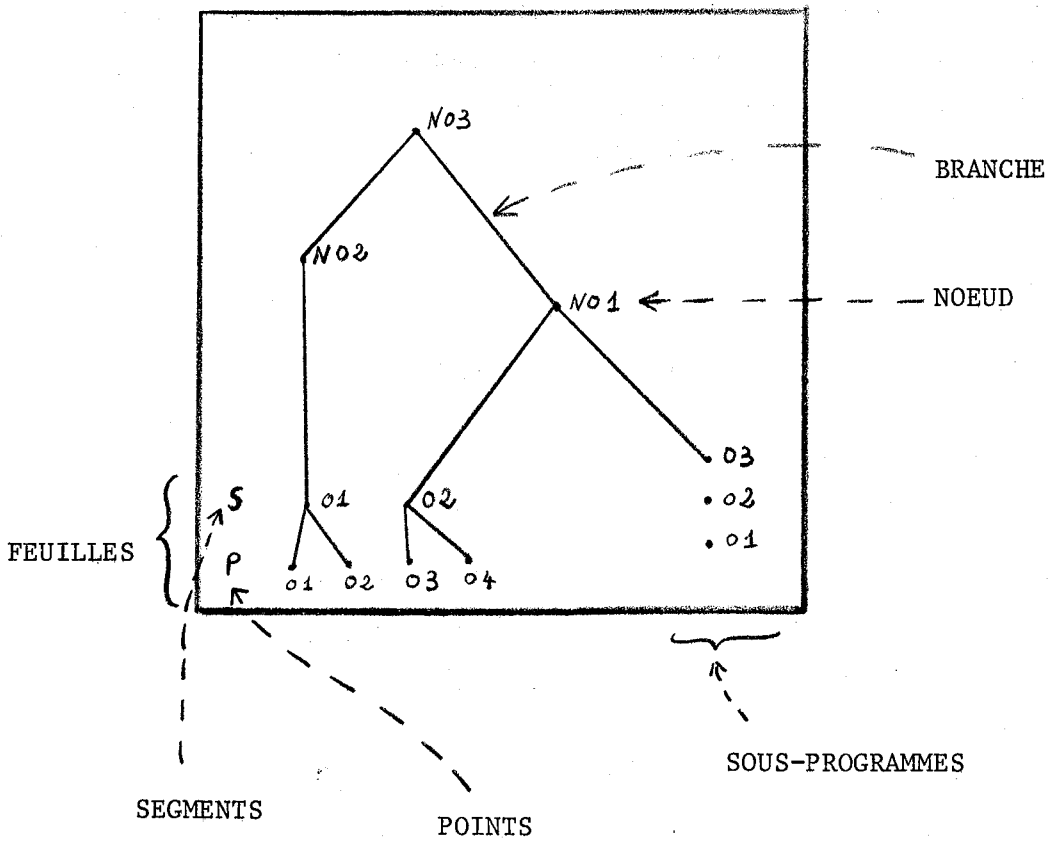
En début de travail, ou après utilisation de la touche "début", la structure est réduite à sa plus simple expression:



Dans la partie droite de l'écran apparaît une colonne de points, suivis d'un numéro. Chaque point représente un sous-programme pré-défini.
S'il n'existe aucun sous-programme, cette colonne n'apparaît évidemment pas.

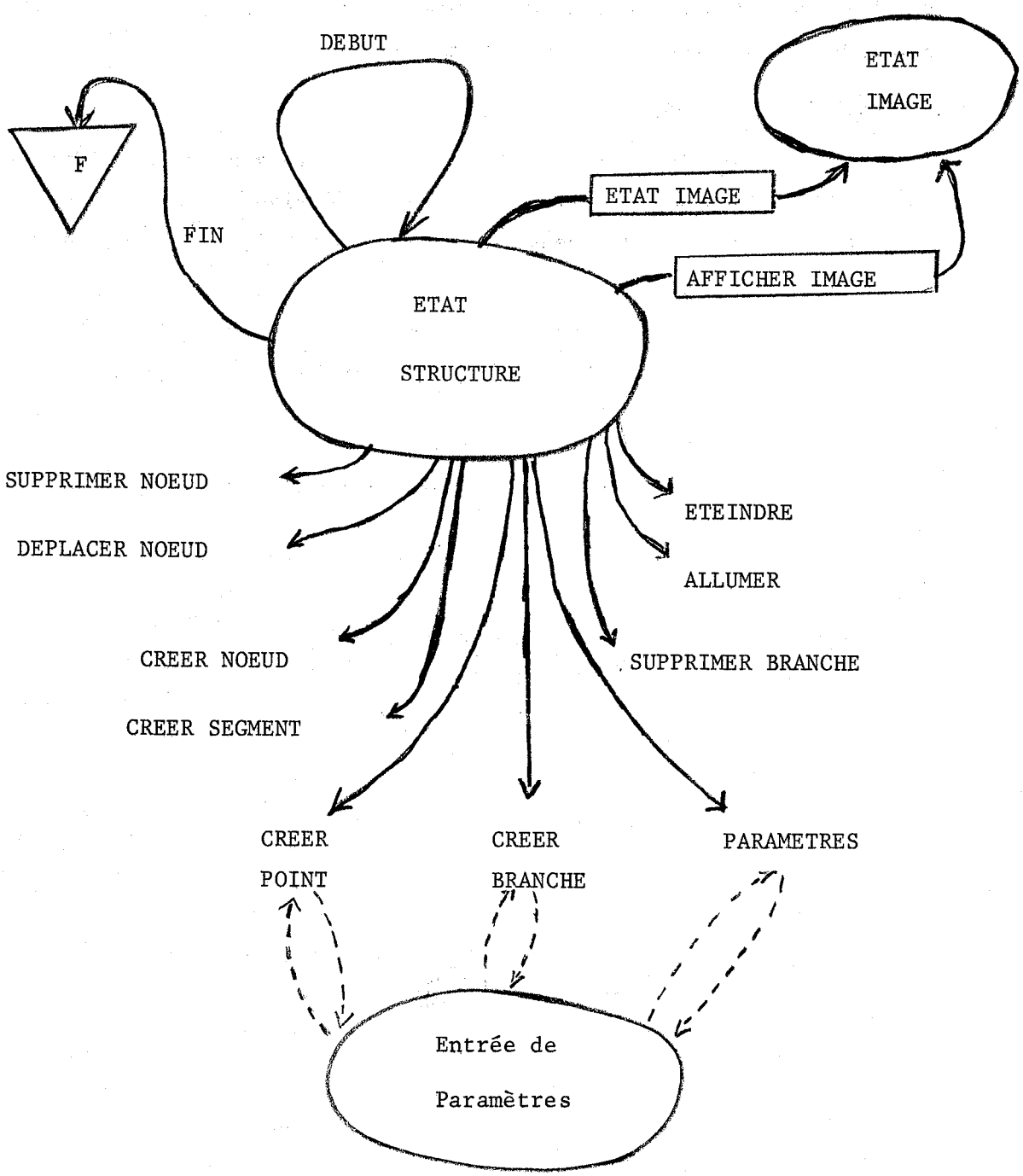
Dans un coin inférieur gauche, les lettres S(pour segments) et P (pour points) sont affichées. Nous verrons plus loin leur signification.

Après un certain temps de travail, on peut obtenir par exemple la configuration suivante :



On retrouve ici la forme de graphe décrite précédemment.

Les commandes disponibles dans cet état "structure" sont représentées par le schéma suivant :



Description des commandes

Les deux commandes de création d'objets "point" et "segment" ont

le même but que les deux commandes de même nom présentes dans l'état "image". Il s'agit encore de créer un point ou un segment.

Créer point :

Une nouvelle feuille est ajoutée dans le graphe. Cette feuille est automatiquement placée sur la ligne horizontale désignée par la lettre P, à la suite des autres feuilles qui peuvent s'y trouver. Un numéro est attribué par le système à cette feuille.

Ainsi, si nous avions :

P .01 .02 .03

l'effet de la touche "créer point" sera de produire la configuration suivante :

P .01 .02 .03 .04

Il reste encore à définir ce point, c'est-à-dire à indiquer ces coordonnées. Le système passe dans l'état "entrée de paramètres", que nous décrirons plus loin, et les valeurs des coordonnées sont indiquées par l'intermédiaire du clavier de touches de fonctions.

Créer segment :

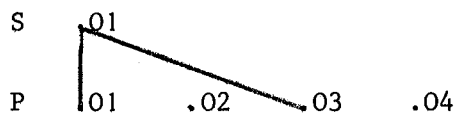
Comme dans l'état "image", un segment est défini à partir de deux points préalablement existants. Une feuille est ajoutée à la ligne S pour représenter ce nouveau segment. On désigne alors, avec le pointeur optique, deux feuilles de la ligne P qui représenteront les extrémités du segment.

Ainsi, en partant de

S

P .01 .02 .03 .04

après création du segment reliant les points 01 et 03, nous aurons :



Notons que cette définition des segments et des points est arbitraire et que l'on peut concevoir d'autres systèmes de définition. Par exemple,

un point pourrait être défini comme intersection de deux segments etc....

Créer noeud :

Cette commande permet d'indiquer avec le pointeur optique la position dans le graphe d'un nouveau noeud, (on fait encore usage du rideau de caractères); ce noeud est automatiquement numéroté. Le niveau de figure représenté par ce noeud reste vide tant que l'on n'a pas indiqué les éléments composant cette figure.

Créer branche :

Pour ajouter un élément (point, segment, sous-programme ou autre figure) à une figure, il suffit de tracer une branche reliant la figure à l'élément désiré. On montre pour cela avec le pointeur optique les deux sommets dans l'ordre,figure puis élément. La branche apparaît alors, puis le système passe dans l'état "entrée de paramètres". Ces paramètres seront décrits plus loin.

Changer paramètres :

permet de changer les valeurs des paramètres associés à la branche que l'on désigne au pointeur optique. On passe alors dans l'état "entrée de paramètres".

Supprimer branche :

permet de détruire la branche désignée. Ceci permet de modifier la composition d'une figure en supprimant certains de ses éléments.

Eteindre-Allumer :

Ces deux commandes n'ont pas d'effets apparents au niveau du graphe. Elles précisent si, dans l'état "image", l'élément désigné au pointeur optique doit ou non être affiché.

La signification est évidente pour les sommets : selon l'état ("allumé" ou "éteint") de ce sommet-là, la figure qui lui correspond sera ou ne sera pas affichée.

Cette option peut également être indiquée pour une branche. Elle s'applique alors au noeud qui est extrémité inférieure de la branche, mais uniquement pour l'utilisation de ce noeud correspondant à cette branche. L'option reste propre à la branche et ne devient pas une propriété du noeud. En revenant à l'aspect "appel de sous-programme" d'une branche, l'option "éteint" revient à supprimer temporairement l'appel du sous-programme. La commande "supprimer branche" correspond, elle, à une suppression définitive.

Notons que cette commande est beaucoup plus souple que son équivalent dans l'état "image".

Déplacer noeud

Cette commande permet de modifier l'aspect du graphe par déplacement d'un noeud (désigné au pointeur optique) vers une autre position de l'écran (position repérée par l'intermédiaire du rideau de caractères).

Supprimer noeud.

Un noeud isolé, c'est-à-dire qui n'est ni origine, ni extrémité d'une branche peut être désigné au pointeur optique après action sur cette touche. Le noeud disparaît alors du graphe.

Afficher image.

Cette commande permet d'obtenir le tracé de la figure qui correspond au sommet désigné au pointeur optique. Après l'affichage, le système passe dans l'état "image".

Etat image.

A la différence de la commande précédente, celle-ci permet de passer dans l'état "image", mais sans affichage de figure.

Paramètres associés à une branche.

Nous avons signalé d'une part, l'analogie entre une branche et un

appel de sous-programme, et d'autre part l'existence possible de paramètres (translation, homothétie et rotation) conditionnant l'affichage d'une figure.

Dans le cadre de cette étude, nous nous sommes intéressés aux deux premiers types de paramètres. Nous avons donc associé trois paramètres à une branche: deux pour la position et un pour l'échelle de la figure.

Etant donné un dessin, on peut vouloir faire apparaître sur l'écran la totalité de ce dessin. Dans ce cas, on a une vue d'ensemble dans laquelle les détails ne sont pas très nets. Afin d'améliorer la perception de ces détails, on peut chercher à n'afficher qu'une portion du dessin. Cette portion dispose alors de la totalité de l'écran et semble agrandie. Nous dirons que nous avons augmenté l'échelle. Ce processus de sélection et d'agrandissement peut encore se répéter. Le processus inverse (diminution de l'échelle) permet de revenir à une vue globale du dessin, avec atténuation des détails perceptibles.

Nous devons donc disposer de deux possibilités de réglage: d'une part sur la valeur de l'échelle, d'autre part sur la position de la portion de dessin à afficher.

Conventions utilisées.

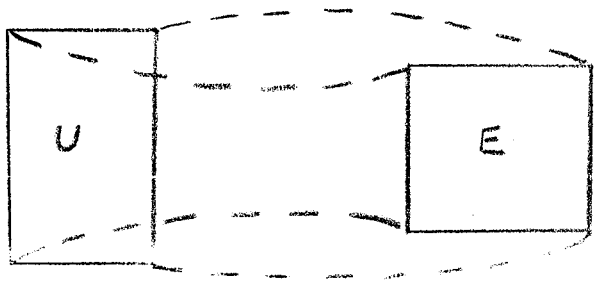
Nous pouvons considérer qu'un dessin est défini dans l'espace $\mathbb{Z} \times \mathbb{Z}$. (Nous ne considérons que des coordonnées entières, cas auquel on peut se ramener par un changement d'unités ou par des arrondis).

On veut afficher une portion U de cet espace, portion que l'on estime intéressante à un moment donné.

Posons $U = \left[X_i , X_s\right] \times \left[Y_i , Y_s\right]$

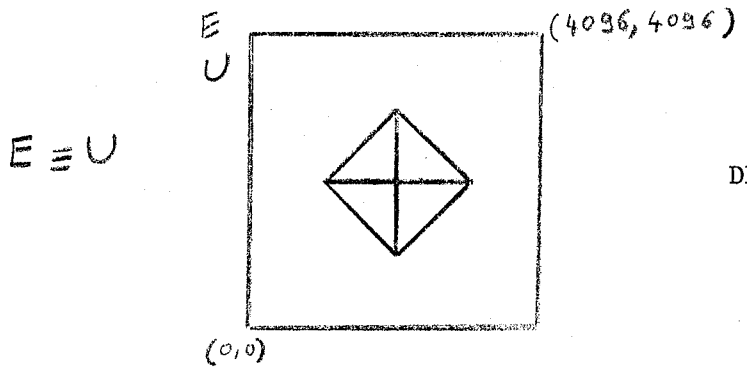
Soit $E = \left[0,4096\right] \times \left[0,4096\right]$, l'espace constitué par le système de coordonnées de l'écran.

Le processus d'affichage consiste en une certaine application de U sur E.



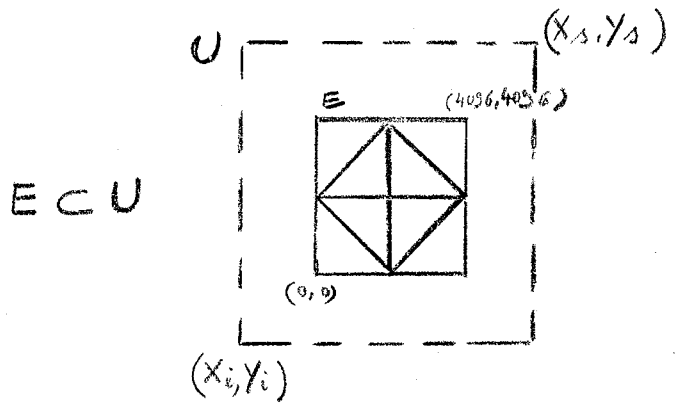
Lorsque U est identique à E, la transformation est évidente: à un point (x,y) de U correspond le point (x,y) de E. Nous dirons alors que le dessin est à l'échelle 1.

Ainsi nous supposerons par convention qu'un dessin préalablement défini (tel qu'un sous-programme) appartient entièrement à l'espace E. De plus, nous supposerons que le centre de ce dessin est le point $\left(\frac{4096}{2}, \frac{4096}{2}\right)$.



DESSIN INITIAL

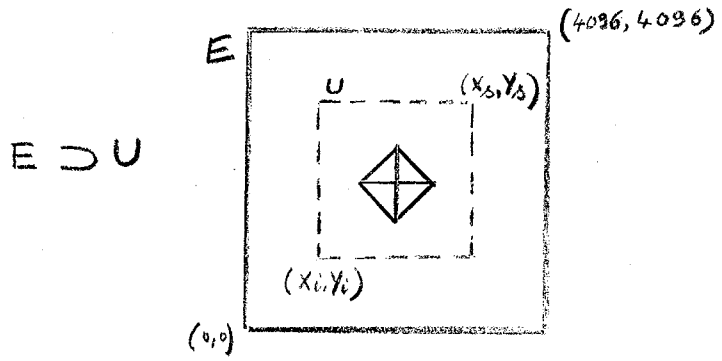
(ECHELLE 1)



ECHELLE > 1

(ici, échelle 2)

Tout l'écran est associé à une portion de U.

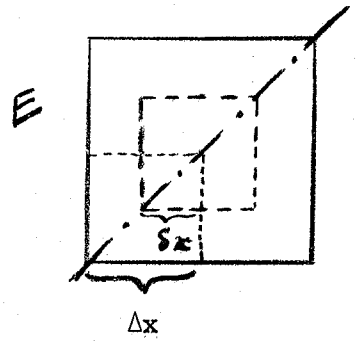


Echelle < 1

Ici, E est considéré comme étant un espace englobant U. U ne dispose donc que d'une portion de l'écran et par suite le dessin apparaît plus petit

Le langage GSP que nous utilisons contient une fonction qui assure la correspondance entre un point de E et un point de U, à partir des valeurs des bornes de l'espace U. Les formules de transformation sont simples et il ne paraît pas utile de les faire figurer ici.

Notre problème est en fait légèrement différent :en supposant le dessin original défini sur E et centré sur le centre de E, il s'agit de déterminer les valeurs de X_i, X_s, Y_i et Y_s correspondant à un facteur d'échelle donné.



Δx : Constante

Posons $X_o = \frac{4095}{2}$, $Y_o = \frac{4095}{2}$ (CENTRE de l'ECRAN)

On a

$$\begin{cases} X_i = X_o - \delta x \\ X_s = X_o + \delta x \end{cases}$$

avec $\Delta x = e \ \delta x$ e : facteur d'échelle

Enfin pour une échelle donnée, nous pouvons examiner différentes parties du dessin initial. Il suffit d'effectuer une translation, définie par deux valeurs T_x et T_y.

Les formules complètes sont alors :

$$X_i = X_o - \delta x + Tx$$

$$X_s = X_o + \delta x + Tx$$

$$Y_i = Y_o - \delta y + Ty$$

$$Y_s = Y_o + \delta y + Ty$$

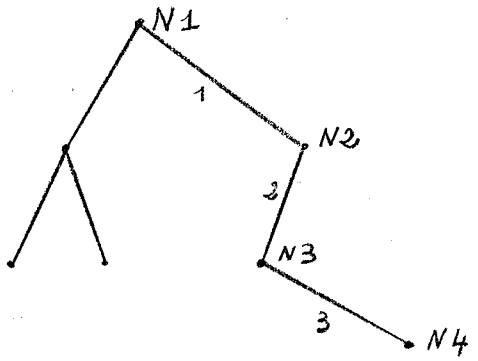
Les trois paramètres associés à une branche déterminent bien toutes les valeurs qui nous sont nécessaires.

REMARQUE : Afin d'éviter l'utilisation de nombres non entiers, nous avons fait les conventions suivantes: le facteur d'échelle sera entier, positif ou négatif.

Si e est négatif, il sera interprété comme $\frac{1}{|e|}$.

Cumul des paramètres

Considérons la portion de structure suivante :



La commande "afficher l'image correspondant au noeud N1" aura en particulier l'effet d'afficher ce qui correspond au noeud N4. Il faut au préalable tenir compte des paramètres associés aux branches 1,2, et 3.

Par convention, nous ferons la somme algébrique des paramètres et le résultat fournira les paramètres utilisés pour l'affichage.

Ainsi dans ce cas, nous aurons :

$$e = \sum_{i=1}^{3} e_i$$

$$T_x = \sum_{i=1}^{3} T_{xi}$$

$$T_y = \sum_{i=1}^{3} T_{yi}$$

Etat "entrée des paramètres".

On peut pénétrer dans cet état soit pour changer (ou initialiser les valeurs des paramètres associés à une branche, soit pour donner les coordonnées d'un point que l'on définit dans l'état "structure",soit pour changer les coordonnées d'un point déjà défini.

Les valeurs effectives des paramètres de la branche (X pour "translation en X", Y pour "translation en Y", E pour "facteur d'échelle") ou des coordonnées du point (X pour "abscisse" et Y pour "ordonnée") sont alors affichées sur l'écran.

On désigne ensuite la composante que l'on désire modifier, en appuyant sur la touche correspondante du clavier. Une nouvelle valeur décimale entière, positive ou négative, est indiquée par actions sur les touches; cette valeur apparaît sur l'écran à la place de l'ancienne.

Lorsque les modifications sont terminées, une touche permet de sortir de cet état pour continuer le travail, les nouvelles valeurs étant alors rangées en mémoire dans le bloc correspondant à la branche ou au point selon le cas.

B.2. Description interne.

Compte tenu de l'orientation de notre programme vers un travail sur une structure en graphe, c'est cette structure que nous avons cherché à représenter en mémoire.

Dans une première approche réalisée en langage FORTRAN et employant le langage graphique GSP, nous avons utilisé des tableaux comme représentation interne du graphe. Ceci nécessitait des réservations inutiles de place en mémoire (il fallait demander toute la place dès le début de l'exécution), et conduisait à des algorithmes relativement lourds et peu efficaces.

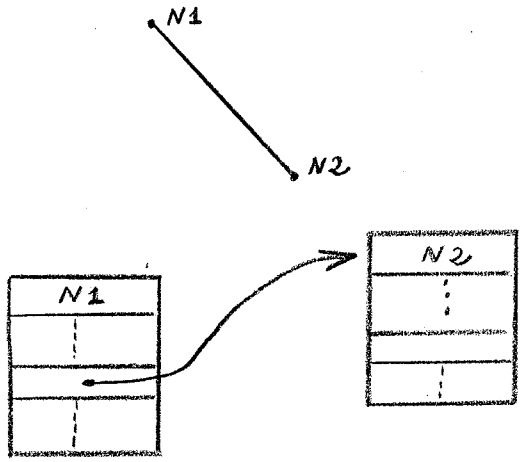
Aussi nous avons réécrit ce programme, en utilisant toujours GSP, mais cette fois en langage d'assemblage 360. Nous disposons alors de possibilités dynamiques d'allocation de mémoire et de toute la souplesse voulue.

a) Représentation en plex.

Chaque élément du graphe (noeud, point, segment, sous-programme) sera représenté par un bloc en mémoire, obtenu dynamiquement au moment de la création de cet élément. La dimension du bloc varie avec le type de l'élément représenté.

Il faut encore exprimer les relations entre ces divers éléments. Dans la représentation externe, ces relations sont les branches.

Pour exprimer le fait qu'il existe une branche allant de N1 à N2, on peut envisager de mettre dans le bloc représentant N1 un pointeur vers le bloc représentant N2.

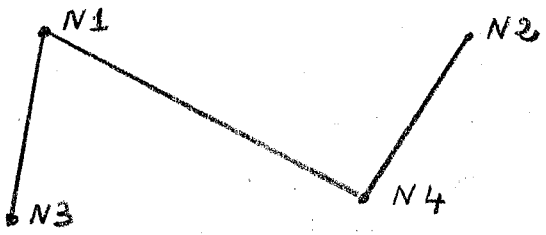


Une deuxième branche issue de N1 nécessite un deuxième pointeur, etc...
On est alors conduit à prévoir à l'avance (i.e. au moment de la création du bloc) la place nécessaire pour les pointeurs futurs, ce qui entraîne un gaspillage si on réserve trop de place et une limitation si on en réserve trop peu.

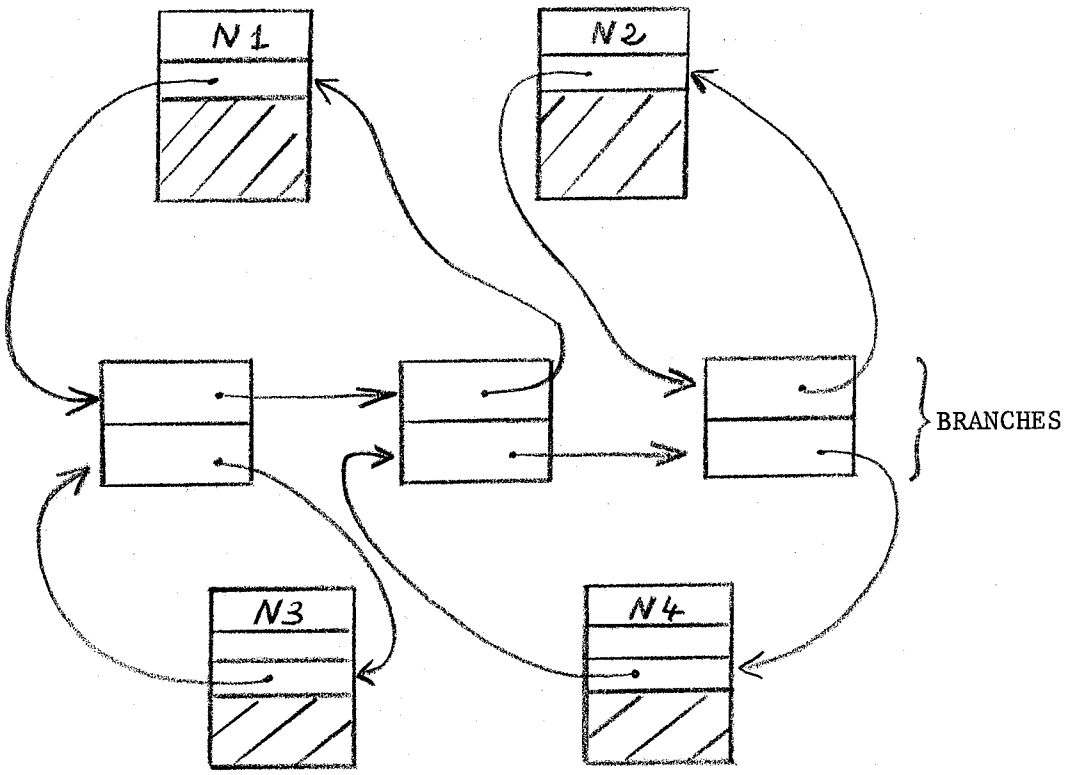
Aussi, nous avons adopté la solution suivante : une branche sera également représentée par un bloc.

Toutes les branches issues d'un même noeud sont placées sur un anneau dont la tête est ce bloc noeud.

Ainsi, la situation suivante :



sera représentée par :



Les dimensions de tous les blocs peuvent ainsi être fixées dès le départ, et sont invariables.

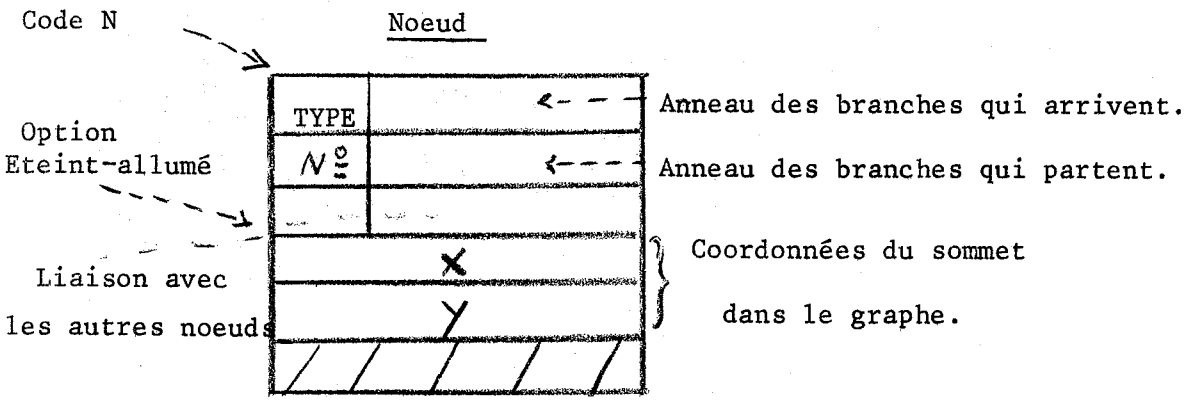
Les valeurs des paramètres associés à chaque branche seront stockées dans le bloc branche correspondant.

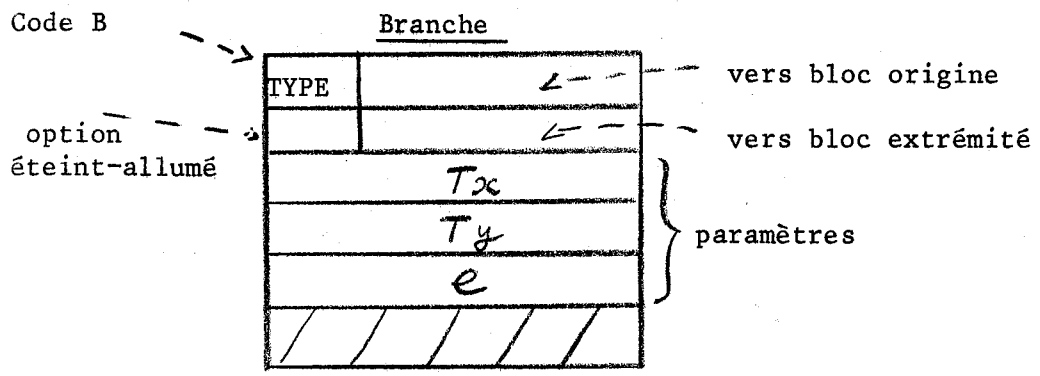
Certaines liaisons entre blocs n'ont pas d'équivalents directs dans le graphe et ont pour but d'améliorer l'efficacité de certains algorithmes d'exploration du graphe. Ainsi, tous les blocs d'un même type sont placés sur un même anneau (anneau des noeuds, des points, des segments et des sous-programmes) à la seule exception des blocs branches.

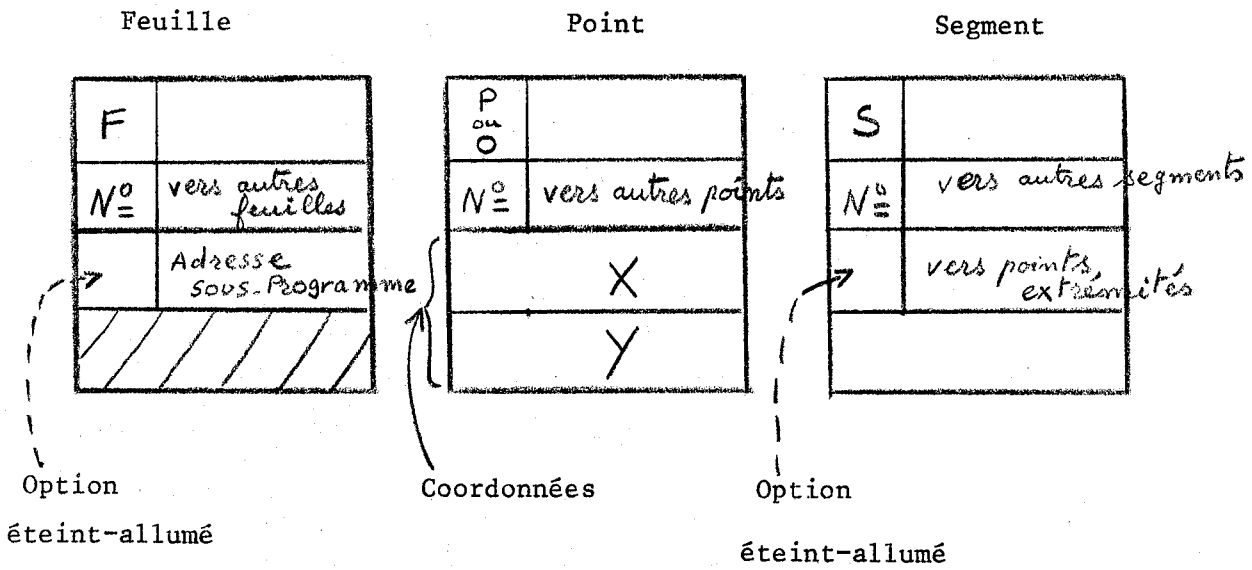
b) Forme interne des blocs représentatifs.

Les modèles de chaque type de bloc sont décrits complètement dans les

schémas qui suivent :







c) Structure du programme.

Afin de faciliter d'une part la recherche des erreurs, d'autre part l'adjonction de nouvelles commandes, le programme est conçu sous forme modulaire. Un élément important étant la communication entre le système et l'utilisateur, il nous faut tout d'abord préciser comment se fait la gestion des interruptions dans le langage GSP.

Parmi les sources d'interruptions possibles (les 32 touches de fonction et le pointeur optique), nous pouvons choisir celles qui nous intéressent à un instant donné en utilisant l'instruction ENATN(enable attention), suiviedes codes désignant les interruptions à autoriser. Tant qu'un autre ENATN n'est pas rencontré, toutes les autres sources d'interruptions sont ignorées.

On se place ensuite en état "attente d'interruptions" par l'instruction RQATN(request attention) suivie des codes des interruptions attendues. C'est à ce moment, et à ce moment seulement, que l'on peut soit traiter une interruption précédemment envoyée(et empilée automatiquement par GSP), soit attendre que se produise une interruption.

Ce système de gestion des interruptions nous a conduit à l'organisation suivante.

Un programme principal sert à initialiser le système GSP, et joue ensuite le rôle de centre de contrôle des interruptions. Nous avons ainsi dans ce programme deux instructions RQATN, l'une correspondant à l'état "image", l'autre à l'état "structure".

Après réception d'une interruption, nous appelons le programme spécial chargé de la traiter. Tous ces programmes sont indépendants les uns des autres et ils gèrent eux-mêmes leurs propres interruptions.

Il faut noter que toutes ces interruptions se font à un seul niveau. Seul le programme en cours peut prendre en compte une interruption, et seulement lorsqu'il le désire. Il n'est pas possible, par exemple, d'envoyer une interruption non prévue ayant pour effet de stopper le programme en cours (cas d'une touche Panique).

L'organisation générale du programme correspond exactement à l'organisation des commandes. Chaque commande est associée à un programme de traitement correspondant.

B.3 Développements ultérieurs envisageables.

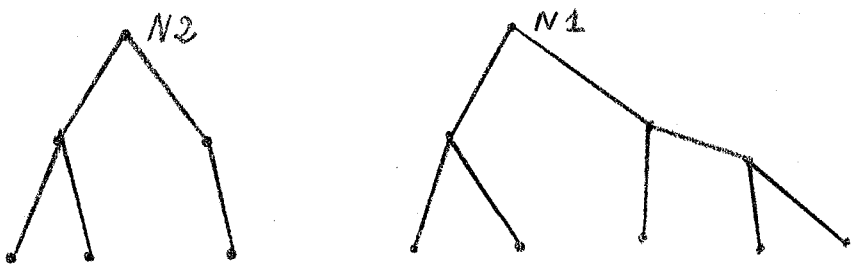
Il est aisé d'envisager de compléter le programme par l'adjonction de nouvelles commandes, celles-ci étant pour l'instant relativement peu nombreuses.

a) Les structures que l'on veut construire deviennent rapidement assez compliquées et ceci conduit à un encombrement de l'écran, qui ne facilite pas la compréhension du graphe.

Il parait donc souhaitable de pouvoir éliminer la partie du graphe correspondant à une figure que l'on estime complètement définie. Une telle figure pourrait alors être classée parmi les sous-programmes. Tous les éléments du graphe propres à cette figure peuvent alors être supprimés.

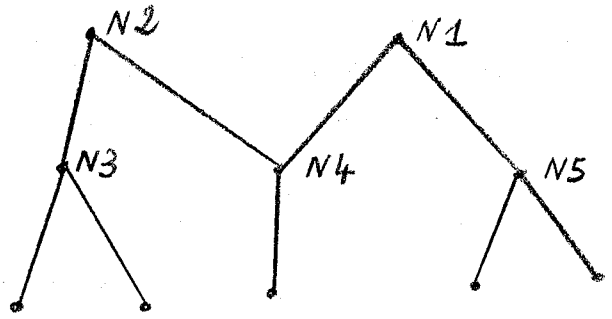
La difficulté est justement de déterminer ces éléments. En fait, tous les sommets représentent des éléments qui peuvent toujours être utilisés pour construire d'autres figures. Les faire disparaître revient à supprimer la possibilité de les utiliser, ce qui est évident, mais d'autre part supprime la possibilité de construire des figures ayant des liens entre elles.

Soient, par exemple, les figures N1 et N2.



Ici N1 et N2 sont indépendantes. On peut envisager de supprimer le graphe de sommet N1.

Considérons maintenant la situation suivante :



Ici, le niveau N4 est commun à N1 et N2. On ne peut donc le supprimer.

La suppression du niveau N5 et de ses descendants oterait toute possibilité pour une autre figure d'utiliser la figure qu'il représente.

Aussi, la commande devrait-elle se réduire à supprimer le sommet et les branches directement issues de ce sommet, puis à faire réapparaître le sommet dans la colonne des sous-programmes. La simplification perd de ce fait beaucoup de son intérêt.

b) Opérations sur les graphes.

La notion de graphe a fait l'objet de diverses études mathématiques Leurs résultats pourraient être utilisés pour définir des opérations de hau niveau tels que somme ou produit de figures, par exemple. Les avantages présentés par de telles opérations ne sont toutefois par évidents.

c) Contraintes.

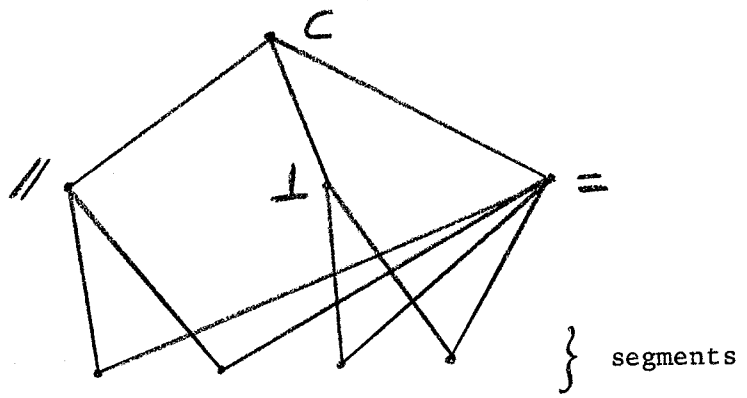
Par contrainte, on désigne généralement une certaine condition à laquelle doivent satisfaire un ou plusieurs éléments d'une figure, et ce quelles que soient les autres conditions imposées à cette figure.

Prenons pour exemple les contraintes géométriques. Ce sont des conditions imposées à la forme du dessin. Plusieurs conditions peuvent être impossibles à satisfaire simultanément, et il s'agit alors de trouver une solution optimale suivant certains critères.

Il est possible de représenter ces contraintes également sous la forme d'un graphe. Un sommet représentera un niveau de contrainte, les

extrémités seront les éléments soumis à ces contraintes, les branches serviront à indiquer les opérandes et la hiérarchie des divers niveaux.

Exemple :



La contrainte C est une contrainte de haut niveau construite à partir de plusieurs contraintes élémentaires. Elle représente une condition déjà complexe qui a quatre segments comme opérandes.

De tels graphes de contraintes représentent des informations supplémentaires qu'il serait agréable de pouvoir faire apparaître ou disparaître à volonté au cours d'un travail.

Insistons une dernière fois sur les points importants de ce système. La possibilité d'accès à la représentation interne est une facilité nouvelle qui présente deux avantages. Tout d'abord, il devient possible de prendre conscience concrètement de l'existence d'un aspect différent de l'image, ceci pouvant conduire à des prises de décision plus rationnelles.

D'autre part, certaines ambiguités existant au niveau de l'image sont levées dans la représentation interne. Certaines relations entre les éléments de la figure deviennent aisément accessibles et sont susceptibles d'être modifiées et complétées.

La définition de ce que nous avons appelé niveau de regroupement (ou niveau de signification)est une opération simple, dans le graphe, alors qu'elle serait beaucoup plus difficile à exprimer autrement.

Une application simple de cette possibilité est l'élimination des détails inutiles. Pour éliminer, à un instant donné, des éléments, jugés superflus, de l'image, il suffit de construire un niveau qui regroupe tous ces éléments, puis d'appliquer l'option "éteint" à ce niveau. Lorsque l'on repasse dans l'état "image", ces éléments ont alors disparu et ils sont ignorés par le système, qui évite ainsi de perdre du temps à les tracer. C'est seulement pour voir l'image complète que l'on replacera ce niveau dans l'état "allumé".

La présence de paramètres associés à chaque utilisation d'une figure donne également la possibilité de composer une image à partir d'une collection de sous-programmes prédéfinis. Chaque figure peut être placée à l'endroit désiré, avec la dimension désirée. Une même figure peut être réutilisée un nombre quelconque de fois, chaque exemplaire étant affecté de paramètres différents.

Enfin, il faut remarquer que l'on savait jusqu'ici construire des objets, tels que points, segments... en dessinant directement sur l'écran. On peut voir qu'il est également possible de construire de la même façon les relations (abstraites) qui existent entre ces objets, alors que ceci était jusqu'à présent dans le domaine d'une programmation classique.

Il nous semble que ce développement d'une programmation réellement graphique est un point intéressant, qui serait susceptible d'être quelque peu développé.

CONCLUSION

Le concept de structure de représentation des données que nous avons étudié sous divers angles est une notion vaste et complexe. Selon les objectifs et la mise en oeuvre d'un système donné, les structures utilisées seront différentes, même si certaines idées de base se retrouvent fréquemment.

La réalisation de chacun de ces systèmes reste assez empirique et il n'est pas possible de définir une structure "universelle" utilisable dans toutes les situations.

Partant de cette constatation, de nombreux outils ont été créés afin de faciliter le développement d'une structure adaptée aux besoins propres de chacun. Il faut cependant remarquer que d'une part ces outils ont pris la forme de langages de programmation, et que d'autre part le choix entre ces langages se fait bien souvent non pas en fonction de leurs diverses possibilités, mais selon leurs disponibilités dans une installation particulière.

Parmi les directions de recherche les plus intéressantes nous citerons les recherches les plus formelles qui essaient de regrouper par une théorie mathématique cohérente les divers modèles de structures utilisés. Il est en effet frappant de constater que dans le domaine de l'image et de sa fabrication, on est loin du niveau de formalisation mathématique atteint dans le domaine voisin de la reconnaissance de formes. Il pourrait être intéressant de faire le lien entre ces deux aspects.

Les travaux de A.SHAW [3-SHA] sur l'analyse des figures représentables par des graphes montrent aussi une direction de recherches intéressante.

Le mécanisme d'adressage utilisé dans les calculateurs est également mis en question car il ne facilite pas la manipulation de structures complexes.

Lorsque les structures sont non seulement complexes mais, en plus, volumineuses, il devient nécessaire de travailler sur plusieurs niveaux de mémoire. Il reste encore à trouver des méthodes qui éviteraient l'éparpillement de groupes de données qui forment un tout logique, et qui se trouvent dispersés dans un système de pagination classique.

Remarquons enfin que ce concept de structures d'information se retrouve également dans les études sur l'intelligence artificielle. Il s'agit ici encore de représenter en mémoire des informations externes sans perdre les éléments indispensables et en conservant la cohérence de ces informations, afin de permettre leur traitement.

Il y a encore beaucoup de travail à faire si l'on pense qu'à la limite nous avons la structure du cerveau humain.

Nous espérons toutefois que la complexité de ce domaine ne découragera pas des études ultérieures.

BIBLIOGRAPHIE

1. OUVRAGES GENERAUX SUR LES CONSOLES DE VISUALISATION ET LEURS APPLICATIONS.

1-ADI M.ADIBA
"Reconnaissance de formes".Rapport de D.E.A. Université de Grenoble 1968.

1-AHC D.V.AHUJA et S.A.COONS
"Geometry for construction and display" IBM system journal Vol 7 n°3 et 4 1968 pp.188-205.

1-DAM A.VAN DAM et S.MATSA
"Computer graphics I.Reference material". ACM professional development seminar. 1968

1-JOH C.I.JOHNSON
"Principle of interactive system"
IBM system journal Vol 7 N° 3 et 4. 1968 pp.147-173

1-LEW Morton H.LEWIN
"An introduction to computer graphics terminal".
Proc IEEE Vol 55 n°9 Sept 1967.

1-LUC M.LUCAS
"Techniques de programmation et d'utilisation en mode conversationnel des terminaux graphiques".
Thèse 3ème cycle Université de Grenoble Juin 1968.

1-POO Harry H.POOLE
"Fundamentals of display systems"
Mac Millan London 1966

1-PPG R.D.PARSLOW, R.W.PROWSE et R.E.GREEN
"Computer graphics : techniques and applications."
Plenum Press London 1969.

1-PRI David PRINCE
"Man-computer graphics for computer-aided design."
Proc IEEE Special issue on computers Dec.1966.

1-ROR D.T.ROSS et J.E.RODRIGUEZ
"Theoretical foundations for the computer-aided design system".
Proc SJCC 1963.

1-SID R.A. SIDERS and OTHERS
"Computer graphics: a revolution in design."
American Management Association New York 1966.

1-SUT2 I.E.SUTHERLAND
"Computer graphics: ten unsolved problems."
Datamation Mai 1966.

1-SUT3 I.E.SUTHERLAND
"Three kinds of graphic data processing."
Proc IFIP Congress 1965 pp.582-583

2. SYSTEMES ET LANGAGES GRAPHIQUES

2-CAG1 J.M. CAGNAT
"Etude d'un système de communication homme-machine : SKETCHPAD d'I.E.SUTHERLAND".
Rapport de D.E.A Université de Grenoble 1968

2-CHP C.CHRISTENSEN et E.N.PINSON
"Multi-fúnctions graphics for a large computer system".
AFIPS conf.Proc. 1967 FJCC

2-CLE E.CLEEMANN
"Un macro-langage pour la programmation des terminaux graphiques".
Thèse 3ème cycle. Université de Grenoble 1968.

2-CLL E.CLEEMANN,O.LECARME et M.LUCAS
"Langages de programmation graphique"
R.I.R.O. N°12 1968 pp.71-88

2-IBM1 IBM system/360 "Graphic programming services for IBM 2250 display unit." Form C27-6909.

2-IBM2 IBM system/360 "graphic programming services for FORTRAN IV."
Form C27-6032.

2-IBM3 IBM System/360 Component description.
"IBM 2250 Display Unit Model I". Form A27-2701.

2-JOH T.E.JOHNSON
"SKETCHPAD III: a computer program for drawing in three dimensions"
Proc SJCC 1963.

2-KNO1 K.C.KNOWLTON
"A computer technique for producing animated movies".
AFIPS conf.proc.Vol 25.1964 pp 67-87

2-KUL H.E.KULSRUD
"A general purpose graphic language"
Comm.ACM Vol 11.N°4 Avril 1968 pp.247-254

2-LEC O.LECARME
"Contribution à l'étude des problèmes d'utilisation des terminaux graphiques: un système de programmation graphique conversationnelle".
Thèse de Doctorat d'Etat Université de Grenoble Sept 1970.

2-LEL O.LECARME et M.LUCAS
"LAGROL:un langage pour l'utilisation d'un terminal graphique".
Note technique IMAG Janv.1968.

2-MEZ L.MEZEI
"SPARTA: a procedure oriented programming language for the manipulation of abitrary line drawings"
IFIP Congress Edinburgh Aout 1968

2-NOY J.NOLAN et L.YARBROUGH
"An on-line computer drawing and animation system."
IFIP Congress Edinburgh 1968

2-SIR Y.SIRET
"Contribution au calcul algébrique formel sur ordinateur"
Thèse de doctorat d'état Université de Grenoble Juillet 1970.

2-SUT1 I.E.SUTHERLAND
"SKETCHPAD: a man-machine graphical communication system."
Lincoln Lab.tech.report N°296 ou Proc.SJCC 1963.

Voir aussi 1-LUC

3. LES STRUCTURES D'INFORMATION.

3-BAM H.B.BASKINS et S.P.MORSE
"A multilevel modelling structure for interactive graphic design"
IBM system journal Vol.7 N°3 et 4.1968 pp.218-228

3-CAG2 J.M.CAGNAT
"Structures pour la représentation des données en programmation graphique".
Séminaire de programmation I.M.A.G 1969.

3-CAG3 J.M.CAGNAT
"Construction simultanée d'une structure en forme de graphe et de l'image correspondante".
Colloque sur les traitements graphiques. Grenoble 1970.

3-CHI D.L.CHILDS
"Feasibility of a set-theoretic data structure: a general structure based on a reconstituted definition of relation".
IFIP Congress Edinburgh 1968.

3-DUB J.J.DUBY
Cours sur les structures d'information.
Université de Grenoble 1968-1969

3-HAN W.J.HANSEN
"Compact list representation"
Comm.ACM Vol.12 N°9 Sept 1969 pp.499-507

3-HAR Y.HARRAND
"Traitement des files et des listes".
Bibl.de l'automaticien. Dunod (Paris) 1967 120 pp

3-HOA C.A.R.HOARE
"Data structures in two-level store".
IFIP Congress Edinburgh 1968

3-IMP M.E.D'IMPERIO
"Data structures and their representation in storage".
Annual Review in Automatic Programming
Vol.5 1968 Ed.Halpern and Shaw.Pergamon Press

3-JOD Jane C.JODEIT
"Storage organisation in programming system".
Comm.ACM Vol.11 N°11 Nov 1968 pp.741-746

3-KNU D.E.KNUTH
"The art of computer programming"
Vol.1 Chap.2 "Information structures".
Addison Wesley Reading Mass 1968

3-NAU P.NAUR
"DATALOGY: the science of data and data processes and its place in education".
IFIP Congress Edinburgh Aout 1968

3-ROS1 D.T.ROSS
"A generalized technique for symbol manipulation and numerical calculation".
ACM Conf. on symbol manipulation Mai 1960.

3-SHA Alan C.SHAW
"Parsing of graph-representable pictures".
JACM Vol 17 N°3 Juillet 1970 pp 453-481.

4.LANGAGES DE DEFINITION ET DE MANIPULATION DE STRUCTURES.

4-CAV J.M.CAGNAT et F.VEILLON
"Cours de programmation en langage PL/1". 3 tomes à paraître chez Armand Colin Paris.

4-DOD G.DODD "APL : a language for associative data handling in PL/1." AFIPS Conf.Proc.FJCC Vol.29 1966

4-EVA D.EVANS
"Data structure programming system".
IFIP Congress Edinburgh 1968

4-GHG GELERNTER, HANSEN et GERBERICH
"A FORTRAN-compiled list processing language FLPL."
J.ACM 7 1960 pp.87-101.

4-GPP R.E.GRISWOLD, J.F.POAGE et I.P.POLONSKY
"The SNOBOL4 programming language"
Prentice Hall New Jersey 1969.

4-GRA J.C.GRAY
"Compound data structures for computer-aided design: a survey.
Proc.ACM 1967 Thomson Books;

4-GRI R.E.GRISWOLD
"String processing and the SNOBOL4 language."
International NATO summer school Copenhague Aout 1969.

4-IBM3 IBM system reference library "PL/1 Reference manual" form C28-8201-1

4-IBM4 IBM system/360 Operating system "Assembler language"
form C28-6514-5

4-ICC "Symbolic languages in data processing". Proc.of the International Computation Center 1962 Gordon and Breack N.Y.

4-KNO2 K.KNOWLTON
"A programmer's description of L^6."
Comm.ACM Vol.9 N°8 Aout 1966.

4-LAG C.A.LANG et J.C.GRAY
"ASP : a ring-implemented associative structure package".
Comm.ACM Vol.11 N°8 Aout 1968

4-MCC J.Mc CARTHY
"LISP 1.5 Programmer's manual".
MIT Press 1962

4-NEH A.NEWELL, J.EARLEY et F.HANEY "*1 Manual"
Carnegie Mellon University Juin 1967

4-NET A.NEWELL et F.M.TONGE
"An introduction to information Processing Language V." Comm ACM

4-ROF P.D.ROVNER et J.A.FELDMANN
"The LEAP language and data structure" IFIP Congress Edinburgh 1968.

4-ROS2 D.T.ROSS
"AED-0" MIT Memorandum

4-STA T.A.STANDISH
"A data definition facility for programming languages".
Ph.D.Thesis Carnegie Institute of technologie Pittsburg 1967.

4-SUT W.R.SUTHERLAND
"The CORAL language and data structure". MIT Lincoln Lab. Tech. Report 405 Mai 1966.

4-WEI J.WEINZENBAUM
"Symmetric list processor".
Comm.ACM Vol.6 N°9 Sept 1963.