HAL open science

Étude et réalisation d'outil de programmation pour IRIS 80 mode C, LP80-metteur au point

Marc Pequignot

To cite this version:

Marc Pequignot. Étude et réalisation d'outil de programmation pour IRIS 80 mode C, LP80-metteur au point. Langage de programmation [cs.PL]. 1977. dumas-00307037

HAL Id: dumas-00307037

https://dumas.ccsd.cnrs.fr/dumas-00307037

Submitted on 28 Jul 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 au

CENTRE UNIVERSITAIRE

D'EDUCATION ET DE FORMATION DES ADULTES DE GRENOBLE

pour obtenir le titre

d' INGENIEUR du CONSERVATOIRE NATIONAL des ARTS et METIERS

par

Marc PEQUIGNOT

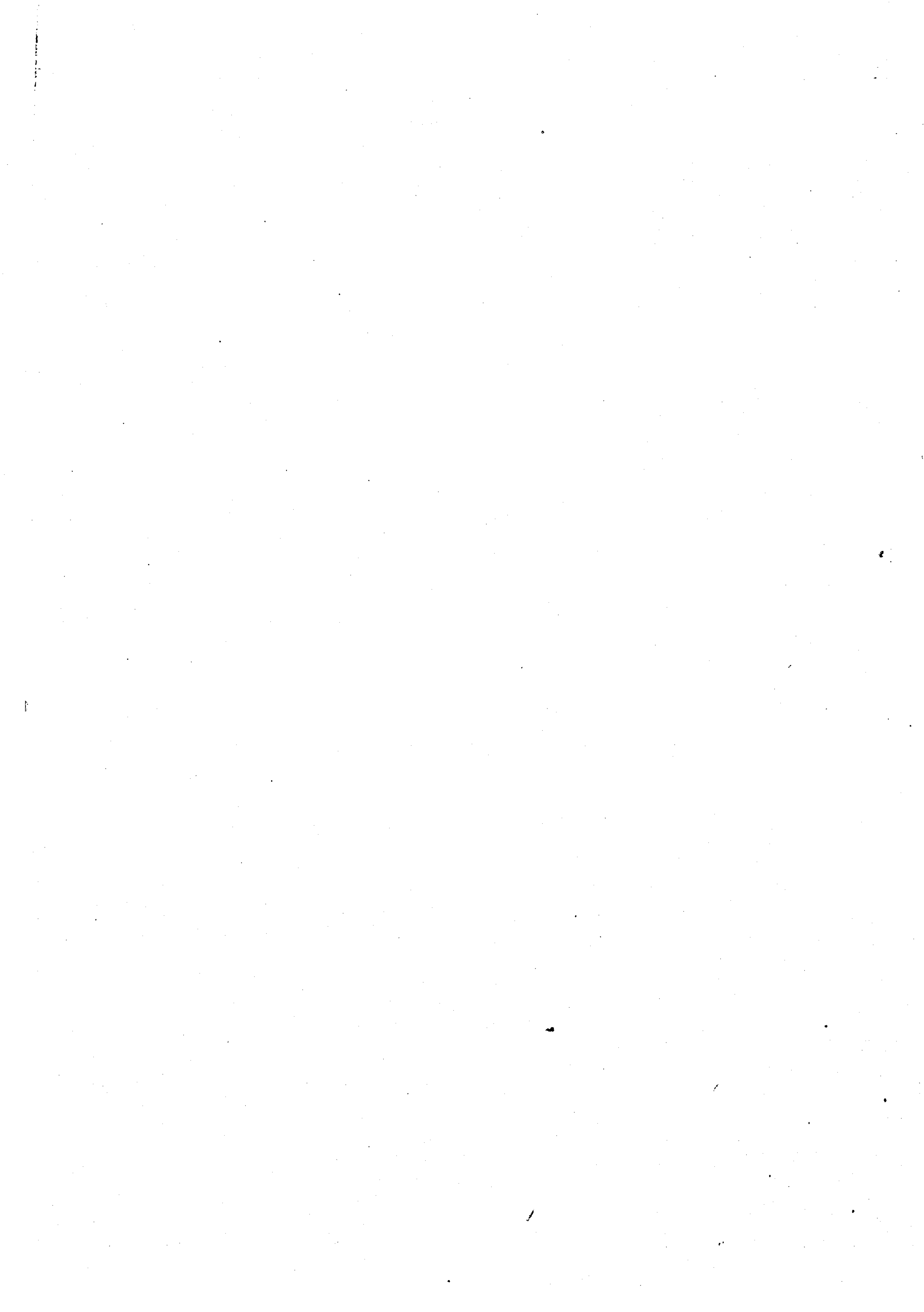
—o—

Etude et réalisation d'outils de programmation pour IRIS80 mode C LP80 – Metteur au point

—o—

Thèse soutenue le 7 Novembre 1977 devant la commission d'examen

Président	Monsieur	L. Bolliet
Président adjoint	Monsieur	P. Namian
Examinateurs	Messieurs	C. Delobel
		D. Suty



Je tiens à remercier:

Monsieur le Professeur L. BOLLIET qui a bien voulu me faire l'honneur de présider le jury et qui a toujours montré la plus grande bienveillance dans mon travail,

Monsieur le Professeur P. NAMIAN qui a accepté d'être président adjoint du jury,

Messieurs C. DELOBEL et D. SUTY pour leurs suggestions, leurs conseils et leurs précieuses critiques apportées au manuscrit,

Tous mes collègues utilisateurs des premières versions du compilateur pour leurs remarques et leur aide efficace dans la découverte des erreurs initiales.

Je voudrais aussi remercier:

Mademoiselle C. SALLUSTIO qui a assuré la frappe du manuel d'utilisation,

Le service tirage qui a réalisé ce document.

INTRODUCTION

A la fin de l'année 1974, le C.I.C.G. a reçu un IRIS 80 sur lequel pouvait fonctionner de manière standard le système SIRIS8. A cette époque, le prototype du système GEMAU paraissait suffisament développé pour qu'il semble intéressant d'en faire un produit disponible pour les utilisateurs du centre.
Ce prototype était écrit pour un ordinateur 10070 au moyen du langage LP70. Ce langage permet de générer du code pour ce type de machine ainsi que pour l'IRIS 80, mais dans un mode de fonctionnement restrictif. En effet, l'IRIS 80 possède trois modes de travail, dont les deux premiers s'apparentent à celui du 10070. Le dernier, beaucoup plus évolué en particulier pour les applications système, permet un fonctionnement en adressage basé. Un choix se posait donc entre une possibilité d'implantation rapide de GEMAU, en acceptant un fonctionnement plus restrictif, (mode B) ou dans la réécriture d'un nouveau système utilisant toutes les ressources du matériel (mode C). La deuxième solution ayant été adoptée, il fallait disposer d'un traducteur à même de produire du code basé.

Notre déception fut grande lorsque l'on s'aperçut qu'aucun logiciel livré avec la machine était capable de rendre ce service. Il devenait donc primordial de combler

cette lacune, à la fois pour les besoins particuliers du projet GEMAU, mais aussi pour les besoins généraux des utilisateurs du centre, désireux de posséder un langage parfaitement adapté à leurs problèmes.
Cependant, afin de gagner du temps lors de la réécriture du système, il semblait intéressant de définir un langage issu de LP70 afin de permettre un transport rapide, si possible transparent, des programmes déja écrits.
La mise en évidence de cette possibilité fut rapidement faite au moyen de la première version de LP80, qui n'était en fait qu'une modification, relativement importante, du compilateur LP70.
La version suivante devait être entièrement étudiée en tenant compte d'un certain nombre de contraintes:

- extension du langage
- compilateur entièrement écrit en LP80
- interfaçage bien défini, afin de permettre:
  + un développement sur SIRIS8
  + un transport rapide sur OURS/P3S, projet succédant à GEMAU
- système de mise au point simple, susceptible d'être intégré éventuellement dans le noyau du système MAS.

Ces buts ont été atteints et nous pouvons dire que la version actuelle de LP80 (V2.2) donne satisfaction à ses utilisateurs.
Bien que le langage finalement adopté pour le projet OURS soit L.I.S. (Langage d'Implémentation Système), langage qui permet aussi de générer du code basé, et qui est théoriquement mieux adapté, le compilateur LP80 a trouvé des adeptes dans d'autres projets importants:

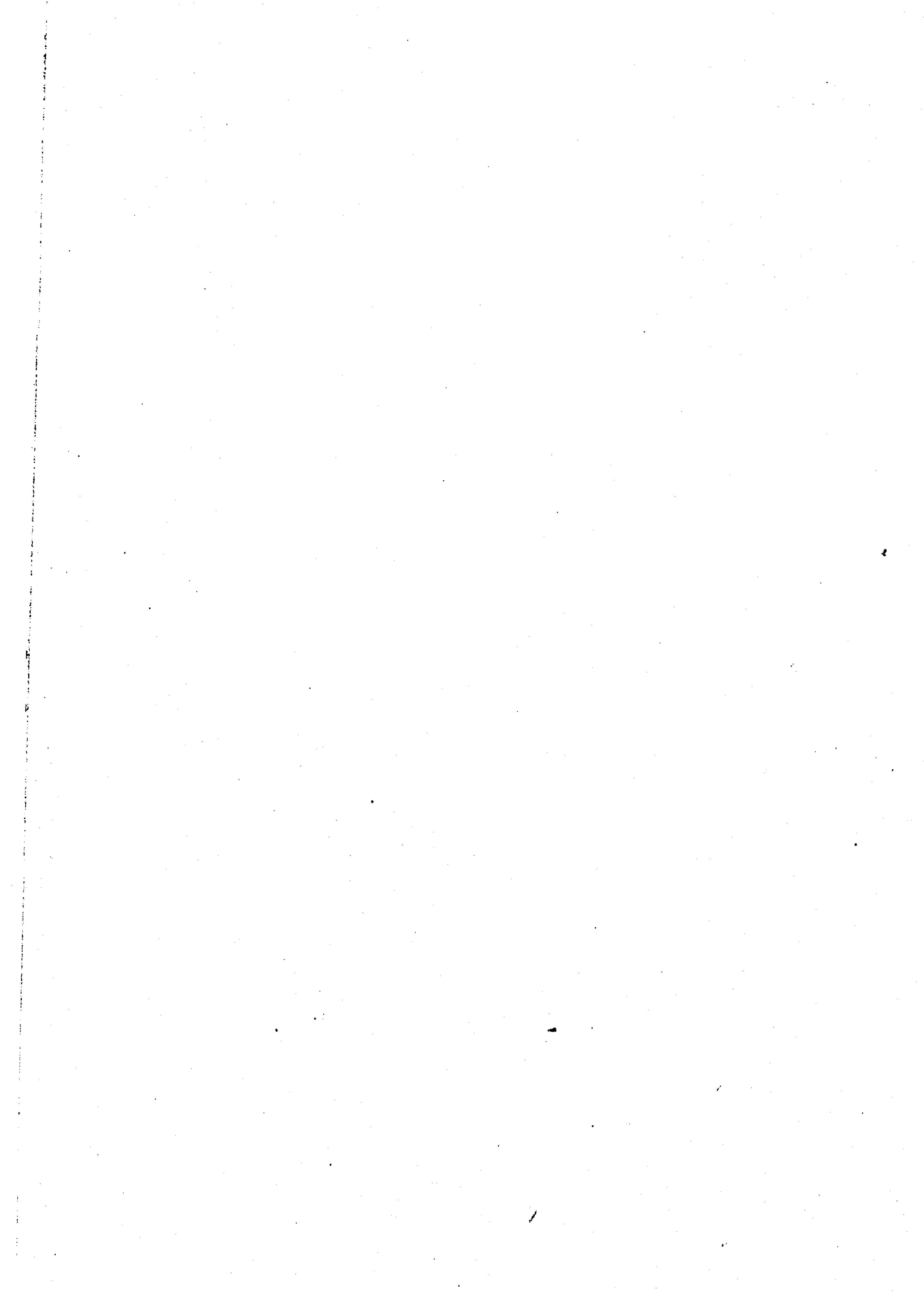
- écriture du compilateur ALGOLW
- écriture du noyau de LISP
- transport du système PIAF
- écriture des services de P3S (éditeur de textes, compilateur LP80, éditeur de liens)

- écriture de certains services sur SIRIS8, en particulier un programme de mise en forme de textes ayant permis la sortie de cette brochure (SCRIPT).

Nous présentons ici en tant que mémoire CNAM les deux manuels qui doivent accompagner tout produit informatique.

Le premier, le manuel d'utilisation, donne la définition du langage LP80, les fonctions de mise au point et doit permettre à tout programmeur d'employer au mieux les possibilités de l'IRIS80.
Afin qu'un programmeur puisse utiliser LP80 sans avoir à se référer à un autre manuel, nous avons introduit dans cette partie une brève présentation des systèmes IRIS80 et SIRIS8.

Le deuxième, le manuel d'implémentation, s'adresse aux personnes désireuses de connaitre la structure interne du compilateur et du metteur au point. C'est le manuel indispensable à la maintenance des deux produits.
Cette partie tente en outre de justifier certains choix que nous avons été ammenés à prendre.



LP 80 - METTEUR AU POINT

MANUEL D'UTILISATION

Marc PEQUIGNOT

A T T E N T I O N

Cette brochure est un tiré à part d'un manuel.

Par conséquent, la sérialisation des numéros de page n'est pas assurée entre la partie définition du langage et les annexes.

INTRODUCTION AUX SYSTEMES IRIS80 ET SIRIS8

Remarque: cette partie peut etre sautée par les lecteurs connaissant l'ordinateur IRIS80.

1. L'ordinateur IRIS80 12
  1. La Mémoire centrale 12
    1. Format des informations 12
    2. Adressage de la mémoire 14
    3. Protection de la mémoire 15
      1. Protection de la mémoire virtuelle 15
      2. Protection de la mémoire réelle 15
  2. L'Unité Centrale 16
    1. Les registres 16
    2. Les instructions 18
      1. Liste des instructions 19
      2. Format des instructions 21
      3. Adressage basé 23
      4. L'indirection 23
      5. L'indexation 23
      6. Remarques 24
  3. Les mots d'état programme 25
  4. Déroutements et interruptions 27
2. Le Système SIRIS8 28

1. L'ordinateur IRIS80

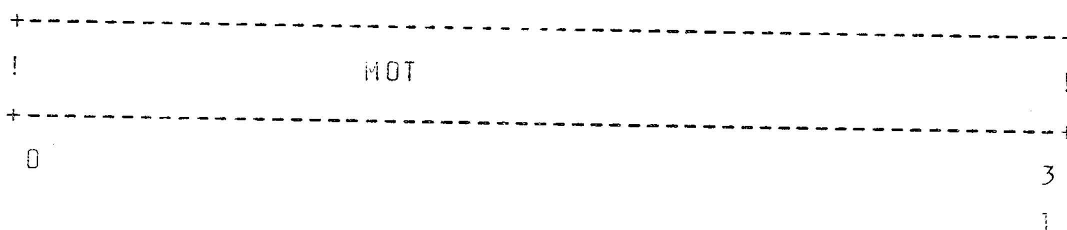
La structure de base du système IRIS80 consiste en:
- une mémoire centrale à tores de ferrites
- une ou deux unités centrales
- des unités d'échange directes ou multiplexées
- des unités de liaison
- des unités périphériques (disques à tête fixe, disques amovibles, bandes magnétiques, lecteur de cartes, imprimante, terminaux interactifs ...)

1.1. La mémoire centrale

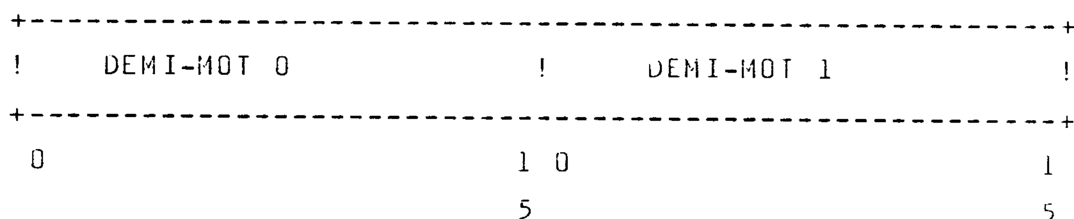
La mémoire centrale est constituée d'unités de mémoire autonomes (asynchrones) à accés multiples avec entrelacement. La capacité de la mémoire centrale peut varier de 128 Koctets (1 Koctets = 1024 octets) jusqu'à 4 Moctets (1 Moctets = 1024 * 1024 = 1048576 octets).
La mémoire centrale est partagée par les unités centrales de traitement et les unités d'échange. Les accés sont réglés par un système de priorité interne.

1.1.1. Format des informations en mémoire

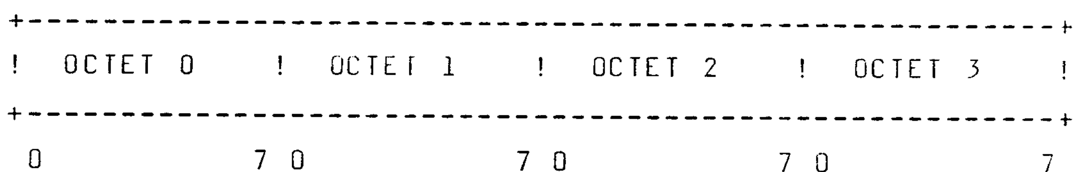
Le système est basé sur une structure de MOTS de 32 bits. Des multiples et des sous-multiples de mot sont aussi accessibles.
Les bits sont toujours numérotés de 0 à N, de gauche à droite, c'est à dire que le bit de poids le plus fort a toujours le numéro 0.



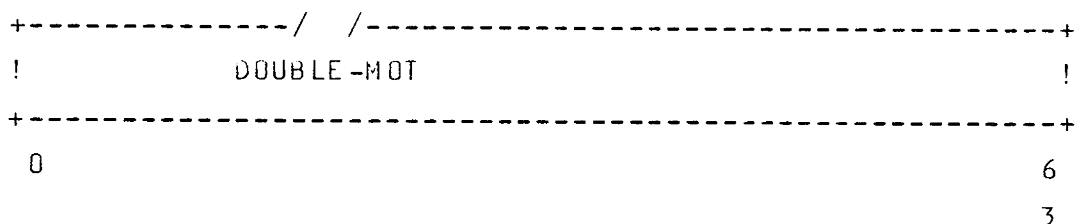
On peut découper un mot en deux demi mots de 16 bits chacun:



ou en 4 octets de 8 bits chacun:



On peut grouper 2 mots pour faire un DOUBLE-MOT débutant à une adresse paire de mots.



Contenu des différentes informations:

OCTET:

- un caractère en code EBCDIC
- un entier sans signe de valeur comprise entre 0 et 255

L'IRIS80 permet de traiter les chaines de 64k-1 octets consécutifs.

- un chiffre décimal dilaté sous forme (ZONE , CHIFFRE).
- du décimal condensé signé à raison de 2 chiffres décimaux ou un chiffre et un signe par octet.

L'IRIS80 dispose d'instructions de traitement de suite de 1 à 31 chiffres décimaux condensés.

DEMI-MOT:
- entier binaire signé, représenté en complément à 2.

MOT:
- entier binaire signé, représenté en complément à 2.
- nombre flottant de simple précision

DOUBLE-MOT:
- entier binaire signé, représenté en complément à 2
- nombre flottant de double précision

1.1.2. Adressage de la mémoire

La mémoire est constituée de mots de 32 bits numérotés à partir de 0. En mode C, l'IRIS80 peut utiliser un système de traduction dynamique des adresses et il faut distinguer l'adressage de la mémoire REELLE et celui de la mémoire VIRTUELLE.

Une adresse de mémoire réelle comporte 20 bits, ce qui autorise l'adressage 1 Mmots.

Une adresse de mémoire virtuelle comporte 24 bits ce qui autorise l'adressage de 16 Mmots.

Ceci est obtenu par la décomposition de l'espace d'adressage virtuel en:

128 SEGMENTS (numérotés de 0 à 127 sur 7 bits)
de 256 PAGES (numérotées de 0 à 255 sur 8 bits)
de 512 MOTS (numérotés de 0 à 511 sur 9 bits)

Une page fait 512 mots ou encore 2048 octets.

Un segment fait 256 pages, soit 128 Kmots, ou encore 512 Koctets.

La mémoire réelle maximale de 1 Mmots (4 Moctets) est donc constituée de 2048 pages réelles numérotées de 0 à 2047 sur 11 bits.

1.1.3. Protection de la mémoire

1.1.3.1. Protection de la mémoire virtuelle

Elle n'est active qu'en mode esclave (programme), tous les accès sont permis en mode maître (superviseur).

La protection effective (PE) est obtenue par la REUNION logique de PS (protection segment) et de PP (protection page) selon la convention suivante:

P E	autorise		interdit
0 0	tout accès	!	Rien
0 1	Lecture et exécution	!	Ecriture
1 0	Lecture seule	!	Ecriture et exécution
1 1	Aucun accès	!	Tout

1.1.3.2. Protection de la mémoire réelle

Cette protection ne s'applique qu'aux 128K premiers mots de la mémoire.

A chaque page de 2K octets est associé un Verrou de Protection VP de 2 bits implanté dans une mémoire annexe. Il s'agit seulement d'une protection contre l'écriture. La clé d'accès CA du mot d'état programme (PSD) et celle de la page référencée sont comparées afin de déterminer les possibilités d'accès.

La protection de la mémoire réelle basse agit APRES la protection de la mémoire virtuelle.

2. l'Unité Centrale

2.1. Les registres

L'unité centrale dispose de 24 registres numérotés de 0 à 23 (x'00' à x'17').

Ces registres apparaissent comme implantés en mémoire basse aux adresses mémoire de 0 à 23. Ils se substituent aux mots d'adresse réelle de 0 à 23 qui ne sont accessibles qu'à certaines instructions RD, WD ou servent à mémoriser les paramètres de l'instruction PSS. Il est possible d'éxécuter des instructions en registres.

En conséquence, il n'existe pas d'instructions registre/registre puisque les instructions registre/mémoire en tiennent lieu lorsque les adresses mémoire sont inférieures ou égales à 23.

Les 24 registres ont les fonctions spécialisées suivantes :

0-15: arithmétique virgule fixe et flottante, opérations logiques, décalages, comparaisons

12-15 : accumulateur décimal

0-7 : registre d'index (adresse d'index sur 3 bits)

0-15 : mémoires pouvant contenir des instructions exécutables

16 : compteur de longueur pour les instructions sur chaines de caractères

17-23: registres de base pour l'adressage

Les adresses x'10' à x'17' constituent un ensemble de 8 registres de base référencés B0 à B7 (adressage par 3 bits).

Le registre B0 n'est pas utilisable comme base et sert de compteur dans les opérations sur octets.

```
DEC ! HEX !        Fonctions

  0 !  00 ! -----------R0 !
  1 !  01 ! X1 !       R1 !
  2 !  02 ! X2 !       R2 !
  3 !  03 ! X3 !       R3 !
  4 !  04 ! X4 !INDEX  R4 !
  5 !  05 ! X5 !       R5 !
  6 !  06 ! X6 !       R6 !
  7 !  07 ! X7 !       R7 !
  8 !  08 !            R8 ! GENERAUX
  9 !  09 !            R9 !
 10 !  0A !            RA !
 11 !  0B !            RB !
 12 !  0C !----!       RC !
 13 !  0D !----! ACC   RD !
 14 !  0E !----! DEC   RE !
 15 !  0F !----! ------RF !
 16 !  10 ! -----------B0
 17 !  11 ! -----------B1 !
 18 !  12 !            B2 !
 19 !  13 !            B3 ! BASES
 20 !  14 !            B4 !
 21 !  15 !            B5 !
 22 !  16 !            B6 !
 23 !  17 ! -----------B7 !
```

2.2. Les instructions

Les instructions de l'Iris 80 sont de longueur FIXE sur un mot. Le code opération est constitué des bits 1 à 7. Ceci autoriserait jusqu'à 128 codes instructions différents; seuls 118 codes instruction sont valides parmi les 128 possibles. La majorité des instructions sont ininterruptibles, c'est à dire qu'aprés le début de leur exécution on est assuré de leur déroulement complet. Le déroulement de certaines instructions comprend souvent plusieurs accès mémoires consécutifs. En configuration bi-processeur, il n'est pas exclu que chacun des processeurs opère des accès mémoire incompatibles entre eux et qui rendraient le fonctionnement de certaines instructions aléatoire. Cet inconvénient est éliminé en forçant les séquences des accès mémoire. Ce mode d'acces est dit à CYCLES MEMOIRE CONSECUTIFS.

Infin, certaines instructions telles que celles qui opèrent sur les chaines de caractères peuvent avoir un temps d'exécution prohibitif vis à vis des impératifs de certaines applications en temps réel. Ces instructions sont interruptibles, mais leurs paramètres (adresse d'opérande, compteur) sont en registres et aprés interruption leur exécution est reprise avec les valeurs courantes de leurs paramètres.

2.2.1. Liste des instructions

Les codes opération et les modes d'adressage correspondant sont données dans le tableau suivant.

Tous les codes opération encadrés d'un trait continu ont le même type d'adressage indiqué par le nombre entre parenthèses avec la signification suivante :

(0) Adressage par octet
(1) Adressage par octet immédiat
(4) Adressage par demi-mot
(8) Adressage par mot
(9) Adressage par mot immédiat
(12) Adressage par double-mot

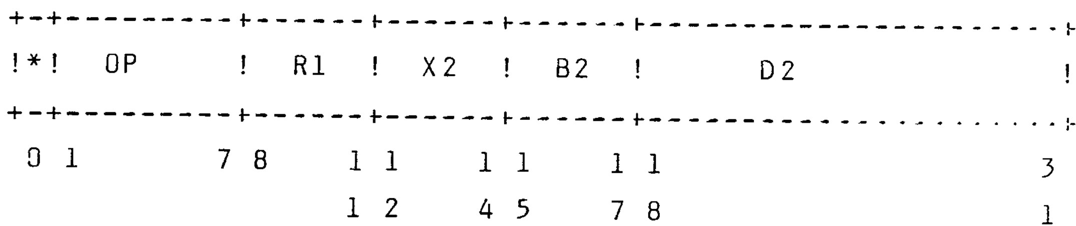
Par exemple, l'instruction MW qui a pour code opération x'37' c'est à dire x'17' + x'20" est dans le cadre (8) opère un adressage par MOT.

x	00+x	20+x	40+x	60+x
00	----	AI	TTBS	CBS
01	LIAI	CI	TBS	MBS
02	LCFI	LI	----	----
03	---(9)	MI	---(1)	EBS
04	CAL1	SF	ANLZ	BDR
05	CAL2	S	CS	BIR
06	CAL3	LAS	XW	AWM
07	CAL4	LBR	STS	EXU
08	PLW	CVS	EOR	BCR
09	PSW	CVA	OR	BCS
0A	PLM	LM (8)	LS	BAL
0B	PSM	STM	AND	INT
0C	PLS	LRA	SIO	RD
0D	PSS	----	TIO	WD
0E	LPSD	WAIT	TDV	AIO
0F	XPSD	LTSP	HIO	MMC
10	AD	AW	AH	LCF
11	CD(12)	CW	CH	CB
12	LD	LW	LH	LB
13	MSP	MTW	MTH	MTB
14	----	LVAW	----	STCF
15	STD	STW	STH(4)	STB(0)
16	----	DW	DH	PACK
17	----	MW	MH	UNPK
18	SD	SW	SH	DS
19	CLM	CLR	----	DA
1A	LCD	LCW	LCH	DD
1B	LAD	LAW	LAH	DM
1C	FSC	FSS	----	DSA
1B	FAL	FAS	----	DC
1E	FDL	FDS	----	DL
1F	FML	FMS	----	DST

2.2.2. Format des instructions

La majorité des instructions satisfont à des formats standards au nombre de quatre. Ces formats sont dénotés RX, AX, RI et IX.

FORMAT RX Registre/Mémoire (base, déplacement) indexé et indirect
OP* r1,d2(x2,b2)

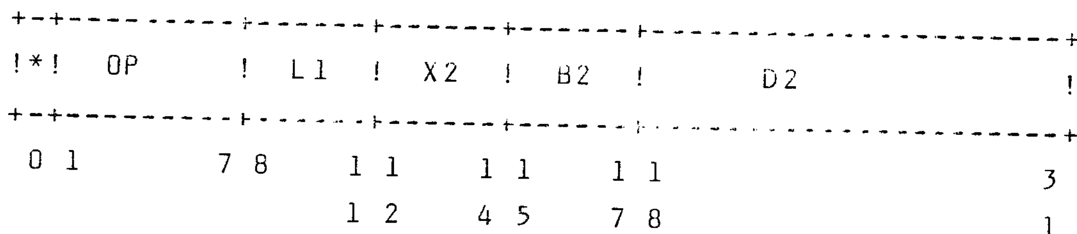


Bit 0 : INDIRECTION (0=sans, 1=avec)
Bits 1-7 : Code opération
Bits 8-11 : Registre général (1-15). Premier opérande

L'adressage du second opérande est calcul
é à partir de:
Bits 15-17 : Registre de base (1-7), 0 signifie pas de base.
Bits 18-31 : Déplacement par rapport à la base B2 (0-16383) sur 14 bits
Bits 12-14 : Registre d'index (1-7), 0 signifie pas d'indexation

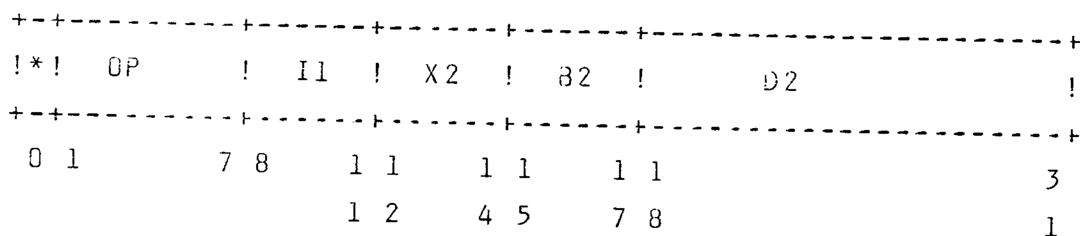
N.B. L'adressage par base-déplacement n'est pas lié directement à la notion de page telle qu'elle est décrite plus haut à propos de l'espace d'adressage virtuel. A savoir qu'une base couvre 16K mots, soit 64K octets ou encore 32 pages.

FORMAT AX Longueur/Mémoire(base,déplacement) indexé et indirect
OP* l1,d2(x2,b2)



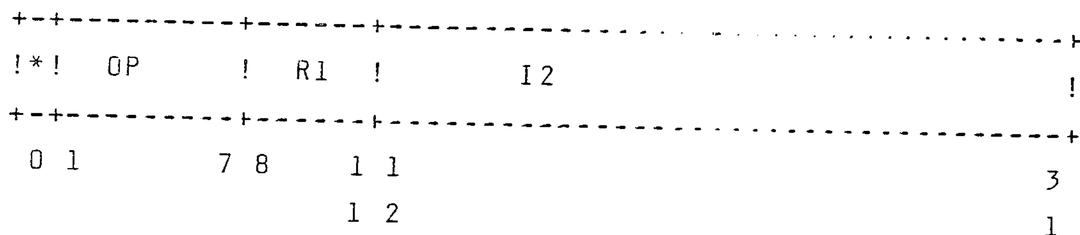
Ce format, utilisé pour les opérations en arithmétique décimale, est identique au format RX si ce n'est que la machine ne disposant que d'un accumulateur décimal (constitué des 4 registres généraux x0C à x0F), son emploi est défini implicitement par le code opération, et les 4 bits 8 à 11 (L1) indiquent la longueur de l'opérande.

FORMAT IX Opérande immédiat/opérande mémoire
OP* i1,d2(x2,b2)



Analogique au format RX si ce n'est que les bits 8 à 11 désignent comme premier opérande non pas un registre, mais un entier signe sur 4 bits (de -8 à +7).

FORMAT RI Registre/Opérande immédiat OP r1,i2



La zone I2 longue de 20 bits (12 à 31) contient le second opérande sous forme de nombre algébrique binaire représenté en complément à 2. Le bit 12 apparaît donc comme

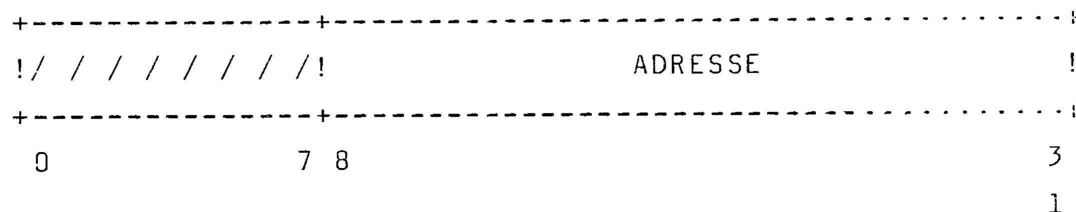
bit de signe. L'adressage indirect (bit 0) est interdit dans ce format et provoque un déroutement pour instruction illégale (voir plus loin les déroutements).

2.2.3. Adressage basé

Adressage du second opérande dans les instructions de type X: (base, déplacement) indexé et indirect:
1 - On calcule d'abord l'adresse initiale par addition du contenu de la base b2 (on prend pour valeur 0 quand il s'agit de B0) au déplacement d2.
2 - On applique ensuite l'indirection
3 - Puis enfin l'indexation

2.2.4. L'indirection

Si l'indirection est demandée par la présence du bit 0 de l'instruction, l'adresse initiale désigne alors l'adresse d'un mot d'indirection dont le format est le suivant :



L'adresse de 24 bits est une adresse virtuelle de mot, dont le contenu devient l'adresse initiale.

2.2.5. L'indexation

L'indexation a un double rôle:
- faciliter l'adressage d'éléments de tableaux à une dimension
- autoriser l'adressage des octets, demi-mots et doubles mots alors que la machine est orientée mot.

La nature de l'opérande mémoire d'une instruction RX (octet, 1/2 mot, mot, double mot) est déterminée par son code opération. Lors de l'opération d'addition du contenu du registre d'index X2 il est multiplié par 1/4, 1/2, 1, 2 pour les opérations portant respectivement sur octet, 1/2 mot, mot, double mot. (Une multiplication ou une division par 2 d'un entier binaire correspond respectivement à un décalage de 1 position vers la gauche ou la droite). L'adresse virtuelle de mot décrite précédemment passe respectivement à 26, 25, 24, 23 bits, ces nombres correspondent au nombre de bits de poids faible considérés dans l'index.

Ainsi le contenu d'un registre d'index permet toujours de référencer le ième élément d'une liste quelle que soit sa nature. Ceci autorise entre autres comptage et indexation simultanés.

Bien retenir que le processus d'adressage opère dans l'ordre indirection puis indexation: il s'agit d'un système à POST_INDEXATION.

Enfin, dans certaines opérations, où le second opérande ne désigne pas une adresse d'opérande, tout ou partie du registre d'index X2 indique un compteur ou un opérande immédiat.

Le coût de l'indexation est constant et de l'ordre de la microseconde.

2.2.6. Remarques

Puisque les registres généraux de l'IRIS 80 sont banalisés (accumulateurs virgule fixe et flottante) il n'est pas nécessaire de particulariser les opérations de chargement et de déchargement à partir des registres en fonction du type d'opérande.

Dans certaines opérations de type RX le registre R1 désigne une paire de registres.

Une PAIRE COMPLETE est constituée d'un registre général de numéro pair (le REVen) et du registre général de numéro immédiatement supérieur, donc impair (le RODd). Une paire REDUITE est constituée d'un seul registre impair (ROD).

Une PAIRE, sans autre qualificatif, désigne une paire complète ou une paire réduite.

Les opérations sur paire réduite de registres fonctionnent en général de façon dégénérée sur le ROD seul par rapport au cas normal.

Enfin, lorsque le code opération implique un opérande double mot en mémoire, le bit des unités de l'adresse est ignoré.

2.3. Mots d'état programme PSD

Nous ne chercherons pas à justifier le mnémonique PSD (Program Status Doubleword) sachant que l'état d'un programme IRIS 80 est décrit dans TROIS mots. De plus le logiciel travaillant sur des doubles mots ces trois mots sont arrondis à quatre pour se caser dans deux double mots consécutifs, en quelque sorte un quadruple mot (sur frontière de double mot).

Seuls les trois premiers mots jouent le rôle usuel de mots d'état programme, le quatrième étant un mot de remplissage. (Voir à ce sujet l'instruction XPSD). Le seul cas d'utilisation de ce quatrième mot est d'indiquer l'adresse virtuelle de l'instruction ayant provoqué un déroutement.

Les mots d'état programme comportent 27 champs fonctionnels contenant:

- Des informations sur le déroulement de certaines instructions
- Des masques pour autoriser ou inhiber certains modes de fonctionnement
- Des adresses (compteur ordinal, pointeur de la table des segments)

Il faut toujours distinguer entre le PSD actif constitué d'un certain nombre de registres matériels dans l'UC et son image en mémoire soit avant, soit après une instruction de chargement ou de sauvegarde.

Comme beaucoup de machines, l'IRIS 80 distingue deux modes de déroulement des programmes usuellement notés de l'une des façons suivantes:

Superviseur / Programme
Maître / Esclave
Moniteur / Normal

Ce choix est indiqué par

- MN : Mode Normal. Bit 8: 0 = Maître, 1 = Esclave

En mode maître, toutes les instructions sont valides, en mode esclave les instructions dites PRIVILEGIEES provoquent un DEROUTEMENT.

L'IRIS80 dispose d'une mémoire de comparaison généralisée sous forme de 4 bits IC1 à IC4: Code condition (CC). Selon les instructions, aucun, tout ou partie des bits IC sont susceptibles d'être modifiés. Dans certaines instructions le code condition est utilisé étrangement comme compteur.

Par ailleurs, le PSD comporte 4 indicateurs de virgule flottante:

- AF : Bit 4 - Arrondi
- PF : Bit 5 - Précision
- ZF : Bit 6 - Zéro
- NF : Bit 7 - Normalisation

Des masques peuvent inhiber les déroutements au cours de certains types d'opérations. Ce sont:

- MD : Bit 10 - Masque décimal :
  1 = déroutement possible
  0 = déroutement inhibé
- MA : Bit 11 - Masque arithmétique
  1 = déroutement possible
  0 = déroutement inhibé

Nous ne décrirons pas les autres champs du PSD dont la connaissance n'est pas indispensable à la programmation élémentaire. Le lecteur pourra se reporter à la brochure du constructeur.

2.4. Déroutements et interruptions

L'IRIS 80 distingue entre les événements internes à l'unité centrale (opération illégale, erreur hardware ...) qui provoquent des DEROUTEMENTS et des événements externes (fin d'entrée/sortie, horloge, erreur mémoire ...) qui provoquent des INTERRUPTIONS.

L'IRIS 80 peut reconnaître 10 classes de déroutements:

-1 Opération interdite
-2 Débordement pile
-3 Débordement virgule fixe
-4 Erreur virgule fixe
-5 Erreur virgule flottante
-6 Erreur décimale
-7 Déroutement programme
-8 Instruction CAL
-9 Compteur temps réel interne et déroutement programme
-10 Surveillance secteur

Le mécanisme de traitement des déroutements et des interruptions n'est pas décrit ici.

Sous SIRIS8, le programmeur peut récupérer le contrôle dans la plupart des cas (macro-instructions :TRAP, :INT et conditions anormales de traitement des fichiers ainsi que la gestion du temps).

2. Le système SIRIS8

SIRIS8 est un système de multiprogrammation fonctionnant sur l'ordinateur IRIS80. Il offre simultanément deux types de services:
- un service de traitement par lots (BATCH)
- un service de traitement conversationnel (T.S.)

La coexistence des deux possibilités offre aux utilisateurs une plus grande souplesse d'emploi. La compatibilité entre les deux moyens d'accès est théoriquement totale.
SIRIS8 permet la gestion de tâches fonctionnant dans les deux modes B ou C.
Dans les deux cas, les services offerts sont sensiblement identiques et sont ceux généralement disponibles sur les grands systèmes.
Les différences les plus importantes sont les suivantes:
- absence d'un générateur de code basé
- absence d'un système de mise au point pour le mode C
- possibilité d'utiliser un espace virtuel d'adressage plus grand en mode C

.... mais impossibilité d'utiliser cet adressage étendu par des taches conversationnelles.

Le point le plus important à noter est la gestion de la mémoire d'un espace utilisateur.

En mode C on peut théoriquement disposer de 128 segments.
La structure de SIRIS8 entraine que certains sont inaccessibles soit en partie, soit totalement.
Le segment zéro (détaillé plus loin) est ainsi partagé entre le moniteur (de l'adresse 0 à x3FFF) et l'utilisateur. Les segments 1 à 7 sont réservés à l'usage exclusif du moniteur.
Enfin, les segments 8 à 127 sont -théoriquement- employables par la tâche utilisateur.
Théoriquement, car la version C09 de SIRIS8 n'étant pas paginée, on ne peut jamais obtenir de l'espace que jusqu'à

concurence de la totalité de la mémoire réelle.
Cette absence de pagination entraine une restriction importante en T.S.. Celui-ci, introduit aprés coup sur un système ancien n'ayant pas la possibilité matérielle de paginer fonctionne grâce à un mécanisme de "swapping" complet de l'espace utilisateur.
Pour des raisons d'efficacité, seul le segment 0 est donc employable. Le mécanisme d'adressage étendu n'est pas utilisable en T.S..

Description du segment 0

```
+------------------------------+
!                              !  0
!                              !
!                              !
!   MONITEUR SIRIS8            !
!                              !
!                              !
+------------------------------+
!                              !  x3E00
!                              !
!   Zone programme             !
!                              !
!                              !
+------------------------------+
!                              !
!                              !
!   Zone dynamique privée      !
!                              !
! - - - - - - - - - - - - - -  !
!              !               !
!              !               !
!              V               !
!                              !
!                              !
!              ^               !
!              !               !
!              !               !
! - - - - - - - - - - - - - -  !
!                              !
!                              !
!   Zone dynamique commune     !
!                              !  x1FFFF
+------------------------------+
```

LE LANGAGE LP80 - LE METTEUR AU POINT

1. Introduction 35
2. Notations et définitions de base 35
  1. Opérandes 35
  2. Symboles de base 36
  3. Commentaires 37
3. Identifieurs 38
4. Valeurs 39
  1. Nombres entiers 39
  2. Nombres flottants 40
  3. Chaines 41
  4. Valeurs booléennes 41
5. Déclarations 42
  1. Registres 42
  2. Mémoires 42
    1. Booléens 42
    2. Variables 44
    3. Pointeurs 47
    4. Textes 48
    5. Tableaux 49
    6. Piles 51
  3. Constantes 52
  4. Fonctions assembleur 53
  5. Conditions 54
  6. Etiquettes 55
  7. Procédures 56
6. Instructions 57
  1. Blocs 57
  2. Instructions d'affectation 58
    1. Désignation d'une variable 59

2. Désignation d'une variable indirecte 60
3. Désignation des registres 61
4. Désignation des bases 61
5. Assignation simple de registre 62
6. Assignation simple de variable 65
7. Assignation simple de base 66
8. Assignation multiple 66
9. Echange de mot 67
10. Assignation simple de logique 68
11. Assignation multiple de logique 70
3. Instructions de contrôle 71
1. Instruction IF 71
2. Instruction CASE 72
3. Instruction WHILE 74
4. Instruction FOR 75
5. Instruction REPEAT 76
4. Instructions de branchement 77
1. Instruction LOOP 77
2. Instruction GOTO 77
3. Instruction CYCLE 78
4. Instruction EXIT 79
5. Instruction NULL 80
6. Instruction STOP 80
7. Appel de fonction 81
1. Fonction assembleur 81
2. Fonctions composées 82
1. Fonctions MOVE,COMP,TRANS 82
2. Fonctions PUSH,PULL 83
3. Fonctions STORE,LOAD 84
4. Fonction RETURN 85
5. Fonction GEN 86
8. Appel de procédure 87
9. Adressage et segmentation 88
1. Segment principal 88
2. Segment secondaire 89
3. Déclaration de procédure externe 91

4. Déclaration de point d'entrée 92
5. Déclaration de blocs de données 92
1. Déclaration de GLOBALDATA 92
2. Déclaration de COMMONDATA 94
3. Déclaration de DUMMYDATA 95
6. Structure du code généré 96
1. Structure générale 96
2. Structure détaillée de la section programme 97
10. Macro-instructions de SIRIS8 100
1. Fonctions 100
2. Liste des macro-instructions 100
3. Notation 101
4. Syntaxe des macro-instructions 102
5. Code généré 106
11. Facilités de mise au point 107
1. Les ordres de trace 107
1. Déclaration de trace 107
2. Déclaration de fin de trace 109
3. Instruction de silence 109
4. Instruction de fin de silence 110
2. Instruction SNAP 110
3. Accompagnateur LP80 114
1. Buts 114
2. Mise en oeuvre 114
3. Requêtes de mise au point 117
4. Restrictions sous SIRIS8 121
12. Mise en oeuvre du compilateur 122
1. Commandes au compilateur 122
2. Options du compilateur 123
3. Listes fournies 127
13. Traitement des erreurs 129
1. Erreurs sur les options 129
2. Erreur de gestion des fichiers 130
3. Erreurs dues au compilateur 130
4. Erreur de syntaxe 131
5. Erreurs de sémantique 131

AVERTISSEMENT

Tous les exemples qui figurent dans cette brochure sont extraits des listes de compilation d'un ensemble de programmes élaborés en vue d'illustrer la présentation du langage.

Nous espérons que cela permettra de réduire au maximum les erreurs qui n'auraient pas manqué de se produire sans l'adoption de cette méthode.

Le lecteur voudra bien cependant nous faire toutes les remarques qu'il jugera utiles.

Nous le prions par ailleurs de bien vouloir nous passer l'anglicisme consistant à employer identifieur au lieu d'identificateur.

1. INTRODUCTION

LP80 est un langage de programmation spécialement étudié pour la programmation en mode C sur les ordinateurs IRIS 80. Il contient toutes les facilités du langage machine mais présente en plus une structure définie par une syntaxe récursive.

Le compilateur est en un seul passage.
Il traduit le code source en modules objets dans le format défini pour le système SIRIS8.

Les buts visés par le langage LP80 sont :
- avoir accès à toutes les possibilités offertes par le système IRIS80.
- permettre l'écriture et l'amélioration de programmes dans un langage clair,
- encourager les utilisateurs à écrire simplement et de manière compréhensible.

Pour programmer correctement en LP80, il est nécessaire de connaître le système IRIS80.

2. NOTATIONS ET DEFINITIONS DE BASE

2.1. Opérandes

Le langage LP80 permet de référencer 2 sortes d'opérandes :
- les opérandes en mémoire centrale
- les registres

2.1.1. Les éléments de mémoire centrale

Le langage distingue 7 types d'éléments de mémoire qui diffèrent selon le type de la valeur qu'ils peuvent contenir :
- booléen : 8 bits ou 1 octet pouvant prendre 2 valeurs logiques VRAI ou FAUX
- byte ou octet : 8 bits pouvant contenir un nombre entier non signé ou un caractère.

- short : 1/2 mot de 2 octets pouvant contenir un nombre entier signé de 16 bits
- word : mot de 4 octets pouvant contenir un nombre entier signé de 32 bits
- long : double mot de 8 octets pouvant contenir un nombre entier signé de 64 bits
- real : mot de 4 octets pouvant contenir un nombre flottant simple précision.
- longreal : double mot de 8 octets pouvant contenir un nombre flottant double précision.

2.1.2. Les registres

Le matériel IRIS 80 offre deux types de registres :
- les registres généraux
- les registres de base

A l'intérieur de chacune de ces catégories on distingue des sous-groupes :

. les registres généraux (emplacement mémoire 0 à 15)
  Il existe 16 registres généraux. Les registres 1 à 7 peuvent servir d'index
  Les registres 12 à 15 peuvent servir d'accumulateur décimal.
  Les opérations flottantes s'effectuent dans les mêmes registres généraux.

. Les registres de base (emplacement mémoire 16 à 23)
  Les 8 registres de base sont notés de 0 à 7
  La base 0 est utilisée dans les opérations sur les chaînes.
  Les bases 1 à 7 servent à l'adressage.

2.2. Symboles de base du langage

- les lettres de A à Z
- les chiffres de 0 à 9
- les symboles spéciaux :
  + - / * @ % _ ' " : = ; , < > ( ) & | .
- espace.
- les mots-clés (cf. Annexe 3)

2.3. Commentaires

Les commentaires permettent d'obtenir une plus grande clarté des programmes. En LP80 un commentaire est une suite de caractères quelconques placée entre deux perluets (&).

Il s'ensuit que le caractère & ne doit pas figurer dans un commentaire.

Le commentaire agit comme séparateur d'unités syntaxiques au même titre que le caractère d'espacement.

Exemple :

```
&*** DEBUT DE COMMENTAIRE ...
.......... SUITE ............
.......... ET FIN **********&
```

3. IDENTIFIEURS

1. Syntaxe

```
<identifieur>   ::=  <lettre>
                   | <identifieur> <lettre>
                   | <identifieur> <chiffre>
                   | <identifieur> <séparateur>
<lettre>        ::=  A|B| ........ Z
<chiffre>       ::=  1|2| ........ 9
<séparateur>    ::=  _|'   (caractères souligné ou apostrophe)
```

La taille d'un identifieur est limitée à 63 caractères.

2. Sémantique

Les identifieurs servent à identifier les différents éléments de mémoire, les registres ou les procédures.

Tout identifieur utilisé dans un programme LP80 doit être déclaré avant d'être référencé.

Il existe un ensemble d'identifieurs prédéclarés qui peuvent être utilisés sans déclaration préalable et qui seront présentés plus loin.

```
BYTEO    SHØRTO    WØRDO    LØNGO
RO    R1    R2 .............. RF
RO_1    R2_3    .............. RE_F
FO    F1    F2 .............. FF
FO_1    F2_3    ..............
BO    B1        .............. B7
```

ainsi que certaines fonctions assembleur.

Les mots-clés ne doivent pas être utilisés comme identifieur.

Ce sont des mots réservés.

3. Exemples

```
TETE
EN_TETE
POINT_D'ARRET
TABLE1            & SONT DES IDENTIFIEURS VALIDES &
```

4. VALEURS

4. 1. Nombres entiers

1. Syntaxe

```
<nombre entier> ::= <nombre décimal entier>
                  | <nombre hexédécimal>
<nombre décimal entier>  ::=  <suite de chiffres>
                           |  <suite de chiffres>  L
<suite de chiffres>      ::=  <chiffre>
                           |  <chiffre> <suite de chiffres>
<nombre hexadécimal>     ::= #  <suite de caractères hexadécimaux>
                           |#<suite de caractères hexadécimaux>  L
<suite de caractères hexadécimaux>  ::=  <caractère hexadécimal>
                                       | <caractère hexadécimal>
                                             <suite de caractères hexadécimaux>
<caractère hexadécimal>  ::=  <chiffre>
                           |A|B|C|D|E|F
```

2. Sémantique

Un nombre hexadécimal est un nombre entier représenté en base 16. Il occupe un ou deux mots mémoire selon sa taille. Celle-ci est donnée :

- soit par le nombre de chiffres hexadécimaux

  1 à 8 chiffres donnent un mot

  8 à 16 chiffres donnent un double-mot

- soit par la lettre L qui force l'occupation d'un double-mot

Un nombre hexadécimal forme un tout qui ne doit pas être coupé par un blanc. (espace).

Un nombre décimal entier peut aussi occuper un ou deux mots.

3. Exemples

#3E
#3EL
#123456789ABCDEF0
10
3000000000
2L & SONT DES NOMBRES ENTIERS &

4.2. Nombres flottants

1. Syntaxe

<nombre flottant> ::= <mantisse>
| <mantisse> <exposant>
| <mantisse> L
| <mantisse> <exposant> L
<mantisse> ::= <nombre décimal entier> . <nombre décimal entier>
| <nombre décimal entier> .
| . <nombre décimal entier>
<exposant> ::= E <nombre décimal entier>
| E <signe> <nombre décimal entier>
<signe> ::= + | -

2. Sémantique

De la même manière que les nombres entiers, les nombres flottants peuvent être courts ou longs.
La lettre L permet de forcer une valeur longue

3. Exemples

3.14159
.314159
1.
0.1E-28
.1E28
1.E-29
2.E+4
0.123456789
3.14159L
4.5E+35L
0.L & SONT DES NOMBRES FLOTTANTS VALIDES &

4.3. Chaînes de caractères

1. Syntaxe

<chaîne de caractères> ::= " <suite de caractères> "

<suite de caractères> ::= <symbole de base>
| <suite de caractères> <symbole de base>
| <vide>

<vide> ::=

2. Sémantique

Une chaîne de caractères est une suite éventuellement vide de caractères placés entre deux guillemets (").
Si le caractère " doit apparaître dans la chaîne, il doit être doublé.

3. Exemples

```
"CHAINE DE CARACTERES"
"AVEC GUILLEMETS ("")"
""                              & CHAINE VIDE &
```

4.4. Valeurs booléennes

1. Syntaxe

<valeur booléenne> ::= TRUE | FALSE

2. Sémantique

Les valeurs booléennes sont les valeurs VRAI et FAUX qui peuvent venir lors de l'affectation de logiques au bas de la déclaration de logiques.

3. Exemples

```
TRUE
FALSE          & SONT LES DEUX VALEURS BOOLENNES &
```

5. DECLARATIONS

Les déclarations servent à associer des identifieurs (3.1) aux entités utilisées dans le programme, à attribuer certaines propriétés permanentes à ces entités (type) et à déterminer leur taille.
La portée des déclarations est liée à la structure de blocs.

5.1. Déclaration de registres

Tous les registres de l'IRIS80 -registres généraux et bases- sont utilisables en LP80.
Ils sont déclarés automatiquement au niveau de bloc 0 (bloc englobant le programme utilisateur) et sont utilisés par :

R0, R1, R2, ... RF : type entier court
R0_1, R2_3 ...... RE_F : type entier long
F0, F1, F2 ...... FF : type flottant court
F0_1, F2_3, ... FE_F : type flottant long
B0, B1; B2, ... B7 : registres de base.

On peut déclarer des registres par synonymie.
Les registres prédéclarés peuvent être redéclarés dans des blocs intérieurs.

5.2. Déclaration de mémoires

5.2.1. Déclaration de logique

1. Syntaxe

<déclaration de logique> ::= LOGICAL <liste de déclaration de logique>
<liste de déclaration de logique> ::= <élément de déclaration de logique>
| <élément de déclaration de logique> , <liste de déclaration de logique>

<élément de déclaration de logique> ::= <identifieur non déclaré>
| <identifieur non déclaré> := <valeur booléenne>
| <identifieur non déclaré> SYN <synonyme1>

<synonyme1> ::= <nombre entier>
| <identifieur de mémoire>
| <identifieur de mémoire> <déplacement complémentaire>

<déplacement complémentaire> ::= (<valeur entière>)

<valeur entière> ::= <nombre entier>
| - <nombre entier>

2. Sémantique

Une déclaration de booléen introduit les identifieurs correspondants et leur associe un octet de mémoire.
Cet octet de mémoire est réservé dans la zone de données. Cependant si la déclaration est faite par synonymie (redéfinition d'une zone mémoire déjà déclarée ou bien en adresse absolue) aucune réservation n'est effectuée.
Le nombre entier utilisé dans une déclaration avec synoyme doit être inférieur ou égal à la valeur #3FFF.
L'initialisation d'un logique permet de réserver la zone mémoire correspondante avec les valeurs vrai (TRUE) et faux (FALSE).

3. Exemple

```
WORD VARIABLE;

LOGICAL BOOL1 SYN #348;
LOGICAL BOOL2 SYN VARIABLE;
LOGICAL BOOL3 = TRUE,
        BOOL4 = FALSE;
```

5.2.2. Déclaration de variables

1. Syntaxe

```
<déclaration de variable>  ::= <type> <liste déclaration de variable>
<type>  ::=  BYTE | SHORT | WORD | LONG | REAL | LONGREAL
<liste déclaration de variable>  ::= <élément déclaration de variable>
                                   | <liste déclaration de variable> ,
                                     <élément déclaration de variable>
<élément déclaration de variable>  ::= <identifieur non déclaré>
                                     | <identifieur non déclaré> =
                                       <initialisation>
                                     | <identifieur non déclaré> SYN
                                       <synonyme2>
<initialisation>  ::=  <chaîne>
                     | <nombre>
                     | <signe><nombre>
                     | <adresse>
                     | <déplacement>
<nombre>          ::=  <nombre entier>
                     | <nombre flottant>
<adresse>         ::=  @<identifieur d'adresse>
<déplacement>     ::= % <identifieur déplacement>
<identifieur d'adresse> ::= <identifieur de segment externe>
                          | <identifieur déplacement>
<identifieur déplacement> ::= <identifieur d'étiquette>
                            | <identifieur de procédure>
                            | <identifieur de pile>
                            | <identifieur de mémoire>
                            | <identifieur de mémoire> <déplacement complé-
                                                              mentaire>
<identifieur de mémoire>  ::= <identifieur de logique>
                            | <identifieur de variable>
                            | <identifieur de pointeur>
                            | <identifieur de pile>
<synonyme2>               ::= <synonyme1>
                            | <identifieur de registre>
                            | <identifieur de base>
```

2. Sémantique

Pour toutes ces déclarations à l'exception des déclarations synonymes (SYN), le compilateur réserve des cellules de mémoire dans une zone de données distincte de la zone où est engendré le code.

A un identifieur de variable est associé un des types suivants :
- type BYTE - un octet
- type SHORT - un demi-mot
- type WORD - un mot
- type LONG - un double-mot
- type REAL - un mot
- type LONGREAL - un double-mot

Un octet, un demi-mot, un mot, sont toujours placés à une adresse de mot. Un double-mot est placé à une frontière paire de mots. Il s'ensuit qu'une déclaration de BYTE ou de SHORT réserve toujours un mot, une déclaration de LONG ou LONGREAL deux ou trois mots.

Le type d'un identifieur de variable permet au compilateur de générer l'instruction afférente à ce type. Pour une variable de type SHORT des instructions sur demi-mot seront générées.

La syntaxe des identifieurs de tableau, de pointeur, de pile, de procédure et de segment externe sera donnée ultérieurement.
L'initialisation doit être en accord avec le type de la déclaration.
En particulier, une adresse est une quantité de type mot. Cette adresse est résolue suivant le type de la quantité dont elle est l'adresse et suivant les règles d'adressage de l'ordinateur. Le nombre entier qui peut apparaître dans une "adresse" est un déplacement-mot qui est ajouté à l'adresse mot de l'identifieur avant résolution.
L'initialisation de variables au moyen d'une adresse permet d'obtenir la valeur absolue de l'adresse d'une variable résolue selon le type de cette variable. La valeur absolue doit tenir sur un mot. Seules les variables de type WORD ou LONG peuvent être initialisées avec l'adresse d'une variable.
L'initialisation de variables au moyen d'un déplacement permet d'obtenir la valeur du déplacement en mots d'une variable considérée. Cela peut être utile lorsque l'on veut gérer des ensembles de données importants, comportant des chaînages relatifs de données. Les variables de type SHORT peuvent être initialisées par des déplacements.

Une déclaration de synonyme ne peut donner lieu à initialisation.

- Synonyme d'un nombre entier

  La déclaration synonyme d'un nombre entier permet de faire de l'adressage absolu.

  L'identifieur ainsi déclaré a pour adresse le nombre entier spécifié à droite de SYN.

- Synonyme d'un registre ou d'une base

  Cette déclaration permet de renommer les identifieurs standard de registres, ce qui augmente en général la lisibilité des programmes.

- Synonyme d'un identifieur de mémoire

  Cette déclaration permet de considérer une même donnée comme un objet de plusieurs types ou formée d'éléments de plusieurs types.

  Le nouvel identifieur ainsi déclaré a pour adresse mot (adresse non résolue), l'adresse de l'identifieur à droite de SYN, augmentée le cas échéant d'un nombre entier considéré comme un déplacement mot quel que soit le type.

Il existe un certain nombre de variables prédéclarées.

SYSLNGDATA C'est une variable de type mot qui contient le nombre total de mots de variables locales déclarées dans le segment courant. Cette variable trouve son emploi en relation avec l'option REENT de définition de segment (§9.2).

BYTEO, SHORTO, WORDO, LONGO sont quatre variables de type BYTE, SHORT, WORD et LONG déclarées synonymes de l'adresse 0. Cela permet de faire un adressage direct de variables quelconques

- soit au moyen d'un index
- soit par l'emploi d'une base explicite (§6.2.2.).

3. Exemples

```
SHORT S1,S2,S3;
BYTE B01 = "A";
LONG L1 = #3E;
WORD RINDEX SYN R6;
WORD W1 = "ABC",
     W2 = @L1,
     W3 = %B01;
WORD BASE SYN B6;
BYTE B02 SYN W1;
WORD W4 = @W2(1);
```

5.2.3. Déclaration de pointeur

1. Syntaxe

<déclaration de pointeur> ::= POINTER <type> <liste déclaration de variable>

2. Sémantique

La déclaration POINTER est en général utilisée pour désigner une variable par indirection : le type de cette variable est fixé par le type de la déclaration POINTER. Dans ce cas, il importe de charger ou d'initialiser le pointeur avec l'adresse de cette variable.
Ce qui précède sur les déclarations de variables simples vaut pour les déclarations de pointeur. L'adresse d'un pointeur est de type mot.

L'usage de l'indirection en mode C est déconseillé. En effet, le coût de l'exécution d'instructions indirectes est relativement élevé, et on peut obtenir un résultat identique par l'emploi des bases.

3. Exemple

```
POINTER WORD PTR1;
POINTER WORD PTR2,PTR3;
POINTER LONG PTR4 = @L1;
POINTER BYTE PTR5 = @B01;
ARRAY 4 POINTER WORD PTR6;
```

5.2.4. Déclaration de texte

1. Syntaxe

<déclaration de texte> ::= STRING <liste déclaration de texte>

<liste déclaration de texte> ::= <élément déclaration de texte>
| <liste déclaration de texte> , <élément déclaration de texte>

<élément déclaration de texte> ::= <identifieur non déclaré>
| <identifieur non déclaré> = <chaîne>

2. Sémantique

La déclaration de texte permet de réserver en zone données une suite de caractères (chaîne) préfixée par la longueur de la chaîne. Cette réservation est utile en particulier pour l'emploi de certains services système (SIRIS8 ou autre) : impression de messages par exemple.

3. Exemple

```
STRING MSG1 = "MESSAGE 1",
       MSG2 = "MESSAGE 2";
STRING MSG3 = "MESSAGE 3";
STRING MSG4;
```

5.2.5. Déclaration de tableau

1. Syntaxe

```
<déclaration de tableau> ::= ARRAY <liste déclaration de tableau>
<liste déclaration de tableau> ::= <élément déclaration de tableau>
                                 | <élément déclaration de tableau> ,
                                   <liste déclaration de tableau>
<élément déclaration de tableau> ::= <nombre entier> <type> <liste de tableau>
                                   | <nombre entier> POINTER <type> <liste
                                                               de tableau>
                                   | <nombre entier> LOGICAL <liste de tableau>
<liste de tableau> ::= <tableau>
                     | <tableau> , <liste de tableau>
<tableau> ::= <identifieur non déclaré>
            | <identifieur non déclaré> = <liste d'initialisation>
            | <identifieur non déclaré> SYN <synonyme>
<liste d'initialisation> ::= <élément d'initialisation>
                           | <liste d'initialisation>
                                 <élément d'initialisation>
<élément d'initialisation> ::= <valeur initiale>
                             | * <nombre entier> ( <valeur initiale> )
<valeur initiale> ::= <initialisation>
                    | <valeur booléenne>
```

2. Sémantique

Le nombre entier spécifié dans une déclaration de tableau indique le nombre d'éléments du tableau.
La taille de la zone de données réservée par une déclaration de tableau est égale à la longueur d'un élément du tableau, multipliée par le nombre d'éléments et arrondie à un nombre entier de mots.
Les éléments d'initialisation d'un tableau doivent être en accord avec le type de ce tableau.

Tout ou partie des éléments d'un tableau peut être initialisée.
Dans <liste d'initialisation> chaque <valeur initiale> se rapporte à un élément de tableau. L'ordre des initialisations suit l'ordre des éléments du tableau, le premier élément initialisé est le premier élément du tableau...
L'initialisation répétitive est obtenue au moyen du facteur de répétitions
* <nombre entier> .

EXCEPTION Dans l'initialisation d'un tableau d'octets par des quantités de type "chaîne" plusieurs éléments consécutifs du tableau peuvent être initialisés par une chaîne unique.

Il n'existe aucune différence entre une déclaration de variable simple avec synonyme et une déclaration de tableau avec synonyme. Dans les deux cas l'identifieur de variable simple ou de tableau est synonyme d'une quantité déjà déclarée (variable simple, variable de tableau, pointeur ou d'un nombre entier.

En LP80 il n'y a des tableaux qu'à une seule dimension.
Un tableau n'est qu'une suite de variables d'un même type accessible au moyen d'un seul nom.

3. Exemples

```
ARRAY 25 WORD W5 = *24(2) -1,
      20 WORD W6 = *20("ABCD");
ARRAY 3 SHORT S4 ;
ARRAY 4 SHORT S5 = "AB" -1  "  " #12;
ARRAY 80 BYTE B03 = "CHAINE DE CARACTERES";
ARRAY 132 BYTE LIGNE = *132(" ");
ARRAY 5 REAL FLOAT = 0.1 0.2 3.14159 .5E-28 1.3E+40 ;
ARRAY 4 LOGICAL BOOLEENS = TRUE *2(FALSE) TRUE;
```

5.2.6. Déclaration de pile

1. Syntaxe

```
<déclaration de pile>  ::=  STACK <nombre entier> <liste pile>
<liste pile>     ::=  <élément pile>
                    | <liste pile> , <élément pile>
<élément pile>   ::=  <identifieur non déclaré>
                    | <identifieur non déclaré> ( <masque1> )
<masque1>        ::=  00 | 01 | 10 | 11
```

2. Sémantique

La déclaration de pile réserve en zone données :

- un double-mot pour le pointeur pile
- un nombre entier de mots (spécifié dans la déclaration) pour la pile elle-même.

L'initialisation du pointeur pile est effectuée de la manière suivante :

- l'adresse du sommet de la pile (positions binaires 8 à 31) est initialisée avec l'adresse du premier mot de la pile.
- le compte des espaces (positions binaires 33 à 47) est initialisé avec le nombre de mots de la pile.
- le compte des mots (positions binaires 49 à 63) est initialisé à zéro.
- les masques de déroutement sur débordement du compte des espaces (TS : position binaire 32) et de déroutement sur débordement du compte des mots (TW position binaire 48) sont positionnés selon les valeurs du masque spécifié par la déclaration :

Masque TS	TW	Signification
0	0	(Aucun déroutement n'est inhibé)
0	1	(Déroutement sur appel mot pile inhibé)
1	0	(Déroutement sur rangement mot pile inhibé)
1	1	(Tous les déroutements sont inhibés)

Le masque par défaut est 00.
La pile ne peut être ni initialisée, ni synonyme d'aucune entité.
Par contre, un identifieur de variable ou de pointeur peut être synonyme d'une pile.

3. Exemple

```
STACK 10 PILE1;
STACK 16 PILE2,PILE3(11);
WORD ADRSOMMET SYN PILE1;
```

5.3. Déclaration de constante

1. Syntaxe

```
<déclaration de constante> ::= CONSTANT <liste déclaration de constante>
<liste déclaration de constante> ::= <élément déclaration de constante>
                                   | <liste déclaration de constante> ,
                                     <élément déclaration de constante>
<élément déclaration de constante> ::= <identifieur non déclaré> =
                                       <expression arithmétique entière>
<expression arithmétique entière> ::= <expression entière>
                                    | <signe> <expression entière>
<expression entière> ::= <nombre entier>
                       | <nombre entier> <opérateur arithmétique>
                         <expression entière>
<opérateur arithmétique> ::= <signe> | * | /
```

2. Sémantique

La déclaration de constante permet d'associer un nombre entier à un identifieur.
L'identifieur de constante une fois déclaré peut apparaître partout où un nombre entier est autorisé.
La déclaration de constante ne provoque aucune réservation d'espace mémoire.

Il existe trois "constantes" prédéclarées. Elles contiennent des valeurs courantes de compteurs internes au compilateur.

SYSPTRPROG : contient la valeur courante du compteur d'emplacement

SYSPTRDATA : contient la valeur courante du compteur d'emplacement dans la section de données courante.

SYSPTRCARD : contient le numéro courant de la ligne source.

Ces trois valeurs permettent au programmeur une plus grande souplesse d'utilisation de LP80.
Par exemple la valeur SYSPTRCARD peut être fournie en paramètre à une routine de traitement d'erreur afin d'indiquer la ligne source dans le programme où l'appel est effectué. Ou encore SYSPTR DATA indique en fin de description d'un bloc de données la longueur en mots de ce bloc qui peut dès lors être utilisée afin d'acquérir dynamiquement de la mémoire.

3. Exemple

```
CONSTANT LNG_ELEMENT = 10;
CONSTANT NB_ELEMENTS = 5;
CONSTANT NB_TOTAL = LNG_ELEMENT * NB_ELEMENTS,
         CST1 = 10;
CONSTANT CST2 = #3F,CST3 = SYSPTRDATA;
```

5.4. Déclaration de fonction assembleur

1. Syntaxe

<déclaration de fonction assembleur> ::= INSTRUCTION <liste déclaration de fonction>

<liste déclaration de fonction> ::= <élément déclaration de fonction>
| <liste déclaration de fonction> , <élément déclaration de fonction>

<élément déclaration de fonction> ::= <identifieur non déclaré><nombre entier>

2. Sémantique

La valeur donnée par le nombre entier est la valeur d'un code opération admis par l'IRIS 80. Cette valeur doit être comprise entre 0 et 127. Toutes les instructions de l'IRIS 80 peuvent aussi être employées après avoir été déclarées. Un certain nombre d'instructions sont prédéclarées. (Cf. annexe 2).

3. Exemple

```
INSTRUCTION CHARGE_INDICATEURS #02;
INSTRUCTION ANALYSER #44,CHARGEMENT_BASE_IMMEDIATE #01;
```

5.5. Déclaration de condition

1. Syntaxe

```
<déclaration de condition> ::=  CONDITION <liste de condition>
<liste de condition>  ::=  <élément de condition>
                         | <liste de condition> , <élément de condition>
<élément de condition>  ::= <identifieur non déclaré> <masque2>
<masque2>  ::= <nombre entier>
             | ¬ <nombre entier>
```

2. Sémantique

La valeur donnée par le nombre entier est la valeur d'un masque permettant de tester si la condition est satisfaite. Ce nombre doit donc être compris entre 0 et 15. Si le signe de négation est présent (¬), la condition sera satisfaite si aucun des bits correspondants au masque dans le code condition est positionné.

3. Exemple

```
CONDITION EGAL #3,NON_EGAL ¬ #3;
CONDITION TEST_TTBS #1;
```

5.6. Déclaration d'étiquette

1. Syntaxe

<déclaration d'étiquette> ::= LABEL <liste déclaration d'étiquette>
ENTRY <liste déclaration d'étiquette>

<liste déclaration d'étiquette> ::= <identifieur non déclaré>
| <liste déclaration d'étiquette> ,
<identifieur non déclaré>

2. Sémantique

Dans un programme LP80 toute utilisation d'une étiquette doit être précédée d'une déclaration de cette étiquette. Les étiquettes peuvent être de 2 types :

- LABEL : elles ne sont alors connues que du segment courant et peuvent apparaître dans n'importe quel niveau de bloc.

- ENTRY : elles sont connues du segment courant, mais aussi des segments externes. Elles ne peuvent alors apparaître que dans le bloc de niveau 1.

La déclaration des étiquettes ne réserve aucun emplacement mémoire.

3. Exemple

```
LABEL ETIQ1;
LABEL ETIQ2,ETIQ3;
ENTRY ENTRY1,ENTRY2;
```

5.7. Déclaration de procédure

1. Syntaxe

<déclaration de procédure> ::= PROCEDURE <définition de procédure> ; <instruction>

<définition de procédure> ::= <identifieur non déclaré>
| <identifieur non déclaré> <identifieur de registre>

La syntaxe <instruction> est donnée au chapitre 6.

2. Sémantique

L'identifieur de registre spécifie le registre de retour de la procédure.
Par défaut le registre RF est utilisé. La déclaration de procédure donne lieu à la génération de code :

- un branchement avant la procédure permettant de sauter le corps de procédure
- un branchement de retour à la fin de la procédure.

3. Exemple

```
PROCEDURE VIDE;
NULL;

PROCEDURE ERREUR RE;
BEGIN &
        .
        .
        .
        .
        &
END;
```

6. INSTRUCTIONS

6.1. Blocs

1. Syntaxe

```
<bloc>   ::=   <début de bloc>  <liste d'élément de bloc>   END
<début de bloc>  ::=   BEGIN
                     |<nombre entier> ) BEGIN
<liste d'élément de bloc>  ::=  <élément de bloc>
                              |<liste d'élément de bloc>  <élément de bloc>
<élément de bloc>  ::=  <déclaration>
                      | <instruction>
                      | <liste étiquette>  :  <instruction>
<liste étiquette>   ::=  <identifieur d'étiquette>
                       | <liste étiquette>  :  <identifieur d'étiquette>
```

2. Sémantique

Un bloc en LP80 suit les règles classiques, communes à tous les langages possédant cette notion.
Les blocs sont numérotés (implicitement, et le numéro apparaît sur la liste dans la marge) selon leur degré d'imbrication, le premier bloc étant de niveau 1.
Il existe un bloc implicite, ouvert au début de la compilation. Il contient les déclarations de registres généraux et de bases (implicites) et les déclarations explicites de segments externes et de blocs de données. Il est désigné dans ce document par le terme : bloc de niveau 0.

<Nombre entier>est un numéro qui peut être associé à un bloc afin d'être employé par les instructions CYCLE et EXIT.
Un bloc est un ensemble de déclarations et d'instructions comprises entre un début de bloc et une fin de bloc.
Une déclaration faite dans un bloc n'est connue que dans ce bloc. L'identifieur ainsi déclaré est dit local au bloc. Les identifieurs connus dans le bloc mais déclarés dans un bloc englobant sont dits globaux.
On peut redéfinir un identifieur dans un bloc englobé.

3. Exemples

```
BEGIN
     WORD A,B;
     A := R0;
     BEGIN
          ETIQ1 :
          WORD A;
          A := R0;
          B := R0;
          1)BEGIN
               WORD A,B;
               A := R0;
               B := R0;
          END;
     END;
     BEGIN
          WORD B;
          A := R0;
          B := R0;
     END;
     B := R0;
END;
```

6.2. Instructions d'affectation

Les instructions d'affectation sont en rapport avec la structure de la machine. Elles peuvent donc se faire :

- de registre à registre
- de base à base
- de registre à mémoire
- de mémoire à registre
- de mémoire à base
- de mémoire à mémoire pour les opérations sur les LOGIQUES.
- de registre à base
- de base à registre

Afin de ne pas alourdir les descriptions des différentes instructions la description des unités syntaxiques particulières :

<variable> <variable indirecte> <registre> <base> <logique>

est donnée préalablement.

6.2.1. Désignation d'une variable

1. Syntaxe

<variable> ::= <identifieur de variable>
| <identifieur de variable> <complément d'adressage>

<complément d'adressage> ::= <spécification base>
| <spécification base> <déplacement-indexation>
| <déplacement-indexation>

<spécification base> ::= . <base>

<déplacement-indexation> := <déplacement complémentaire
| <déplacement complémentaire <index>
| <index> <déplacement complémentaire
| <index>

<index> ::= (<registre>)

La syntaxe de <logique> est la même que celle de <variable> en remplaçant <identifieur de variable> par <identifieur de logique> .

2. Sémantique

Le déplacement est toujours considéré comme un nombre entier de mot quel que soit le type de la variable.
La base doit être comprise entre 1 et 7. Si la base B0 est spécifiée, il s'agira alors d'un adressage absolu d'une variable dont l'adresse sera donnée par le déplacement. L'usage explicite d'une base permet de changer la base associée de manière standard à la variable.
Le registre index spécifié doit être compris entre R1 et R7.

3. Exemple

```
ARRAY 20 BYTE CHAINE;
CHAINE               & DESIGNE LE PREMIER OCTET &
CHAINE(2)            & DESIGNE LE 9 EME OCTET   &
CHAINE(R1:=2)        & DESIGNE LE 3 EME OCTET   &
CHAINE(2)(R1:=2)     & DESIGNE LE 11 EME OCTET  &
CHAINE(3)(R1:=-1)    & DESIGNE LE 11 EME OCTET  &
CHAINE(-1)(R1:=4)    & DESIGNE LE PREMIER OCTET &
```

6.2.2. Désignation d'une variable indirecte

1. Syntaxe

```
<variable indirecte>  ::=  $ <identifieur de pointeur>
                          | $ <identifieur de registre>
                          | $ <identifieur de pointeur> <complément d'adressage>
                          | $ (<registre>)
                          | $ (<registre>)<index>
```

2. Sémantique

Ce qui a été dit pour désignation de variable reste vrai. Il faut noter deux points importants :

- Lorsqu'un déplacement supplémentaire est indiqué, il concerne la désignation du pointeur et non la variable pointée.
- Lorsqu'un index est précisé, il s'applique sur la variable pointée. L'indexation est une POST-INDEXATION.

Il faut éviter l'emploi de l'indirection.
Ce mécanisme est coûteux et peut être avantageusement remplacé par l'emploi d'une base.
Par exemple pour désigner une variable de type SHORT dont l'adresse (mot) est contenue dans un registre R, il est préférable d'écrire :

```
BASE := R;
SHORT0.BASE :=
```

plutôt que :

```
POINTER SHORT PTR;
PTR := R;
$PTR :=
```

3. Exemples

```
$PTR1               & DESIGNE LE MOT DONT L'ADRESSE EST DANS PTR1 &
$PTR2(R1:=2)        & DESIGNE LE MOT DONT L'ADRESSE EST EGALE AU
                      CONTENU DE PTR2 PLUS 2                      &
$PTR6(2)            & DESIGNE LE MOT DONT L'ADRESSE EST CONTENUE
                      DANS LE TROISIEME ELEMENT DU TABLEAU PTR6   &
$RA                 & DESIGNE LE MOT DONT L'ADRESSE EST CONTENUE
                      DANS LE REGISTRE RA                         &
```

6.2.3. Désignation des registres

1. Syntaxe

<registre> ::= <identifieur de registre> | <assignation simple de registre>

La syntaxe de <assignation simple de registre> est donnée en 6.2.5.

2. Sémantique

L'identifieur de registre permet de lui associer un numéro de registre.
Ce numéro doit être valide, c'est-à-dire correspondre à l'emploi que l'on veut en faire.
En particulier un registre d'index doit être compris entre 1 et 7.

6.2.4. Désignation des bases

1. Syntaxe

<base> ::= <identifieur de base>

2. Sémantique

L'emploi de la base 0 n'est valable que pour les opérations sur les chaînes.
La base 0 ne pouvant servir à l'adressage, son emploi dans ce sens permet de forcer un adressage absolu.

6.2.5. Assignation simple de registre

1. Syntaxe

```
<assignation simple de registre> ::= <identifieur de registre>:=<expression>
<expression> ::= <expression simple>
               | <expression simple> <expression multiple>
<expression simple> ::= <opérande>
                      | <opérateur unaire> <opérande>
                      | @ <identifieur d'adresse>
                      | % <identifieur déplacement>
<opérande> ::= | <base>
               | <identifieur de registre>
               | <variable>
               | <variable indirecte>
               | <nombre>
               | <chaîne>
<opérateur unaire> ::= ABS | -
<expression multiple> ::= <élément d'expression>
                        | <élément d'expression> <expression multiple>
<élément d'expression> ::= <opérateur décalage> <opérande décalage>
                         | + % <identifieur déplacement>
                         | <opérateur arithmétique ou logique> <opérande>
<opérateur arithmétique ou logique> ::= <opérateur arithmétique>
                                      | <opérateur logique>
<opérateur logique> ::= AND |̳ XOR
```

REMARQUE On écrit |̳ pour le distinguer du connecteur métalinguistique.

```
<opérateur décalage> ::= SLAD|SLAS|SLCD|SLCS|SLFD|SLFS|SLLD|SLLS|SRAD|
                         SRAS|SRCD|SRCS|SRFD|SRFS|SRLD|SRLS


<opérande décalage> ::= <nombre entier>
                      | <nombre entier>( <registre> )
                      | <assignation simple de registre>
```

2. Sémantique

L'instruction d'assignation simple de registre réalise une suite d'opérations entre le registre et différentes données, avec résultat dans le registre assigné. Les expressions sont évaluées de la gauche vers la droite sans priorité d'opérateur. Lorsqu'un registre index est assigné, le code de l'assignation de l'index est engendré avant le code de l'instruction utilisant cet index.

3. Code engendré sans opérateur de décalage et sans le caractère @ ou %

Le tableau ci-dessous donne les mnémoniques des instructions engendrées lors d'une assignation simple de registre sans opérateur de décalage ou .

Instructions	Sans opér.	- Unaire	ABS	+	-	*	/	AND	/	XOR
Immédiate	LI			AI		MI				
BYTE	LB									
SHORT	LH	LCH	LAH	AH	SH	MH	DH			
WORD	LW	LCW	LAW	AW	SW	MW	DW	AND	OR	EOR
LONG	LD	LCD	LAD	AD	SD	MD				
REAL	LW	LCW	LAW	FAS	FSS	FMS	FDS			
LONGREAL	LD	LCD	LAD	FAL	FSL	FML	FDL			

- Code engendré avec @ (adresse de)

Le code engendré est une instruction LVAW de l'adresse de l'opérande adresse ou un LW d'un mot contenant l'adresse de l'opérande externe. Lorsque l'opérande adresse est de type mot, l'index éventuel est généré dans l'instruction LVAW. L'adresse obtenue est transformée suivant le type de la variable au moyen d'une instruction SHIFT. L'index éventuel est ensuite ajouté au moyen d'une instruction AW.

- Code engendré avec % (déplacement de)

Le code engendré est une instruction de chargement immédiat du déplacement de l'"opérande adresse" dans le registre assigné ou une addition (AI). Lorsque l'"opérande adresse" est indexé, une addition mot (AW) de l'index dans le registre assigné est ensuite engendrée.

- Code engendré avec opérateur de décalage

Le code engendré est une instruction de décalage (S ou SF).
Le type de décalage dépend de l'opérateur de décalage.
La signification des opérateurs est la suivante :

- 2ème caractère	L	décalage à gauche (left)
	R	décalage à droite (right)
- 3ème caractère	A	décalage arithmétique
	F	décalage flottant
	L	décalage logique
	C	décalage circulaire
- 4ème caractère	S	un seul registre
	D	deux registres (couple pair-impair)

Lorsque l'opérande de décalage est un registre, le sens du déplacement n'est plus indiqué par l'opérateur mais par le contenu du registre :
si $(R) < 0$ le décalage s'effectue à droite, si $(R) > 0$ le décalage s'effectue à gauche.

Pour chaque instruction engendrée :
- l'opérateur définit la classe d'instruction à engendrer.

- le premier opérande est le registre assigné (registre à gauche du signe :=). Le type de ce registre n'est pas pris en compte. Le numéro du registre est introduit dans le champ registre de l'instruction.
  Exemple :

  Les identifieurs de registres R2, R2_3, F2_3 spécifient l'adresse registre 2.

- Le deuxième opérande (à droite de l'opérateur) définit le type de l'instruction à engendrer. Lorsque ce deuxième opérande est une variable indirecte, le bit d'adressage indirect est positionné à 1 dans l'instruction engendrée.

Les opérandes nombre entier décimal ou hexadécimal, nombre flottant, chaîne donnent lieu à une réservation de données (mot ou double-mot) lorsqu'ils ne sont pas encore apparus dans les blocs dominant et courant. Toutefois, dans le cas d'instruction de chargement, addition, multiplication, si les nombres entiers ou chaînes sont inférieurs en valeur absolue $2^{19}-1$, les instructions immédiates LI, AI, MI sont engendrées et aucune réservation de données n'est effectuée.

4. Exemples

```
R0 := 4;
R0 := 4 + R1;
R0 := @L1;
R0 := R0 +W1;
R0_1 := - #3E38;
R0 := ABS R0;
R0 := R0 AND R1 XOR #F0F0F0F0;
R0 := B5 + CST2;
F0 := 0.1E-12;
R0 := "ABCD";
R0 := R1 ! R2;
R0 := R0 SRLS 8;
R0 := R0 SRLS R1;
R0 := R0 SLLS 2 (R1 := 5);
```

6.2.6. Assignation simple de variable

1. Syntaxe

```
<assignation simple de variable>  ::=  <variable>  :=  <registre>
                                     |  <variable indirecte>  :=  <registre>
```

2. Sémantique

Cette instruction range le contenu d'un registre dans une cellule de mémoire.

3. Code engendré

Selon le type de la variable, le compilateur engendre l'une des quatre instructions "STORE", directe ou indirecte, du registre dans la variable (instructions STB, STH, STW, STD).

4. Exemples

```
W2 := R0 := W3;
$PTR1 := R0;
```

6.2.7. Assignation simple de base

1. Syntaxe

<assignation simple de base> ::= <identifieur de base> := <registre>
| <identifieur de base> := <opérande>

2. Sémantique

Le registre de base indiqué est chargé avec la valeur indiquée.

3. Code généré

Le code généré est un LBR (Load Base Register)

4. Exemple

```
B0 := 64;
BASE := R0 := W2;
```

6.2.8. Assignation multiple

1. Syntaxe

<assignation multiple> ::= <liste registre-variable> := <registre>
<liste registre-variable> ::= <élément registre-variable>
| <liste registre-variable> := <élément registre-variable>
<élément registre-variable> ::= <variable>
| <variable indirecte>
| <identifieur de registre>
| <identifieur de base>

2. Sémantique

L'assignation multiple permet de ranger dans des registres et cellules de mémoire, la valeur d'un registre.

3. Code engendré

L'assignation multiple est composée d'une assignation simple de registre suivie de n assignations simples de variables.
Le code engendré est celui de l'assignation simple de registre, les assignations de variable étant engendrées ensuite de la droite vers la gauche.

4. Exemples

```
$PTR1 := W1 := R3 := R0;
W2 := W3 := PTR1 := R0 := @W1;
```

6.2.9. Echange-mot

1. Syntaxe

```
<échange-mot> ::= <élément registre-variable> :=: <identifieur de registre>
                | <identifieur de registre> :=: <élément registre-variable>
```

2. Sémantique

Cette instruction permet d'échanger les contenus d'un registre et d'une cellule de mémoire.

3. Code engendré

Le code engendré par le compilateur est une instruction échange mot XW.

4. Exemples

```
R0 :=: W1;
R0 :=: $PTR1;
R0 :=: R1;
$PTR1 :=: R0;
W1 :=: R0;
B0 :=: R0;
R0 :=: B0;
```

6.2.10. Assignation simple de logique

1. Syntaxe

```
<Assignation simple de logique> ::= <logique> := <expression booléenne>
<Expression booléenne> ::=  <valeur booléenne>
                            <expression conditionnelle>
<expression conditionnelle>  ::=  <condition combinée>
                                | <condition alternative>
<condition combinée>  ::=  <condition>
                        |  <condition>  AND  <condition combinée>
<condition alternative>  ::=  <condition>
                           |  <condition> ! <condition alternative>
<condition>           ::= <relation>
                        | <valeur logique>
                        | <condition simple>
<relation>            ::= <opérateur de relation>
                        |  ¬  <opérateur de relation>
<opérateur de relation>::=  =  | <  | >  | > =  | < =
                        | <identifieur de condition>
<valeur logique>      ::=  <logique>
                        | ¬  <logique>
<condition simple>    ::=  <variable> <relation>  0
                        |  <variable indirecte> <relation>  0
                        |  <base>  <relation>  0
                        |  <identifieur de registre> <relation><terme>
                        |  ( <registre> ) <relation> <terme>
<terme>               ::=  <chaîne>
                        | + <nombre>
                        | - <nombre>
                        |   <nombre>
                        |  ( <registre> )
                        |   <base>
                        |   <identifieur de registre>
                        |   <variable>
                        |   <variable indirecte>
```

2. <u>Sémantique</u>

Cette instruction permet d'affecter une valeur booléenne à un logique. Si l'expression conditionnelle est vérifiée, le logique prend la valeur VRAI. Sinon il prend la valeur FAUX.

On dit qu'une expression conditionnelle est vérifiée ou non.

- Une expression conditionnelle qui est une relation entre un registre et un terme est vérifiée si et seulement si la relation entre la valeur courante du registre et le terme est vraie.

- Une expression conditionnelle qui est une relation entre une variable ou une base et la valeur 0 est vérifiée si et seulement si la relation entre la valeur de la variable ou de la base et la valeur 0 est vraie.

- Une expression conditionnelle qui est un logique est vérifiée si et seulement si le logique spécifié a la valeur TRUE.
  (Condition inverse si ¬ est spécifié).

- Une expression conditionnelle qui est une relation est vérifiée si et seulement si le code condition contient la valeur exprimée par l'opérateur de relation.
  (Condition inverse si ¬ est spécifié).

Opérateur de relation prédéclaré	Signification	Valeurs du code-condition si condition vraie (*)
=	égal	-- 00
<	plus petit que	-- 01
>	plus grand que	-- 10
<=	plus petit ou égal à	-- 0-
>=	plus grand ou égal à	-- -0

(*) Le tiret dans cette colonne indique un chiffre binaire du code condition non testé (valeur quelconque).

3. Code généré

L'assignation simple de logique n'utilise pas de registre de travail. Le code condition est employé comme registre travail. Il contient donc la valeur VRAI ou FAUX à la fin de l'assignation simple de logique. Le logique est chargé au moyen d'une instruction STCF.

4. Exemples

```
BOOL1 := TRUE;
BOOL2 := R1 > W2;
BOOL3 := NON_EGAL;
BOOLEENS(R1 := 2) := ^ BOOL3;
```

6.2.11. Assignation multiple de logique

1. Syntaxe

<Assignation multiple de logique> ::= <liste de logique> := <assignation simple de logique>

<Liste de logique> ::= <logique>
| <liste de logique> := <logique>

2. Sémantique

Cette instruction permet d'affecter une valeur booléenne à plusieurs identifieurs de logique.

3. Code généré

Les différents logiques sont changés au moyen d'une suite d'instructions STCF.

4. Exemple

```
BOOL1 := BOOL2 := FALSE;
```

6.3. INSTRUCTIONS DE CONTROLE

Les instructions de contrôle permettent d'effectuer des actions différentes en fonction de la valeur de certaines variables.

6.3.1. Instruction IF

1. Syntaxe

<instruction IF> ::= <proposition IF> <instruction>
<proposition IF> <instruction> ELSE <instruction>

<proposition IF> ::= IF <expression conditionnelle> THEN

2. Sémantique

La première instruction n'est exécutée que si l'expression conditionnelle est vérifiée.
Lorsqu'elle existe, la deuxième instruction n'est exécutée que si l'expression conditionnelle n'est pas vérifiée.

3. Code généré

Première forme : IF condition THEN instruction

```
        :
        Branchement à L1, si condition fausse
        instruction
  L1 :  :
```

Deuxième forme : IF condition THEN instruction1 ELSE instruction2

```
        :
        Branchement à L1 sur condition fausse
        instruction1
        Branchement inconditionnel à L2
  L1 :  instruction2
  L2 :  :
```

6.3.2. Instruction CASE

1. Syntaxe

<instruction CASE> ::= <proposition CASE> / <nombre entier> OF <liste CASE>

<proposition CASE> ::= CASE <identifieur de registre>
| CASE (<assignation simple de registre>)

<liste CASE> ::= <instruction>
| <liste CASE> OR <instruction>

2. Sémantique

L'instruction CASE permet la sélection d'une instruction parmi une suite d'instruction selon la valeur du registre spécifié. Ce registre doit être un registre d'index.
L'instruction dont le numéro en séquence correspond à la valeur du registre est exécutée, les autres étant ignorées. La valeur du registre doit être comprise entre 0 et le nombre entier spécifié, sinon une erreur se produit.
Le nombre d'instruction doit être égal au nombre entier spécifié.
Si la valeur du registre est 0, aucune instruction n'est exécutée.

3. Code engendré

Le compilateur génère un tableau de <nombre entier> éléments. Chaque élément est une instruction de branchement inconditionnel à la séquence d'instruction correspondante.
Une instruction de branchement à l'intérieur de ce tableau, indexée par le registre spécifié permet de déterminer la séquence qu'il faudra exécuter.
Chaque séquence se termine par un branchement en fin de CASE.

Illustration

```
          CASE   R1/3  OF
          instruction1
          OR
          instruction2
          OR
          instruction3
entraîne  la génération de :
          branchement en L(R1)
 L :      branchement en LOUT
          branchement en L1
          branchement en L2
          branchement en L3
 L1 :     instruction1
          branchement en LOUT
 L2 :     instruction2
          branchement en LOUT
 L3 :     instruction3
 LOUT :   .
          .
          .
          .
          .
```

4. Exemple

```
CASE R1 / 3 OF
BEGIN & SI R1 = 1 &

END
OR
BEGIN & SI R1 = 2 &

END
OR
BEGIN & SI R1 = 3 &

END;
```

6.3.3. Instruction WHILE

1. Syntaxe

<instruction WHILE> ::= <proposition WHILE> <instruction>

<proposition WHILE> ::= WHILE <expression conditionnelle> DO

2. Sémantique

L'instruction WHILE permet d'exécuter l'instruction suivant la proposition tant que l'expression conditionnelle est vérifiée.

3. Code généré

```
      .
      .
      .
L1 : branchement en L2 si condition fausse
     instruction
     branchement en L1
L2 : .
      .
      .
```

4. Exemples

```
WHILE BOOL1 DO
BEGIN &
      INSTRUCTION
      &
END;

WHILE R1 > 0 DO NULL;
```

6.3.4. Instruction FOR

1. Syntaxe

<instruction FOR>	::=	<proposition FOR> <instruction>
<proposition FOR>	::=	<en-tête FOR> UNTIL <limite> DO
<en-tête FOR>	::=	FOR <registre>
	|	FOR <registre> STEP <pas>
<pas>	::=	<nombre entier>
	|-	<nombre entier>
<limite>	::=	<registre>
	|	<variable>
	|	<variable indirecte>
	|	<nombre entier>
	|-	<nombre entier>

2. Sémantique

L'instruction FOR permet d'exécuter une instruction un certain nombre de fois sous le contrôle d'un registre.
Ce registre est modifié à chaque étape d'une valeur entière, le <pas> , (par défaut +1) jusqu'à ce qu'une limite soit atteinte.

3. Code engendré

- code de l'assignation simple de registre (si nécessaire)
- instruction de comparaison du registre avec la limite
- branchement après l'instruction FOR si condition vérifiée
- code de l'instruction
- modification du registre de contrôle
- branchement inconditionnel à la comparaison

4. Exemple

```
FOR R1:= 1 UNTIL 8 DO NULL;
FOR R1 := 100 STEP - 1 UNTIL 0 DO NULL;
FOR R1 UNTIL R2 DO NULL;
FOR R1 UNTIL -2 DO NULL;
FOR R1 := 0 UNTIL W1 DO NULL;
```

6.3.5. Instruction REPEAT

1. Syntaxe

<instruction REPEAT> ::= <proposition REPEAT><instruction>

<proposition REPEAT> ::= REPEAT <registre> TIMES

2. Sémantique

L'instruction REPEAT permet d'exécuter une instruction un nombre déterminé de fois sous le contrôle d'un compteur.
La valeur initiale du compteur doit être strictement positive.

3. Code engendré

L'instruction REPEAT traduit l'utilisation de l'instruction BDR :
- code de l'assignation simple de registre (si nécessaire)
- code de l'instruction
- instruction BDR

L'instruction BDR décrémente le registre avec branchement au code de l'instruction si la nouvelle valeur du registre est positive.
Le test étant effectué en fin de boucle, l'instruction REPEAT est exécutée au moins une fois.

4. Exemple

```
REPEAT R1 TIMES NULL;
REPEAT R1 := 5 TIMES NULL;
REPEAT R1 := W1 TIMES NULL;
```

6.4. Instructions de branchement

Ces instructions permettent d'effectuer un branchement inconditionnel.

6.4.1. Instruction LOOP

1. Syntaxe

<instruction LOOP> ::= LOOP <instruction>

2. Sémantique

L'instruction LOOP réalise une boucle infinie sur une instruction.

3. Code engendré

- code de l'instruction
- branchement inconditionnel au code de l'instruction

4. Exemple

```
LOOP NULL; & ON EST PAS PRES DE S'ARRETER &
LOOP
BEGIN &
      INSTRUCTION
      &
END;
```

6.4.2. Instruction GOTO

1. Syntaxe

<instruction GOTO> ::= GOTO <désignation d'étiquette>
| GOTO <variable>
| GOTO <variable indirecte>

<désignation d'étiquette> ::= <identifieur d'étiquette>
| <identifieur d'étiquette> <déplacement-indexation>

2. Sémantique

Cette instruction permet un branchement inconditionnel en un emplacement de mémoire quelconque.
La structure et les possibilités du langage LP80 ne nécessitent pas l'existence de l'instruction GOTO.
Celle-ci a cependant été conservée dans le but de faciliter le transport de programmes mal écrits en LP70. Il est évident que l'emploi d'un GOTO autrement que dans la forme branchement à une étiquette -et encore!- ne peut être le fait que de programmeurs débutants n'ayant pas le souci d'une programmation claire et structurée.

3. Code généré

Le code généré est un branchement inconditionnel éventuellement indexé à l'adresse indiquée.

4. Exemple

```
GOTO $RF;
GOTO $PTR1;
GOTO ETIQ1;
GOTO ETIQ2(2);
GOTO ETIQ1(R1);              & IL VAUDRAIT MIEUX &
GOTO  ETIQ1(1)(R1);          & FAIRE UN CASE OF  &
```

6.4.3. Instruction CYCLE

1. Syntaxe

```
<instruction CYCLE> ::=  CYCLE <nombre entier>
                       | CYCLE
```

2. Sémantique

Cette instruction permet un branchement inconditionnel à l'adresse de la première instruction du bloc courant ou du bloc spécifié. Le nombre entier permet de désigner un bloc étiqueté dans lequel se trouve l'instruction CYCLE.

3. Code engendré

Le code engendré est un branchement inconditionnel à l'adresse de début du bloc désigné.

6.4.4. Instruction EXIT

1. Syntaxe

<instruction EXIT> ::= EXIT
| EXIT <nombre entier>

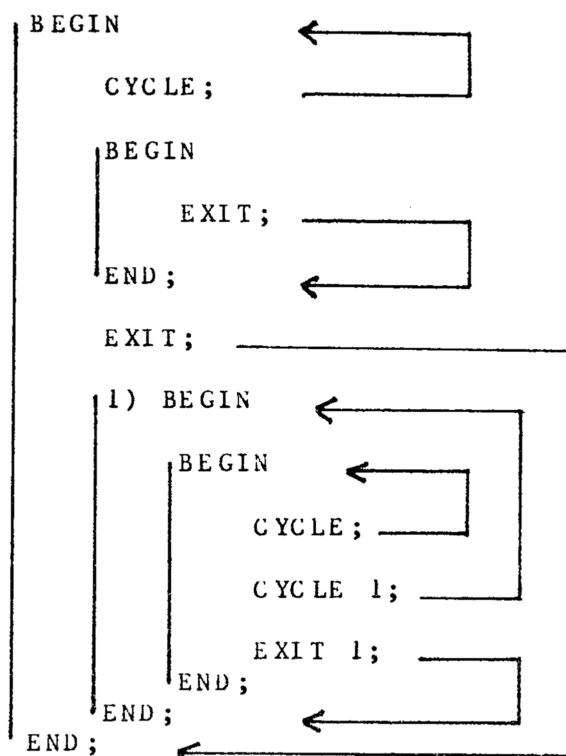
2. Sémantique

Cette instruction permet de sortir d'un bloc.
Le nombre entier s'il est précisé désigne l'étiquette d'un bloc dans lequel se trouve l'instruction EXIT.

3. Code généré

Le code engendré est un branchement inconditionnel à l'adresse de l'instruction qui suit la dernière instruction du bloc.

4. Exemple



6.5. Instruction NULL

1. Syntaxe

<instruction NULL> ::= NULL

2. Sémantique

Cette instruction est l'instruction vide . Elle peut servir à préparer sur un listing des emplacements pour des instructions à venir.

3. Code généré

Aucun.

4. Exemple

```
NULL;
```

6.6. Instruction STOP

1. Syntaxe

<instruction STOP> ::= STOP

2. Sémantique

Cette instruction provoque un arrêt du programme et un retour au moniteur. Cette instruction ne doit pas être employée sous SIRIS8 dans une séquence d'exception.

3. Code généré

Pour la version LP80-SIRIS8 le code généré est un appel moniteur correspondant à la fonction M:RETURN.

4. Exemple

```
STOP;
IF BOOL1 THEN STOP;
```

7. APPEL DE FONCTIONS

L'appel de fonction permet de générer une ou plusieurs instructions machines dont l'emploi n'est pas directement possible à partir du langage LP80.

7.1. Fonctions assembleur

1. Syntaxe

<fonction assembleur> ::= <identifieur d'instruction> (<paramètre1> , <paramètre2>)

<paramètre1> ::= <identifieur de registre>
| <identifieur de base>
| <nombre entier>
| -<nombre entier>

<paramètre2> ::= <nombre entier>
| -<nombre entier>
| <variable>
| <variable indirecte>
| <identifieur de procédure>
| <identifieur externe>
| <identifieur de pile>
| <identifieur d'étiquette

2. Sémantique

Les fonctions assembleur permettent de générer des instructions qui ne peuvent pas être directement générées par Compilateur LP80.

3. Code généré

L'instruction est engendrée avec :

- comme code opération (emplacements binaires 1 à 7) la valeur associée à l'identifieur d'instruction lors de sa déclaration.,

- comme registre opération (emplacements binaires 8 à 11)) le registre ou la valeur précisée par paramètre 1,
- comme index, base, et déplacement (emplacements binaires 12 à 31) les valeurs implicites ou explicites du paramètre 2.

  Si le paramètre 2 est un nombre entier, il sera considéré comme champ immédiat de l'instruction et généré comme tel.

  N.B. Le type de la variable n'est pas pris en compte pour la génération des instructions. Toutes les adresses précisées sont donc des adresses mots.

  Les instructions sur chaîne d'octets ne peuvent avoir en paramètre 2 qu'une valeur immédiate.

4. Exemple

```
ANALYSER(R0,W1);
CHARGE_INDICATEURS(2,#40);
CHARGEMENT_BASE_IMMEDIATE(B3,-SYSPTRPROG);
```

7.2. Fonctions composées

7.2.1. Fonctions MOVE, COMP, TRANS

1. Syntaxe

<fonction chaîne> ::= <nom de fonction chaîne> (<identifieur de registre> , <expression1> , <expression2> , <expression3>)

<nom de fonction chaîne> ::= MOVE | COMP | TRANS

2. Sémantique

Ces fonctions ont pour but de simplifier l'emploi des instructions de manipulation de chaînes de caractères (MBS, CBS, TBS).

L'identifieur de registre désigne un registre pair différent de R0. Le couple de registres pair-impair désigné est modifié par la fonction ainsi que le registre de base B0.

<Expression 1> donne la longueur de la chaîne
<Expression 2> donne l'adresse octet du récepteur
<Expression 3> donne l'adresse octet de l'emetteur.

3. Code généré

Si l'on désigne par R_i le registre pair et par R_{i+1} le registre impair associé :

- R_i := <expression 1>
- R_{i+1} :=-<expression 2>
- BO := R_i;
- R_i := <expression 3>
- MBS(R_i,0) ou CBS(R_i,0) ou TBS(R_1,0)

NB : <expression2> et <expression3> ne doivent pas être fonction de R_i ou R_{i+1} sous peine d'erreurs.

4. Exemple

```
MOVE(R2,8,@CHAINE,@MSG1);
COMP(R2,MSG1,@MSG1+1,@MSG2);
```

7.2.2. Fonctions PUSH, PULL

1. Syntaxe

<fonction pile> ::= <nom de fonction pile> (<identifieur de pile> , <fin de fonction pile>)
<fin de fonction pile> ::= <identifieur de registre>
<identifieur de registre> , <identifieur de registre>
<nom de fonction pile> ::= PUSH | PULL

2. Sémantique

Les fonctions de pile permettent de faciliter l'emploi des piles. Si un seul registre est spécifié celui-ci est empilé ou dépilé dans la pile indiquée. Si 2 identifieurs sont spécifiés, tous les registres compris entre les deux valeurs seront empilés ou dépilés dans la pile indiquée.

3. Code généré

S'il n'y a qu'un seul registre (ou si les deux registres spécifiés sont identiques) il y a génération d'une instruction PSW(PUSH) ou PLW(PULL). Si 2 registres différents sont spécifiés, le code condition nécessaire est chargé au moyen d'un LCFI puis il y a génération d'une instruction PSM (PUSH) ou PLM (PULL).

4. Exemples

```
PUSH(PILE1,R0);
PULL(PILE2,R0,RF);
```

7.2.3. Fonctions STORE, LOAD

1. Syntaxe

<fonction sauvegarde> ::= <nom de fonction sauvegarde> (<corps de fonction>)

<corps de fonction> ::= <désignation mémoire> , <fin de fonction sauvegarde>

<désignation mémoire> ::= <variable>
| <variable indirecte>

<fin de fonction sauvegarde> ::= <identifieur de registre>
| <identifieur de registre> , <identifieur de registre>

<nom de fonction sauvegarde> ::= STORE | LOAD

2. Sémantique

Les fonctions de sauvegarde permettent de faciliter l'emploi des instructions de rangement et de chargement.
Si un seul registre est spécifié, celui-ci est rangé ou chargé dans la zone mémoire indiquée.
Si deux registres sont spécifiées, tous les registres compris entre ces deux valeurs seront rangés ou chargés séquentiellement dans la zone mémoire indiquée.

3. Code généré

S'il n'y a qu'un seul registre (ou si les deux registres spécifiés sont identiques) il y a génération d'une instruction STW (STORE) ou LW (LOAD). Si deux registres différents sont spécifiés, il y a génération d'un LCFI permettant le chargement du code-condition à la valeur nécessaire. Il y a ensuite génération d'un STM (STORE) ou d'un LM (LOAD).

4. Exemples

```
STORE(W1,R0);        & IDENTIQUE A W1 := R0 &
LOAD(W5,R0,R7);
```

7.2.4. Fonction RETURN

1. Syntaxe

```
<fonction return>  ::=  RETURN
                        RETURN ( <liste de paramètre> )
<liste de paramètre>  ::= <expression>
                        | , <liste de paramètre>
                        | <liste de paramètre>   ,   <expression>
```

2. Sémantique

Cette fonction permet de sortir de la procédure courante en retournant éventuellement des valeurs en paramètres de sortie.

3. Code généré

Le résultat de chaque expression est rangé dans des registres successifs à partir de R0.
Un branchement indirect sur le registre de retour de la procédure courante est ensuite généré.

4. Exemples

```
RETURN;

PROCEDURE EXEMPLE_RETURN;
BEGIN


    RETURN(-1,"ABCD");          & EST EQUIVALENT A
                                RO := -1;
                                R1 := "ABCD";
                                GOTO $RF;      &
END;
```

7.2.5. Fonction GEN

1. Syntaxe

<fonction GEN> ::= GEN (<expression arithmétique entière>)

2. Sémantique

La fonction GEN permet de générer un mot dans la section de code des instructions. La valeur est donnée par l'expression.

3. Exemple

```
GEN(-1 + SYSPTRCARD);
GEN(#80000000);
```

8. APPEL DE PROCEDURE

1. Syntaxe

<appel de procédure> ::= <identifieur de procédure>
<identifieur de procédure> (<liste de paramètre>)

<identifieur de procédure> est un identifieur de procédure interne ou externe.

2. Sémantique

Cet appel permet de donner le contrôle a un sous-programme. Si une liste de paramètres est fournie, le résultat de chaque expression est rangé dans des registres successifs à partir de RO. Les paramètres sont donc passés par valeur et dans les registres.

3. Code généré

En plus du chargement éventuel des registres avec les valeurs de paramètres, une instruction BAL à l'adresse de la procédure est générée. Le registre de l'instruction BAL est celui précisé dans la déclaration de procédure.
Si la procédure est une procédure externe, une séquence d'initialisation des bases est générée après l'instruction BAL.

4. Exemples

```
VIDE;
ERREUR(SYSPTRCARD);
ERREUR(-1);
```

9. ADRESSAGE ET SEGMENTATION

Le fonctionnement en mode C de l'IRIS 80 est tel que les instructions ne peuvent contenir des adresses que sous forme relative par rapport à la valeur d'une base.
La portée de l'adressage d'une base est limitée à 16 Kmots. Les instructions qui composent un programme LP80 ne peuvent donc adresser des données que dans cette limite, et la dimension d'un programme LP80 doit être inférieure à celle-ci.
Il peut arriver -cela est hautement souhaitable pour des raisons de clarté- que le programmeur soit désireux de décomposer un programme en plusieurs parties (différentes phases d'un traitement) soit parce que l'encombrement total du programme est supérieur à 16 Kmots, soit afin de rendre la compréhension du composant plus aisée.
Il est donc amené à découper ses données et son programme en différentes parties. Ce découpage peut être opéré selon les règles suivantes :

9.1. Segment principal

1. Syntaxe

```
&lt;segment principal&gt;  ::=  MAIN <instruction>.
                          MAIN <liste global externe> <instruction>.
<liste global externe>  ::=  <déclaration de bloc de données>
                           | <déclaration d'externe>
                           | <liste global externe> <déclaration d'externe>
                           | <liste global externe>
                             <déclaration de bloc de données>
```

2. Sémantique

Un segment principal est une suite d'instructions pouvant référencer des données locales ou des données déclarées dans des blocs de données.
Un segment principal peut appeler des procédures externes déclarées en tête du segment principal.
Un segment principal se termine par un appel moniteur de type M:RETURN sous SIRIS8.
Un segment principal ne peut pas comporter de point d'entrée.
Un segment principal est basé par la base B1 et les données locales par la base B2.

3. Code généré

Voir paragraphe 9.6.

4. Exemple

```
MAIN
EXTERNAL PROCEDURE TRAITEMENT;
BEGIN

     TRAITEMENT;


END.
```

9.2. Segments secondaires

1. Syntaxe

```
&lt;segment secondaire&gt; ::= <déclaration de segment> <instruction> .
                       | <déclaration de segment> <liste global  externe>
                                                  <instruction> .
<déclaration de segment> ::= SEGMENT PROCEDURE <identifieur non déclaré>
                           | SEGMENT PROCEDURE <identifieur non déclaré>
                                               (<caractéristiques>)
<caractéristiques> ::= <caractéristiques code-données>
                     | <caractéristiques code-données> , <identifieur de
                                                          registre>
                     | <identifieur de registre>
<caractéristiques code-données> ::= <caractéristiques code>
                                  | <caractéristiques code> <caractéristiques
                                                            données>
                                  | <caractéristiques données>
<caractéristiques code> ::= <base 1>
                          | <nombre entier>
                          | <base 1> , <nombre entier>
                          | , <nombre entier>
<caractéristiques données> ::= , <base 2>
                             | , NODATA
                             | , <base 2> , REENT
                             | , REENT
```

2. Sémantique

Un segment secondaire est une procédure externe et peut être appelé de la même façon qu'une procédure interne, soit dans un segment principal soit dans un segment secondaire. L'appel d'un segment externe est identique à l'appel d'une procédure interne, mais le compilateur génère à la suite une séquence de restauration des bases.

Les caractéristiques par défaut d'un segment secondaire sont :
- base du code : B1
- base des données locales : B2
- registre de retour : RF

Le programmeur peut modifier ces caractéristiques :

<base 1> désigne la base employée pour le code

<base 2> désigne la base employée pour les variables locales données locales.

<identifieur de registre> désigne le registre de retour du segment procédure.

<nombre entier> est le déplacement initial par rapport à la base. L'emploi de ce paramètre si aucune base n'est précisée pour le code entraine un adressage non basé du code.

REENT Les données locales ne sont pas réservées. Le programmeur doit fournir une zone de mémoire basée par la base indiquée lors de l'appel du segment.

NODATA le segment n'a aucune donnée locale.

3. Code généré

Voir 9.6.

4. Exemples

```
SEGMENT PROCEDURE TRAITEMENT
BEGIN


END.
```

Exemples de caractéristiques

```
(B3)             & CODE B3 DONNEES B2 RETOUR RF   &
(,B4)            & CODE B1 DONNEES B2 RETOUR RF   &
(RE)             & CODE B1 DONNEES B2 RETOUR RE   &
(B3,NODATA)      & CODE B3 PAS DE DONNEES LOCALES &
(B3,#2E00,B4)    & CODE B3 AVEC DEPLACEMENT INITIAL
                   DONNEES B4                     &
(,B2,REENT)      & CODE B1 DONNEES B2 REENTRANTES &
(#40,NODATA)     & CODE NON BASE IMPLANTE EN #40
                   PAS DE DONNEES LOCALES         &
```

9.3. Déclaration de procédures externes

1. Syntaxe

<déclaration d'externe> ::= EXTERNAL PROCEDURE <liste de procédure> ;

<liste de procédure> ::= <définition de procédure>
| <liste de procédure> , <définition de procédure>

<définition de procédure> ::= <identifieur non déclaré>
| <identifieur non déclaré> <identifieur de registre>

2. Sémantique

La déclaration de procédure externe permet l'appel ultérieur d'un segment procédure depuis le segment courant.
L'identifieur indique quel registre de retour devra être changé lors de l'appel.
Par défaut le registre RF est utilisé.

3. Code généré

Lors de la déclaration, aucun code n'est généré.
Lors de la première référence à la procédure ainsi déclarée, un mot contenant l'adresse de la procédure sera généré en fin de segment, dans le vecteur d'addresses externes.

4. Exemples

```
EXTERNAL PROCEDURE EXTERNE1;
EXTERNAL PROCEDURE EXTERNE2 , EXTERNE3 RE;
EXTERNAL SEGMENT TOLERANCE  & TOLERE AFIN DE FACILITER LE TRANSPORT
                              DE PROGRAMMES LP70 ***** &
```

N.B. Afin d'assurer la compatibilité avec LP70, la déclaration EXTERNAL SEGMENT est utilisable. Il est conseillé toutefois de se ramener à la forme LP80.

9.4. Déclaration de point d'entrée

La déclaration de point d'entrée dans le bloc de niveau 1 peut se faire au moyen de la déclaration d'étiquette de type ENTRY (voir § 5.7.).
Lors de la définition du point d'entrée, une séquence d'initialisation des bases est générée.
Cette séquence est en outre générée lors de l'ouverture du premier bloc d'un segment principal ou secondaire (niveau 1).
Cette séquence dépend des caractéristiques du segment, c'est-à-dire qu'elle peut éventuellement être vide s'il n'y a pas de base pour le code et les données et si les seules données accessibles sont de type réentrant ou fictif. (Voir § 9.6.).

9.5. Déclaration de blocs de données

La déclaration de bloc de données permet le partage entre différents segments d'un même ensemble de données, ou la définition pour un segment de bloc de données fictives permettant la gestion de structures à réalisations multiples.

<déclaration de bloc de données> ::= <déclaration de données globales> ;
| <déclaration de données communes>;
| <déclaration de données fictives> ;

9.5.1. Déclaration de bloc de données globales

1. Syntaxe

<déclaration de bloc de données globales> ::= <définition globale>
<déclaration de bloc>

<déclaration de bloc> ::= <élément déclaration de bloc>
| <déclaration de bloc> ;
<élément de déclaration de bloc>

<élément de déclaration de bloc> ::= <déclaration de mémoire>
| <déclaration de constante>

```
<définition globale> ::=  GLOBALDATA <identifieur non déclaré>
                                     <identifieur de base>
                       |  GLOBALDATA <identifieur non déclaré>
                                     <identifieur de base> <nombre entier>
```

2. Sémantique

Une GLOBALDATA permet de partager un ensemble de données entre différents segments. La GLOBALDATA doit être déclarée dans chaque segment qui utilise tout ou partie des données ainsi déclarées. (Une GLOBALDATA LP80 a la même utilisation qu'un COMMUN étiqueté en FORTRAN).

<identifieur de base> indique la base qui sera employée pour l'adressage de données contenues dans le bloc. Cette base est chargée par la séquence d'initialisation de bases. (Voir § 9.6.).

<nombre entier> indique la protection d'accès qui sera appliquée sur les données. Il peut prendre les valeur 0, 1, 2.

Par défaut la protection 0 est appliquée.

La signification de la protection est la suivante :

- 0 tout accès
- 1 lecture exécution
- 2 lecture seule.

3. Code généré

Les données déclarées dans une GLOBALDATA sont générées dans une DSECT ayant pour nom externe le nom de la GLOBALDATA.

4. Exemple

```
GLOBALDATA BLOCKGLOBAL B3
WORD A;
GLOBALDATA BLOCKGLOBAL2 B4 2   & ACCES EN LECTURE SEULEMENT &
WORD B = "BBBB";
```

9.5.2. Déclaration de bloc de données communes

1. Syntaxe

<déclaration de bloc de données communes> ::= <définition COMMON>
<déclaration de bloc>

<définition COMMON> ::= COMMONDATA <identifieur non déclaré>
<identifieur de base>
| COMMONDATA <identifieur non déclaré>
<identifieur de base><nombre entier>

2. Sémantique

Un COMMONDATA permet de déclarer un ensemble de données dans un ensemble logique. Les données ainsi déclarées peuvent être partagées entre plusieurs segments, mais contrairement au GLOBALDATA les données ne doivent pas être redéclarées par un COMMONDATA dans chaque segment. L'emploi d'un COMMONDATA est utile lorsque le programmeur veut partager des données entre plusieurs segments,mais qu'il ne désire utiliser ces données que dans une branche de recouvrement (OVERLAY). En effet les GLOBALDATA doivent figurer dans la racine, ce qui peut être gênant.
Un COMMONDATA ne devant être déclaré que dans un seul segment, la déclaration des variables dans un autre segment devra se faire au moyen d'un DUMMYDATA.
La base indiquée est chargée par la séquence d'initialisation des bases (§ 9.6.).

3. Code généré

Les données déclarées dans un COMMONDATA sont générées dans une CSECT ayant pour nom externe le nom du COMMONDATA.

4. Exemple

```
COMMONDATA BLOCKCOMMON B5
WORD C;
WORD D;
```

9.5.3. Déclaration de bloc de données fictives

1. Syntaxe

<déclaration de bloc de données fictives> ::= <définition de DUMMY>
<déclaration de bloc>

<définition de DUMMY> ::= DUMMYDATA <identifieur non déclaré>
<identifieur de base>

2. Sémantique

Un DUMMYDATA permet de définir une structure fictive de données, c'est-à-dire de décrire un ensemble de données qui ne sont pas générées par le compilateur mais dont les déplacements par rapport à la base du bloc sont mémorisées.
Cette possibilité permet de décrire facilement des blocs de données à réalisation multiple. (blocs chaînés, éléments de pile,...).

3. Code généré

Aucun code n'est généré.

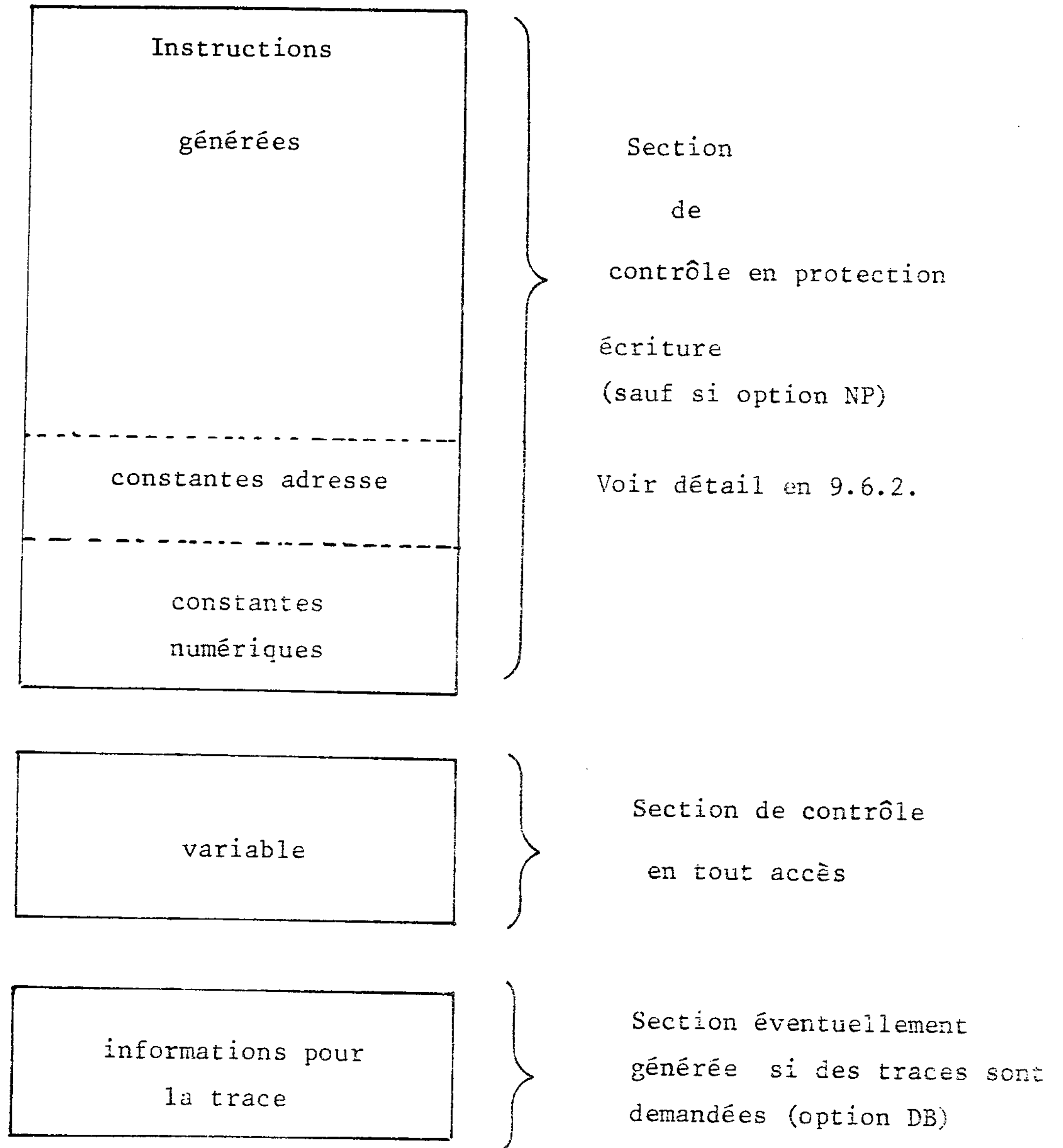
4. Exemple

```
DUMMYDATA BLOCKDUMMY B6
WORD CHAINAGE_AVANT;
WORD CHAINAGE_ARRIERE;
WORD DONNEES;
```

9.6. Structure du code généré

9.6.1. Structure générale

Un programme compilé en LP80 donne un module de structure suivante :



Trois sections sont toujours générées :

- La première est toujours de longueur non nulle.

  Elle contient :

  + le code des instructions
  + les constantes adresses implicites
  + les constantes numériques
  + les blocs de contrôle (en particulier les DCB) générées lors de l'emploi de macro-instructions et qui doivent être en protection écriture.

  Cette section est en protection écriture afin d'assurer la non modification accidentelle de son contenu.
  Cependant l'emploi de l'option NP (voir chapitre 11) permet de lever cette obligation.

- La deuxième peut être de longueur nulle si aucune variable n'est utilisée par le programme compilé.

- La troisième section contient des informations utiles pour la TRACE (Voir paragraphe 11.1.) lorsque celle-ci est demandée.

En plus de ces trois sections, le compilateur peut générer des sections correspondant aux COMMONDATA ou aux GLOBALDATA.

9.6.2. Structure détaillée de la section programme

Nous donnons ci- après un exemple permettant d'illustrer le code généré.

```
SEGMENT  PROCEDURE  PROC1
    :
GLOBALDATA  BLOC1  B3
WORD    W1;
    :
EXTERNAL PROCEDURE PROC2,PROC3;
    :
BEGIN   ------------>   LIAI (B1,-PTRPROG)
    :                   LBR  (B2,=A(variables))      (1)
    :                   LBR  (B3,=V(BLOC1))
ENTRY EPROC1;              :
EPROC1: ------------>   LIAI (B1,-PTRPROG)
    :                   LBR  (B2,=A(variables))      (1)
    :                   LBR  (B3,=V(BLOC1))
    :                      :
PROC2;  ------------>   BAL (RF,$ = V(PROC2))
    :                   LIAI (B1,-PTRPROG)
    :                   LBR  (B2,=A(variables))
    :                   LBR  (B3,=V(BLOC1))
    :
END.    ------------>   GOTO  $RF
```

CODE GENERE

- 1 mot contenant la longueur totale des variables
- 1 mot contenant l'adresse de la CSECT où sont rangées les variables
- 1 mot contenant l'adresse de BLOC1
- 1 mot contenant l'adresse de PROC2

(3)

CONSTANTES

DU

PROGRAMME

PROGRAMME

SOURCE

(1) Chaque point d'entrée dans la section PROC1 doit avoir une séquence d'initialisation des bases.

Cette séquence est la même pour tous les points d'entrée. Elle comporte:

- l'initialisation de la base programme au moyen de l'instruction LIAI avec comme facteur immédiat la valeur négative du compteur d'emplacement moins un.
  Cette instruction permet donc la référence ultérieure de toutes les parties de la section de programme.
- Les bases correspondant à des sections de données peuvent donc être rechargées au moyen de l'instruction LBR qui référence les constantes adresses correspondantes.

(2) L'appel d'un programme externe est effectué par l'instruction BAL (Branch And Link) qui référence la constante adresse du programme externe considéré.
A la suite de cet appel, une séquence de restauration des bases est automatiquement générée.
Elle est identique à la séquence qui figure à chaque point d'entrée.

(3) A la suite de la dernière instruction, une suite de valeurs particulières est automatiquement générée.
Cette suite se compose :

- d'un mot qui contient la longueur totale en mots des variables déclarées dans le programme.
  Ce mot peut être référencé par l'utilisateur, au moyen de l'identifieur SYSLNGDATA
- éventuellement des constantes adresse nécessaires à l'adressage.
  Dans l'exemple donné on a 3 constantes adresses :
  + l'adresse de la section qui contient les variables
  + l'adresse de la section fictive BLOC1
  + l'adresse de la procédure externe PROC1

Seules les constantes nécessaires (c'est-à-dire référencées explicitement ou implicitement) sont générées.
Dans l'exemple donné on voit que la constante adresse de PROC3 n'est pas générée.

(4) En fin de section sont générées toutes les constantes numériques ainsi que certains blocs de constantes utilisées par les macros-instructions de SIRIS8.

10. MACRO-INSTRUCTIONS DE SIRIS8

La plupart des macro-instructions de SIRIS8 sont inclues dans le compilateur LP80.
Les services systèmes privilégiés n'ont pas été introduits, leur emploi par les utilisateurs n'étant pas possible.

10.1. Fonctions

Les fonctions et les paramètres par défaut sont les mêmes que ceux définis par le constructeur dans la brochure :

PROCEDURES SYSTEMES sous SIRIS8

10.2. Liste des macro-instructions

Les macro-instructions suivantes sont disponibles en LP80.

ASSIGN	LINK
CHECK	MOVEDCB
CHKPT	NOTE
CLOSE	OPEN
CVOL	POINT
DCB	PRINT
DELREC	PUT
DEVICE	PUTFSC
ERR	READ
FIND	RESTART
FP	RETURN
FPC	RSS
FSP	SEGLD
GET	SETDCB
GETDAY	SSS
GETFSC	STIMER
GL	STOW
GLC	TIME
GP	TIMER
GPC	TRAP
GSP	TRUNC
IDLE	TSS
INT	TYPE
KEYIN	WAIT
LDTRC	WRITE

10.3. Notation

Afin de ne pas alourdir la présentation des macro-instructions nous avons adopté une notation mixte, peu orthodoxe, mais de lisibilité plus grande. La sémantique des différents paramètres est donnée par la brochure citée en 10.1.

```
<liste clé trap> ::= <clé trap>
                   | <clé trap> , <liste clé trap>
<clé trap> ::=  FX | DEC | CL4 | CL3 | CL2 | PS | UI
              | MPV | PSM | NMA | NI | ABRT | NAO | ALL
<liste clé abn> ::= <clé abn>
                  | <clé abn> , <liste clé abn>
<clé abn> ::= X1 | X2 | X3 | X4 | X5 | X6
<listewait> ::= <élément de wait>
              | <élément de wait> , <listewait>
<élément de wait > ::= $@ <identifieur de registre>
                     | $@ <identifieur de pointeur>
                     | <adr2>
<adr2> ::= @ <identifieur de variable>
         | @ <identifieur de pointeur>
         | @ <identifieur d'étiquette>
         | @ <identifieur de procédure>
<val> ::= <nombre entier>
        | <adr3>
<adr3> ::= <identifieur de variable>
         | <identifieur de pointeur>
         | <identifieur de registre>
         | $ <identifieur de pointeur>
         | $ <identifieur de registre>
         | <adr2>
<adr1> ::= <val>
<liste nombre entier> ::= <nombre entier>
                        | <nombre entier> , <liste nombre entier>
<liste val> ::= <val>
              | <val> , <liste val>
<liste adr> ::= <liste val>
```

On note [ ] un ensemble de paramètres optionnels

et { } des alternatives entre paramètres

Les symboles métalinguistiques < > ont été supprimés afin de rendre la notation plus claire.

10.4. Syntaxe

```
:GL
:GLC
:GP        val
:GPC       val
:FP        val
:FPC       val
:GSP       val
:FSP       val[,adr1]

:SEGLD     adr1
:LDTRC     adr1
:LINK      adr1

                                                   (FX  )
:TRAP      adr2[,(TRAP,liste cle trap)][,(IGNORE,{BOTH})][,(OST,adr1)]
                                                   (DEC )

:TRAP      RESTORE,adr1
:WAIT      nombre entier,liste wait
:RETURN    [adr1]
:ERR       val
:IDLE

:SSS       liste nombre entier
:RSS       liste nombre entier
:TSS       nombre entier
:GETFSC    adr1
:PUTFSC    adr1

:PRINT     [(MESS,adr1)]
:TYPE      [(MESS,adr1)]
:KEYIN     [(MESS,adr1)][,(REPLY,adr1)][,(SIZE,val)]
:INT       adr2

:TIME      adr1
:GETDAY    val
            (SEC)
:STIMER    ({MIN},val),adr1
            (TUN)

           (SEC)
:TIMER     {MIN} [,CANCEL]
           (TUN)
:MOVEDCB adr1,(PTR,adr1)
:ASSIGN    (OPL,{"TEXT"})[,(SIZ,val,val)]
                {adr3  }
:ASSIGN    (OPL,{"TEXT"}),(UNT,OPL,{"TEXT"}),(NAM, adr1 )
                {adr3  }            {adr3  }
                 (NEW)
           [,(STS,{OLD})][,(SIZ,val,val)]
                 (MOD)
```

```
                   (I)
                   (B)     (CID)
:OPEN     adrl,    {U} [,[ {PID} ][,ICY]]
                   (O)     (NID)
                   (S)
                (HLD)      (LVE)
:CLOSE    adrl[,{RLS} ][,{RRD} ][, {NCG} ]
                (MTN)      (RWD)     {KEP}

:GET      adrl,(REC,adrl)[,(RSA,adrl)][,(KEY,adrl)]
:PUT      adrl,(REC,adrl)[,(ARS,adrl)][,(NWK)][,(DFW)]
:TRUNC    adrl
:DELREC   adrl
:CVOL     adrl
:NOTE     adrl,(RCI,adrl)
:POINT    adrl,(RCI,adrl)
:DEVICE   adrl
          [,(CHF,adrl)]
          [,(PAG)]
                  (BKS)
                  (FWS)
          [,(POS, {BOF} )]
                  (EOF)
          [,(WEOF)]
          [,(TRD, {OFF} )]
                  {ON }
          [,(TPH, {OFF} )]
                  {ON }
          [,(MTR, {OFF} )]
                  {ON }
          [,(FRM, {OFF} )]
                  {ON }
          [,(ECH, {OFF} )]
                  {ON }
          [,(IND ( [, {val} ]          ) )]
                 (     {ALL}           )
                 ( , {POL} [,val]      )
                 (   {SEL}             )
          [,(OPN)]
               (BEL)
          [,( {SUS} )]
               (ABO)
          [,(MOD, {BIN} )]
                  {EBC}
:FIND     adrl,(KEY,adrl)[,(RCI,adrl)]
                            (SYN)
:STOW     adrl,(KEY,adrl)[, {DEL} ]
                            (ADD)
:READ     adrl,(BUF,adrl)[,(PTR,adrl)][,(TRL,val)]
             ((SUR)              )
          [, {(CNV,adrl,val)} ][,(ADR,adrl)][,(IND,val)]
             ((ATT)              )
:WRITE    adrl,(BUF,adrl)[,(PTR,adrl)][,(TRL,val)]
          [,(FIN),( {REP} )][,(ADR,adrl)]
                    {DIS}
:CHECK    adrl[,(PTR,adrl)][,(RSA,adrl)][,(IND,adrl)][,(NAM,adrl)]

:CHKPT    adrl[,adrl]
:RESTART  adrl[,adrl]
```

```
:DCB      identifieur non déclaré (OPL,"TEXT")
                   (I)
                   )D(
          [,(ORG,  )C( )]
                   (P)
                   (V)
          [,(FRM,  )F( )]
                   (U)
          [,(REL,nombre entier)]
          [,(KYP,nombre entier)]
          [,(KYL,nombre entier)]
          [,(BKL,nombre entier)]
          [,(MXL,nombre entier)]
          [,( (BHR,nombre entier) )]
              (PFX,"C"          )
          [,(DLC)]
              (NULC)
          [,( )NBC ( )]
              (ULC )
          [,(NBF,nombre entier)]
          [,(BFA,adr2)]
          [,(SIM,nombre entier)]
          [,( (LOC) )]
              (MOV)
          [,(NRT)]
                  AI
                 (AP)
          [,(AM, )AS( )]
                 (VS)
                 (VD)
                 (BD)
                  BT
          [,(ERR,adr2)]
          [,(ABN,adr2,liste clé abn)]
          [,(TLB,adr2)]
          [,(CKP,adr2)]
          [,( (VFC) )]
              (NVF)
          [,(LIN,nombre entier)]
          [,(SPC,nombre entier[,nombre entier])]
          [,(DTA,nombre entier)]
                  (BIN)
                  (BCD)
          [,(MOD, )EBC( )]
                  (PK )
                  (UPK)
                  (ASC)
          [,(SEQ[,"TEXT"])]
          [,( (TAB,liste val) )]
              (LST,liste adr)
          [,(HDR,nombre entier,adr2)]
          [,(CNT,nombre entier)]
```

```
:SETDCB  adr1
         [,(OPL,"TEXT")]
                 (I)
                 )D(
         [,(ORG, )C( )]
                 (P)
                 (V)
         [,(FRM, {F} )]
                 (U)
         [,(REL,val)]
         [,(KYP,val)]
         [,(KYP,val)]
         [,(BKL,val)]
         [,(MXL,val)]
         [,({BHR,val} )]
           {PFX,"C"}
         [,(DLC)]
            (NULC)
         [,({NBC } )]
            (ULC )
         [,(NBF,val)]
         [,(BFA,adr1)]
         [,(SIM,val)]
         [,({LOC} )]
           {MOV}
         [,(NRT)]
                  AI
                 (AP)
         [,(AM, (AS) )]
                 {VS}
                 (VD)
                 (BD)
                  BT
         [,(ERR,adr2)]
         [,(ABN,adr1,liste clé abn)]
         [,(TLB,adr1)]
         [,(CKP,adr1)]
         [,({VFC} )]
           {NVF}
         [,(LIN,val)]
         [,(SPC,val[,val])]
         [,(DTA,val)]
         [,(MOD, val )]
                 (BIN)
                 (BCD)
                 {EBC}
                 (PK )
                 (UPK)
                  ASC
         [,(SEQ[,{"TEXT"}])]
                 {val   }
         [,( TAB,liste val )]
             LST,liste adr
         [,(HDR,val,adr1)]
         [,(CNT,val)]
```

10.5. Code généré

On distingue 2 groupes de macro-instructions :
- celles qui donnent lieu à la création de données (FPT)
- les autres

Les premières peuvent nécessiter la création de données en protection écriture. Dans ce cas celles-ci sont générées dans la zone des constantes de la section de programme (Exception :DCB).

Pour toutes les macros (à l'exception de :DCB) on a la génération des instructions suivantes :
- LCFI
- STM des registres en #3E02
- chargement des registres avec les valeurs nécessaires
- CAL1

Cas particulier de :DCB

Les DCB sont générés dans la section programme.

```
          :
          :
          - branchement inconditionnel en L1
          - suite de 18 mots définissant le DCB
    L1:   :
          :
```

L'utilisation du DCB n'est possible que s'il figure dans des emplacements mémoire en protection écriture.

11. FACILITES DE MISE AU POINT

Le langage LP80 offre des facilités d'aide à la mise au point. Ces aides sont de 2 types :

- trace lors de l'exécution
- impression de valeurs lors de l'exécution

La trace permet de connaître le comportement dynamique d'un programme en faisant apparaître les appels de sous-programmes et/ou les passages sur des étiquettes, ainsi que les affectations de variables.
L'impression de valeurs (SNAP) lors de l'exécution permet de connaître le contenu de la mémoire et/ou des registres à un instant donné.

11.1. Les ordres de trace

11.1.1. Déclaration de TRACE

1. Syntaxe

```
<déclaration de trace>       ::= TRACE <liste identifieur trace>
<liste identifieur trace>    ::= <identifieur trace>
                               | <liste identifieur trace> ,
                                 <identifieur trace>
<identifieur trace> ::= <identifieur de variable> <déplacement-indextation>
                      | <identifieur de logique> <déplacement-indexation>
                      | <identifieur silence>
<identifieur silence> ::= <identifieur de variable>
                        | <identifieur de registre>
                        | <identifieur de base>
                        | <identifieur de logique>
                        | <identifieur de pile>
                        | <identifieur de procédure>
                        | <identifieur d'étiquette>
```

2. Sémantique

La déclaration de trace permet de suivre l'évolution du programme par l'impression de messages lors des appels de procédures, du passage sur des étiquettes et/ou les assignations de variables et de registres.

3. Code généré

Lorsqu'un identifieur fait l'objet d'une déclaration de TRACE, certaines apparitions de cet identifieur entrainent la génération de deux mots.
Cette génération se produit :
+ lors d'une instruction de modification du contenu d'une variable, d'un registre ou d'une base,
+ lors de l'appel d'une procédure interne ou externe
+ lors de la définition d'une étiquette
+ lors de l'utilisation d'une pile.

Les deux mots générés se composent :
- d'un appel superviseur de type CAL4
- d'un mot d'information

En outre des informations concernant l'identifieur (nom, type, adresse) sont générés dans la section de contrôle prévue à cet effet.

4. Effet à l'exécution

Lors de l'exécution de l'instruction CAL4, un déroutement se produit. Ce déroutement est récupéré par l'accompagnateur LP80 qui doit avoir été lié au programme.
Un message d'information est imprimé. Il comporte le numéro de ligne source du programme ainsi que les valeurs exprimées en hexadécimal.

5. Exemple de TRACE

Voir exemple complet au §11.2.

11.1.2. Déclaration de fin de trace

1. Syntaxe

<déclaration de fin de trace> ::= NOTRACE <liste identifieur trace>

2. Sémantique

Cette déclaration annule les effets de l'ordre TRACE précédent.

3. Code généré

Aucun code n'est généré pour cette déclaration.

4. Exemple

Voir exemple complet au §11.2.

11.1.3. Instruction SILENCE

1. Syntaxe

<instruction SILENCE> ::= SILENCE <liste identifieur silence
| SILENCE ALL

<liste identifieur silence> ::= <identifieur silence>
| <liste identifieur silence> , <identifieur silence>

2. Sémantique

Cette instruction permet d'arrêter dynamiquement les effets d'un ordre TRACE.

3. Code généré

Pour chaque identifieur référencé par l'instruction SILENCE deux mots sont générés :

- un appel superviseur de type CAL4
- un mot d'information pour l'accompagnateur.

4. Exemple

Voir exemple complet au §11.2.

11.1.4. Instruction NOSILENCE

1. Syntaxe

<instruction NOSILENCE> ::= NOSILENCE <liste identifieur silence>
| NOSILENCE ALL

2. Sémantique

L'instruction NOSILENCE permet de reprendre dynamiquement la trace éventuellement arrêtée par une instruction SILENCE.

3. Code généré

Pour chaque identifieur référencé par l'instruction NOSILENCE, deux mots sont générés :
- un appel superviseur de type CAL4
- un mot d'information pour l'accompagnateur

4. Exemple

Voir exemple complet au §11.2.

11.2. L'instruction SNAP

1. Syntaxe

<instruction SNAP> ::= SNAP (<paramètre de SNAP>)
<paramètre SNAP> ::= <zone mémoire>
| <zone mémoire> , ALL
<zone mémoire> ::= <identifieur>
| <identifieur> , <identifieur>

2. Sémantique

En fonction des paramètres de SNAP l'impression des valeurs des zones désignées est obtenue en hexadécimal.
Si un identifieur est indiqué, on obtient le "dump" de cet identifieur en hexadécimal.
Si deux identifieurs sont indiqués, on obtient le "dump" de toute la zone mémoire comprise entre ces deux identifieurs.

Le fait de mettre le paramètre ALL permet d'obtenir la valeur courante du PSD et des registres généraux et de base.

3. Code généré

En fonction des paramètres de SNAP deux ou trois mots sont générés :
- un appel superviseur de type CAL4
- un ou deux mots d'information pour l'accompagnateur.

4. Exemple

Les deux listes suivantes illustrent l'emploi des possibilités d'aide à la mise au point lors de la compilation.
La première liste est la liste de compilation.
Celle-ci a été faite avec l'option DB.
Le programme principal MAIN commence donc par un appel implicite au metteur au point pour l'initialisation du système.
La deuxième liste est le résultat de l'exécution du programme.
On peut noter le message d'initialisation en début de liste.

```
COMPILATEUR LP80 - VERSION 2.2 -  CENTRE INTERUNIVERSITAIRE DE CALCUL DE GRENOBLE -  JUIN  1977
  CARTE     @DON  @INS  PF
  00001     0000  0000  00      MAIN
  00002     0000  0000  00      EXTERNAL PROCEDURE TEST;
  00003     0000  0000  00      BEGIN
  00004     0000  0004  01      WORD A,B,C,D;
  00005     0004  0004  01      LOGICAL L;
  00006     0005  0004  01      LABEL ETIQ;
  00007     0005  0004  01      TRACE A,B,L,ETIQ,TEST;
  00008     0005  0004  01      R0 := #1234;
  00009     0005  0005  01      A := C := R0;
  00010     0005  0009  01      ETIQ:
  00011     0005  000B  01      L := R0 = A;
  00012     0005  0013  01      TEST;
  00013     0005  001A  01      END.
 AUCUNE ERREUR
COMPILATEUR LP80 - VERSION 2.2 -  CENTRE INTERUNIVERSITAIRE DE CALCUL DE GRENOBLE -  JUIN  1977
  CARTE     @DON  @INS  PF
  00014     0000  0000  00      SEGMENT PROCEDURE TEST
  00015     0000  0000  00      BEGIN
  00016     0000  0002  01      WORD A;
  00017     0001  0002  01      TRACE A,R1;
  00018     0001  0002  01      A := R0;
  00019     0001  0005  01      NOTRACE A;
  00020     0001  0005  01      A := R0 := 0;
  00021     0001  0007  01      SILENCE R1;
  00022     0001  0009  01      R1 := 4;
  00023     0001  000C  01      NOSILENCE R1;
  00024     0001  000E  01      R1 := 8;
  00025     0001  0011  01
  00026     0001  0011  01      ARRAY 10 WORD B = *5(-1) *5(1);
  00027     000B  0011  01      ARRAY 4 BYTE OCTET = "BYTE";
  00028     000C  0011  01      SNAP(A);
  00029     000C  0013  01      SNAP(A,OCTET);
  00030     000C  0016  01      SNAP(B,OCTET,ALL);
  00031     000C  0019  01
  00032     000C  0019  01
  00033     000C  0019  01      END.
 AUCUNE ERREUR
```

```
* ACCOMPAGNATEUR MODE C VERSION 3 * 14*38*38*44 19*08*77
->C
 00009: A := #00001234
 00010: PASSE PAR ETIQ
 00011: L := TRUE
 00012: APPEL DE TEST
 00018: A := #00001234
 00024: R1 := #00000008
 SNAP A LA LIGNE  00028
 003E58                          00000000                                                        ....
 SNAP A LA LIGNE  00029
 003E58                          00000000  FFFFFFFF  FFFFFFFF  FFFFFFFF  FFFFFFFF  FFFFFFFF        ..............
...
 003E60      00000001  00000001  00000001  00000001  00000001  C2E8E3C5                  ...................BY
 SNAP A LA LIGNE  00030
 PSD         E0FA4239  0800390F
 GPR 0-7     00000000  00000008  00000000  00000000  00000000  00000000  00000000  00000000
 GPR 8-F     00000000  00000000  00000000  00000000  00000000  00000000  00000000  00004216
 BASES       00000000  00004220  00003E5A  0000C000  00010000  00014000  00018000  0001C000
 003E58                                    FFFFFFFF  FFFFFFFF  FFFFFFFF  FFFFFFFF  FFFFFFFF        ..........
...
 003E60      00000001  00000001  00000001  00000001  00000001  C2E8E3C5                  ...................BY
 00012: RETOUR DE TEST
```

11.3. L'accompagnateur LP80

11.3.1. Buts

L'accompagnateur LP80 a pour buts :

- de prendre en compte le traitement des ordres TRACE et SNAP demandés lors de la compilation
- de surveiller l'exécution générale d'un programme (prise en compte des déroutements)
- de permettre la mise au point intéractive de programmes écrits en LP80 (ou en tout autre langage générant du code en mode C).

11.3.2. Mise en oeuvre

L'accompagnateur peut être employé de 2 manières distinctes :

- Soit l'utilisateur procède à l'utilisation d'instructions de mise au point, (TRACE, SNAP) et précise l'option DB.

  Cela a pour effets :

  * de prendre en compte les instructions TRACE et SNAP qui figurent éventuellement dans le programme et de générer les appels correspondant au metteur au point.
  * s'il s'agit d'un segment principal (MAIN), de générer au début du segment un appel d'initialisation au metteur au point.

    Si aucun segment principal n'existe, l'utilisateur doit faire explicitement l'appel d'utilisation au segment de mise au point CDEBUG/

    EXEMPLE :

```
SEGMENT PROCEDURE TEST
EXTERNAL PROCEDURE CDEBUG;

BEGIN
     CDEBUG;          & APPEL D'INITIALISATION
                        DU METTEUR AU POINT     &

END.
```

- Soit l'utilisateur désire employer l'accompagnateur alors que son programme n'a pas été compilé avec l'option DB. Dans ce cas, il existe à sa disposition un processeur de mise au point qui permet d'inclure dynamiquement l'accompagnateur lors de l'exécution.
  Le mécanisme employé consiste à prendre en compte le Module Chargeable qui doit être exécuté et à le modifier afin :
  . de charger en zone commune dynamique le metteur au point,
  . de donner le contrôle à celui-ci en fournissant comme contexte initial le contexte standard de SIRIS8, l'adresse figurant dans le PSD étant l'adresse du point d'entrée initial.

EXEMPLE :

```
!ASSIGN LM,FIL,(STS,OLD),(NAM,nom du module)
!SDEBUG %OPTIONS
```

Les options peuvent être passées de manière standard. En résumé l'utilisateur n'a qu'à remplacer la carte RUN par une carte SDEBUG. Dans les deux cas, l'utilisateur dispose des mêmes possibilités de mise au point à l'exécution.

Etape d'exécution

L'accompagnateur dispose de périphériques logiques qui permettent la communication avec l'utilisateur.

Ces périphériques sont au nombre de cinq :

- imprimante logique : destinée à recevoir le flot d'information en provenance de la TRACE et des SNAPS.
- lecteur de carte logique : permet de fournir des ordres de mise au point.
- machine à écrire (sortie)
- machine à écrire (entrée) : pour les échanges conversationnels de mise au point.
- périphérique rapide : sur lequel pourrait être archivé l'ensemble des échanges par exemple ou bien des DUMPS importants.

Ces périphériques ont reçu pour noms :

LS - SI - TI - TO - LO

+ BATCH SIRIS8 :

Sous SIRIS8 les périphériques LS, TI, TO sont toujours existants. (LS est assigné au LISTING-LOG, TI/TO étant soit la console opérateur, soit le terminal utilisateur).
Les périphériques LS, SI, LO peuvent être assignés ou réassignés au moyen d'une carte ASSIGN se rapportant aux étiquettes logiques :

- DBLS - DBSI - DBLO

N.B. : si DBSI est assigné en DEV, IN, il faudra qu'il soit le premier fichier carte, car il est lu par l'accompagnateur avant le lancement de toute exécution.

+ TIME-SHARING SIRIS8 :

Lors de l'initialisation, l'accompagnateur envoie un message indiquant en particulier la date et l'heure du début de l'exécution. L'accompagnateur se met ensuite dans l'environnement de mise au point.

+ Autre système :

On doit obtenir une utilisation sensiblement identique à celle de SIRIS8.

11.3.3. Les requêtes de mise au point

11.3.3.1. Règles générales

Les requêtes se composent d'un mot-clé qui peut être suivi de 0,1 ou 2 paramètres.

Les paramètres sont des nombres hexadécimaux dont la valeur est comprise entre

00000000 et FFFFFFFF

Tout nombre hexadécimal est ramené à une valeur comprise entre ces deux bornes.

EXEMPLE :

07F est équivalent à 0000007F

123456789 est équivalent à 23456789

Le séparateur entre deux paramètres peut être :

soit le caractère - (tiret)

soit le caractère = (signe égal)

soit l'un des 4 opérateurs arithmétiques

EXEMPLES DE REQUETES VALIDES :

BASE 7

REGISTRE 3=1FFF

AFFICHER 4000-4008

PSD

Le nom de la requête doit être séparé des paramètres par au moins un blanc. Toutes les adresses sont relatives à une origine qui est modifiable par requête et dont la valeur initiale est 0.

11.3.3.2. Liste des requêtes

N.B. La notation suivante est employée.

- Les caractères soulignés indiquent les caractères minimums nécessaires pour la reconnaissance de la requête.
- Le signe | (barre verticale) indique une alternative.

AFFICHER Adr1-Adr2

+ le contenu du mot situé à l'adresse Adr est affiché sur le terminal en hexadécimal.

+ le contenu des mots situés entre Adr1 et Adr2 compris est affiché sur le terminal.

BASES pas de paramètre | Arg1 | Arg1-Arg2 | Arg1=Valeur

Avec Arg1 et Arg2 = numéro de registre de base et $0 \leqslant Arg1 < Arg2 \leqslant 7$

+ Pas de paramètre : afficher en hexadécimal tous les registres.

+ Arg1 : afficher le registre désigné

+ Arg1-Arg2 : afficher les registres Arg1 à Arg2

+ Arg1=Valeur : affecter au registre Arg1 la valeur indiquée.

CONTINUER pas de paramètre | Adr1

+ pas de paramètre : l'exécution est reprise en séquence.

+ Adr1 : l'exécution est reprise en Adr1.

DEVEROUILLER Adr1 | Adr1-Adr2

Cette requête permet de supprimer la protection d'accès à une page.

+ Adr1 : la page contenant le mot d'adresse Adr1 est mise en tout accès

+ Adr1-Adr2 : toutes les pages comprises entre Adr1 et Adr2 (y compris) sont mises en tout accès.

ATTENTION : Sous SIRIS8 tenir compte des DCB qui peuvent être implantés dans les pages ainsi référencées et qui ne pourraient plus être utilisés.

ENLEVER Adr1

Le point d'arrêt posé à l'adresse Adr1 est supprimé.

HEURE pas de paramètre

La date et l'heure sont affichés sur le terminal.

IMPRIMER Adr1-Adr2

Le contenu des mots compris entre Adr1 et Adr2 est listé sur l'imprimante sous forme hexadécimale avec traduction en caractères dans la marge.

LISTER pas de paramètre

Le contexte de la mise au point est listé sur le terminal. C'est-à-dire que la valeur courante de l'ORIGINE et les points d'arrêts posés (adresse et instruction) sont imprimés.

MODIFIER Adr=Valeur

Le contenu du mot situé à l'adresse Adr prend la valeur indiquée.

ORIGINE pas de paramètre | Adr

+ pas de paramètre : l'origine est remise à 0.
+ Adr : l'origine prend pour valeur l'adresse indiquée.

PSD pas de paramètre

Le PSD est affiché au terminal.

REGISTRES pas de paramètre | Arg1 | Arg1-Arg2 | Arg1=Valeur

Avec Arg1 et Arg2 = numéro de registre général et $0 \leq \text{Arg1} < \text{Arg2} \leq F$

+ pas de paramètre : afficher en hexadécimal et en décimal tous les registres.
+ Arg1 : afficher le registre désigné
+ Arg1-Arg2 : afficher les registres Arg1 à Arg2
+ Arg1=Valeur : affecter au registre Arg1 la valeur indiquée.

STOP Adr

Adr désigne une adresse où l'exécution sera suspendue avant que l'instruction en place soit exécutée.

TERMINER pas de paramètre | ARG1

L'exécution est abandonnée.
Si l'argument n'est pas nul,un dump de la tache est obtenu.

VEROUILLER Adr1 | Adr1-Adr2

Cette requête permet de forcer la protection d'accès à une page en lecture-exécution.

+ Adr1 : la page contenant le mot d'adresse Adr1 est mise en accès lecture-exécution.
+ Adr1-Adr2 : toutes les pages comprises entre Adr1 et Adr2 (y compris) sont mises en accès lecture-exécution.

ATTENTION : L'utilisateur doit s'assurer qu'aucun point d'arrêt n'a été posé dans les pages concernées.

% Arg1 opérateur Arg2

La commande % permet d'effectuer des opérations simples en hexadécimal. Les opérateurs possibles sont :

+ pour l'addition
- pour la soustraction
/ pour la division
* pour la multiplication

EXEMPLE :

% 3A2-82

Le résultat suivant est renvoyé :

00000320

11.2.3.3. Accès à l'environnement de mise au point

On entre dans l'environnement de mise au point de 4 façons :

+ lors de l'initialisation
+ lors de la rencontre d'un point d'arrêt
+ lors d'une erreur programme (TRAP)
+ au moyen d'une interruption externe.

N.B. Sous TS /SIRIS8 une interruption externe est obtenue par transmission d'une Attention 2.

11.2.3.4. Environnement de mise au point

L'attente d'une requête est signalée par l'impression du message :
Après l'introduction d'une requête plusieurs cas se présentent :

+ La requête est reconnue, elle est exécutée
  S'il s'agit de la requête CONTINUER le contrôle est rendu au programme utilisateur, sinon on reste dans l'environnement de mise au point.
+ La requête n'est pas reconnue
  Le message suivant est envoyé :
  ???
+ La requête est reconnue, les paramètres sont erronés :
  Le message suivant est envoyé :
  ARGUMENT(S)!
  L'utilisateur doit alors retaper la requête.

11.3.3.5. Erreurs lors de l'exécution d'une requête

+ Requête STOP
  TROP DE POINTS D'ARRET
  Il y a déjà 31 points d'arrêt de posés.
+ Requêtes STOP, AFFICHER, IMPRIMER, CONTINUER, MODIFIER
  Les adresses référencées peuvent être invalides (protection, adressage,...).Dans ce cas un message explicite est envoyé et la requête n'est pas exécutée.

11.3.4. Restrictions sous SIRIS8

L'utilisation de l'accompagnateur LP80 entraine quelques restrictions de la programmation sous SIRIS8.

+ L'usage des services système :INT et :TRAP est interdit, l'interface de l'accompagnateur utilisant ces services. Toutefois, dans le cas où l'emploi de ces services est indispensable (?), l'utilisateur devra réinitialiser l'accompagnateur en faisant un appel explicite.(Point d'entrée CDEBUGINIT). Ceci est cependant fortement déconseillé.
+ Pour poser des points d'arrêt, le programme doit être en accès écriture autorisé. Il faut donc compiler avec l'option NP (No memory Protection). ou employer la requête DEVEROUILLER.

Attention : La déclaration de DCB dans un segment compilé avec l'option NP entraîne la suppression de cette option et par là-même la possibilité de poser des points d'arrêt.
Il y a incompatibilité entre l'option NP et la déclaration de DCB. Le DCB l'emporte.

12. MISE EN OEUVRE DU COMPILATEUR LP80

12.1. Commandes au compilateur

Le compilateur LP80 peut recevoir des ordres incorporés au programme symbolique. Ces commandes sont constituées du caractère / (base de fraction) positionné sur la première colonne d'une ligne, suivi d'un mot-clé.
Il existe six commandes :

/LISTON demande de listage
/LISTOF arrêt du listage
/CODEON demande d'impression du code objet
/CODEOF arrêt de l'impression du code objet
/NEWPAGE saut de page
/COPY appel d'une partition

12.1.1. Commandes LISTON, LISTOF

Ces commandes permettent de gérer l'impression de la liste. Par défaut le compilateur commence la compilation d'un segment avec l'impression (LISTON).
L'impression n'est possible que si l'option LS a été demandée. Seules les lignes comprises entre une commande LISTON et LISTOF sont imprimées.

12.1.2. Commandes CODEON, CODEOF

Ces commandes permettent de gérer l'impression en hexadécimal du code généré.
Par défaut le compilateur commence la compilation d'un segment sans imprimer le code (CODEOF).
L'impression est faite au fur et à mesure de la génération, sur l'étiquette LO.
L'impression du code n'est possible que si du code est généré (option GO) et si l'impression est possible (option LS).

12.1.3. Commande NEWPAGE

Cette commande permet de poursuivre l'impression sur une nouvelle page de liste. Si la commande NEWPAGE survient lorsque l'impression est en début de page, aucun saut supplémentaire (page blanche) n'est commandé.

12.1.4. Commande COPY

Cette commande est suivie sur la même ligne d'un nom de partition. Elle permet de poursuivre la lecture du code source dans la partition indiquée, cette partition devant se trouver dans une au moins des bibliothèques indiquées par l'option OL.

La recherche de la partition est effectuée dans les fichiers correspondants aux différents "oplabel" cités par l'option OL, ceux-ci étant pris dans l'ordre de gauche à droite. Les lignes de programmes contenues dans la partition recherchée peuvent également contenir des commandes COPY. Le niveau d'imbrication est limité à 9.
Les partitions doivent être sous forme compressée.

12.2. Options du compilateur

Le compilateur LP80 accepte différentes options qui permettent d'effectuer un traitement approprié aux cas particuliers des utilisateurs.
Il existe 14 options possibles :

SI	(Source Input) entrée image de carte
CI	(Compress Input) entrée fichier compressé
EI	(Edit Input) entrée fichier éditeur
CO	(Compress Output) sortie fichier compressé
EO	(Edit Output) sortie fichier éditeur
GO	Sortie de binaire objet
LS	Sortie liste du programme
CN	Sortie références croisées (réduite)
CC	Sortie références croisées (complète)
OL	Définition des bibliothèques
NP	Suppression de la protection
DB	Demande de mise au point
PT	Demande de zone "patch"
TR	Trace de l'automate

12.2.1. Description des options

SI : Cette option indique que le source est constitué d'enregistrements logiques de longueur 80 caractères (généralement des cartes). La lecture est effectuée à travers l'étiquette logique SI.
Utilisée conjointement avec l'option CI, cette option indique alors une mise à jour (voir 12.2.2.).

CI (nom de partition)

Cette option indique que l'entrée est faite à partir d'un fichier condensé. La lecture est effectuée à travers l'étiquette CI. Si un nom de partition entre parenthèses suit l'option CI, cela indique que l'entrée est constituée d'une partition.

EI

Cette option indique que l'entrée est faite à partir d'un fichier séquentiel indexé (éditeur SIRIS8). La lecture est effectuée à travers l'étiquette EI.

CO (nom de partition)

Cette option indique qu'une sortie en condensé du source sera effectuée à travers l'étiquette CO. Si un nom de partition entre parenthèses suit l'option CO, cela indique que la sortie doit se faire dans une bibliothèque. Si la partition est déjà présente dans la bibliothèque, elle est préalablement effacée.

EO

Cette option indique qu'une sortie du source selon un format compatible avec l'éditeur SIRIS8 sera effectuée à travers l'étiquette EO.

GO (nom de partition)

Cette option demande la génération de code objet. Le module constitué est sorti à travers l'étiquette GO. Si un nom de partition entre parenthèses suit l'option GO, cela indique que la sortie doit se faire dans une bibliothèque. Si la partition est déjà présente dans la bibliothèque, elle est préalablement effacée.

LS Cette option demande une liste de compilation qui sera sortie à travers l'étiquette LO.

CN Cette option demande la sortie des références croisées.
La liste est sortie à travers l'étiquette logique LO.
Seuls les identifieurs référencés apparaissent dans la liste.

CC Cette option est identique à l'option CN sauf que tous les identifieurs référencés ou non apparaissent dans la liste.

OL (op11 [,op12 [,op13 [,op14]]])
Cette option est utilisée en liaison avec les commandes /COPY. La ou les étiquettes logiques indiquées (4 au maximum) désignent les bibliothèques dans lesquelles les partitions seront cherchées.

NP Cette option indique que le code généré par le compilateur ne sera pas mis en protection écriture. Cette option doit être utilisée lorsque le metteur au point (§11.3) est employé.

DB Cette option indique que les demandes d'ordre de mise au point (SNAP, TRACE) doivent être prises en compte. De plus, si le segment compilé est en MAIN, un appel implicite d'initialisation du metteur au point sera généré.

PT Cette option permet de générer en fin de segment une zone de 100 mots qui prendra pour nom externe le nom du segment procédure préfixé par la chaîne $PATCHS. Le déplacement par rapport à la base programme est indiqué sur la liste en fin de compilation. Cela permet d'obtenir une zone basée par la base standard, dans laquelle l'utilisateur pourra éventuellement mettre des "patchs" (processeur DEBUG SIRIS8).

TR Cette option permet de tracer l'automate du compilateur.
Cette option n'est utilisée que pour la maintenance du compilateur.

NOTA BENE : L'option EI est exclusive des options SI et/ou CI.

12.2.2. Mise à jour d'un programme condensé

L'emploi conjoint des options SI et CI indique que le symbolique en entrée est constitué du fichier condensé lu à travers l'étiquette logique CI, mis à jour au moyen de cartes lues.
L'usage de l'option CO permet à l'utilisateur de produire un fichier condensé d'un programme symbolique qui peut être utilisé en entrée dans des assemblages ultérieurs. Ci-dessous est décrite la manière de mettre à jour un fichier condensé grâce à un paquet de mise à jour. Un paquet de mise à jour est compris entre la première carte commençant par un + en colonne 1 et la carte EOD. Si des cartes-source précèdent la première "carte+", ces cartes-source sont considérées comme précédées de la carte +0, ce qui veut dire que ces cartes sont insérées dans la première ligne du fichier condensé.

LP80 reconnait deux cartes de contrôle de mise à jour :

- +k où k indique que toutes les cartes-source qui suivent la carte +k jusqu'à, mais non incluse, la prochaine carte de contrôle de mise à jour, doivent être insérées après la kième ligne du programme formant le fichier condensé. La commande +0 indique une insertion avant la 1ère ligne du programme.

- +j,k où j et k sont les numéros de ligne du listing d'assemblage produit avec le fichier compressé. On a j < k. Cette carte indique que toutes les cartes-source suivant cette carte jusqu'à, mais non incluse, la prochaine carte de contrôle de mise à jour, remplacent les lignes j à k du programme condensé.

Le nombre de lignes insérées n'est pas obligatoirement égal au nombre des lignes supprimées.
Le nombre des lignes insérées peut être nul. Dans ce cas, les lignes j à k du programme condensé sont supprimées.
La fin de fichier sur SI indique la fin des mises à jour du programme.
Le caractère + de chaque carte de contrôle doit être en colonne 1 suivi immédiatement des informations de contrôle dans lesquelles le blanc est interdit.
Les informations sont considérées comme terminées au premier blanc rencontré.
Optionnellement, des commentaires peuvent suivre le ou les blancs.

12.3. Listes fournies

Deux listes peuvent être obtenues lors d'une compilation LP80 :
- liste du source
- liste des références croisées.

12.3.1. Liste du source

Les enregistrements d'un programme LP80 sont listés sans transcodage et avant analyse.
La ligne d'impression comporte six champs.
- position 4 à 12 : numérotation des lignes sous 3 formes possibles.
  (1) NNNNN : numéro de séquence pour les lignes en provenance de SI ou CI
  (2) NNNN:NNN : clé de l'enregistrement du fichier éditeur en provenance de EI
  (3) PP NNNN : lors de l'impression d'une ligne en provenance d'une partition PP indique la profondeur, et NNNN le numéro de ligne à l'intérieur de la partition.

  Si une option CO et/ou EO est précisée, il y a alors renumérotation séquentielle du type (1) ou (2).

- positions 14 à 17 : déplacement relatif courant de la première place libre dans la zone de données. Ceci permet d'associer les variables et leurs adresses.

- positions 20 à 23 : déplacement relatif courant de la prochaine instruction engendrée dans la zone du code. Compteur ordinal du programme.

- positions 26 à 27 : profondeur courante. Cela permet de repérer aisément la structure de bloc.

- position 29 : le caractère 'c' si la ligne source est un commentaire.

- positions 33 à 132 : les 100 premiers caractères de la ligne source.

12.3.2. Liste de références croisées

Les options CN et CC permettent d'obtenir en fin de compilation les références croisées d'un programme LP80.
La liste alphabétique précise pour chaque identifieur un certain nombre de renseignements sur au moins trois lignes :
- ligne 1 : le nom de l'identifieur
- ligne 2 : dans l'ordre, les renseignements suivants :
  - . le numéro de ligne où l'identifieur a été déclaré
  - . la profondeur de la déclaration
  - . le déplacement et la base correspondant à l'identifieur
  - . le déplacement exprimé dans le type de la variable
  - . l'information ACCES EN LECTURE si l'identifieur n'a jamais été accédé en écriture.
- ligne 3 et suivantes : les numéros de lignes où l'identifieur a été référencé.

La sortie des références croisées nécessite l'emploi d'espace disque temporaire.

13. TRAITEMENT DES ERREURS

Plusieurs types d'erreurs peuvent être détectés par le compilateur LP80.

- erreur sur les options
- erreur de gestion des fichiers
- erreurs dûes aux limitations internes du compilateur
- erreurs de syntaxe
- erreurs de sémantique

13.1. Erreur sur les options

Les erreurs dans les options entrainent l'arrêt de la compilation.
Un message d'erreur est imprimé en dessous de la chaîne d'option. Une étoile est imprimée sous le caractère qui provoque l'erreur.

PARENTHESE ABSENTE
Aprés l'option OL, ou à la fin de l'indication d'une partition, une parenthèse est obligatoire.

OPTION INCONNUE
L'option indiquée est inconnue.

PLUS DE 4 LIBRAIRIES
Dans l'option OL on ne peut spécifier que 4 bibliothèques au maximum.

ERREUR CHAINE OPTION
Double définition d'option ou de partition.

NOM DE CLE INVALIDE
La clé de partition est invalide.

LONGUEUR CLE > 13
La clé de partition est trop longue.

OPLABEL INVALIDE
Dans l'option OL l'OPL indiqué est invalide.

SEPARATEUR ABSENT
Les options doivent être séparées par des virgules.

INCOMPATIBILITE OPTIONS
Les options spécifiées sont incompatibles (CI/SI et FI).

13.2. Erreur de gestion des fichiers

Ces erreurs n'entrainent pas toujours la fin de la compilation, mais conduisent généralement à un traitement incorrect. Un message d'erreur est imprimé.

FIN DE FICHIER CI
Dans une demande de mise à jour, la fin du fichier CI est atteinte avant la fin de SI.

ERREUR COMMANDE M.A.J.
Un ordre de mise à jour est incompréhensible.

ERREUR SEQUENCE M.A.J.
Une demande de positionnement sur un enregistrement précédent ne peut être éffectuée.

FICHIER NON COMPRESSE
Le fichier indiqué sur l'étiquette CI ou OL n'est pas compressé.

ERREUR SUR FICHIER CI
Erreur de lecture sur le fichier compressé.

ERREUR FICHIER CO
Erreur d'écriture sur le fichier compressé.

** ERREUR SUR LE FICHIER :
Le fichier indiqué n'est pas accessible ou provoque une erreur irrécupérable.

13.3. Erreurs dûes au compilateur

Ces erreurs sont dûes soit à une limitation interne de tables du compilateur, soit à une erreur interne du compilateur.
Ces erreurs entrainent l'arrêt de la compilation.
Un message d'erreur est imprimé suivi d'une étoile positionnée sous l'unité-syntaxique qui provoque l'erreur.

MEMOIRE INSUFFISANTE
Le compilateur ne dispose pas d'assez de mémoire.
Relancer la compilation avec une partition supérieure.

DEBORDEMENT PILE
La pile de l'automate est trop petite.
Voir la maintenance.

SOUS DEPASSEMENT PILE
Erreur de l'automate. Voir la maintenance.

*** LP80 ERREUR ***
Une erreur grave s'est produite dans le compilateur.
Voir la maintenance.

3.4. Erreur de syntaxe

Ces erreurs sont dûes à un mauvais emploi du langage LP80.

Un message est imprimé qui indique :

- l'état origine dans l'automate
- le numéro de l'unité syntaxique courante

Une étoile est imprimée sous l'unité syntaxique qui a provoqué l'erreur.

SYNTAXE: ETAT US
: Erreur de syntaxe. Le numéro de l'état courant et de l'unité syntaxique courante est donné.

13.5. Erreur de sémantique

Ces erreurs sont dûes à un mauvais emploi du langage.

Une limitation sémantique n'a pas été respectée par l'utilisateur.

Un message est imprimé ainsi qu'une étoile sous l'unité syntaxique qui provoque l'erreur.

SOUS-DEPASSEMENT PROF.
: Une demande de fermeture de bloc (END) de trop a été formulée.

UNITE SYNTAXIQUE
: L'unité syntaxique courante est trop grande. (identifieur trop long)

NOMBRE TROP GRAND
: La valeur du nombre ne tient pas dans un double-mot.

TROP DE DEF. & REF.
: Il y a trop de références et de définitions externes.

REF. AVANT TROP GRANDE
: Il y a trop de références avant.

ETIQUETTE INCONNUE
: L'étiquette n'est pas déclarée.

ETIQUETTE MULTIDEFINIE
: L'étiquette a déja été définie.

PROCEDURE FIN INVALIDE
: On ne peut pas sortir du bloc par un EXIT.

INST. CASE DEBORDEMENT
: Le nombre d'instructions du CASE a déja été atteint.

INST. CASE SOUS DEPASS
Le nombre d'instructions du CASE est inférieur au nombre déclaré.

INSTRUCTION INVALIDE
Il n'existe pas d'instruction machine à même de réaliser le traitement demandé.

ERREUR INITIALISATION
L'initialisation ne correspond pas au type ou au nombre d'éléments de la variable.

DEBORDEMENT TRACE
Il y a trop d'identifieurs tracés.

ERREUR DE PARAMETRE
Dans le traitement des macro-instructions, le paramètre indiqué est invalide.

TYPE INVALIDE
Le type de la variable ne permet pas d'être synonyme d'un registre.

ADRESSE IMPAIRE
Une variable de type LONG ne peut être synonyme que d'une adresse paire.

ADRESSE INVALIDE
La valeur d'une adresse absolue ou d'un déplacement doit être inférieure a x3FFF.

BASE INVALIDE
La base d'adressage doit être supérieure a 0.

INDEX INVALIDE
Un registre d'index doit être compris entre 1 et 7.

REGISTRE INVALIDE
Dans les instructions sur les chaines, le registre indiqué doit être pair.

INST. SILENCE INVALIDE
On ne peut demander le SILENCE d'une instruction non tracée.

BASE DEJA UTILISEE
La base spécifiée est déja employée pour l'adressage d'un autre bloc.

PROTECTION SANS OBJET
La demande de protection n'a aucun sens pour des données fictives.

EXPRESSION INVALIDE

Dans une expresion arithmétique entière (déclaration de constante ou instruction GEN) la valeur obtenue est trop grande.

** ERREUR IRRECUPERABLE

La compilation s'arrete à la suite de l'erreur précédente.

ETIQUETTE NON DEFINIE:

L'étiquette indiquée a été déclarée, référencée mais non définie.

CODE-CONDITION INVALIDE

Dans une déclaration de condition, la valeur doit être inférieure ou égale a 15.

NOM DE PARTITION ABSENT

Dans un ordre /COPY le nom de la partition n'est pas indiqué.

PARTITION NON TROUVEE

La partition indiquée dans un ordre /COPY n'existe pas dans les bibliothèques précisées par l'option OL.

OPTION NP NON VALABLE

La déclaration d'un DCB a pour effet de supprimer l'option NP si celle-ci a été demandée.

CHAINE TROP LONGUE

La chaine de caractères est trop longue.

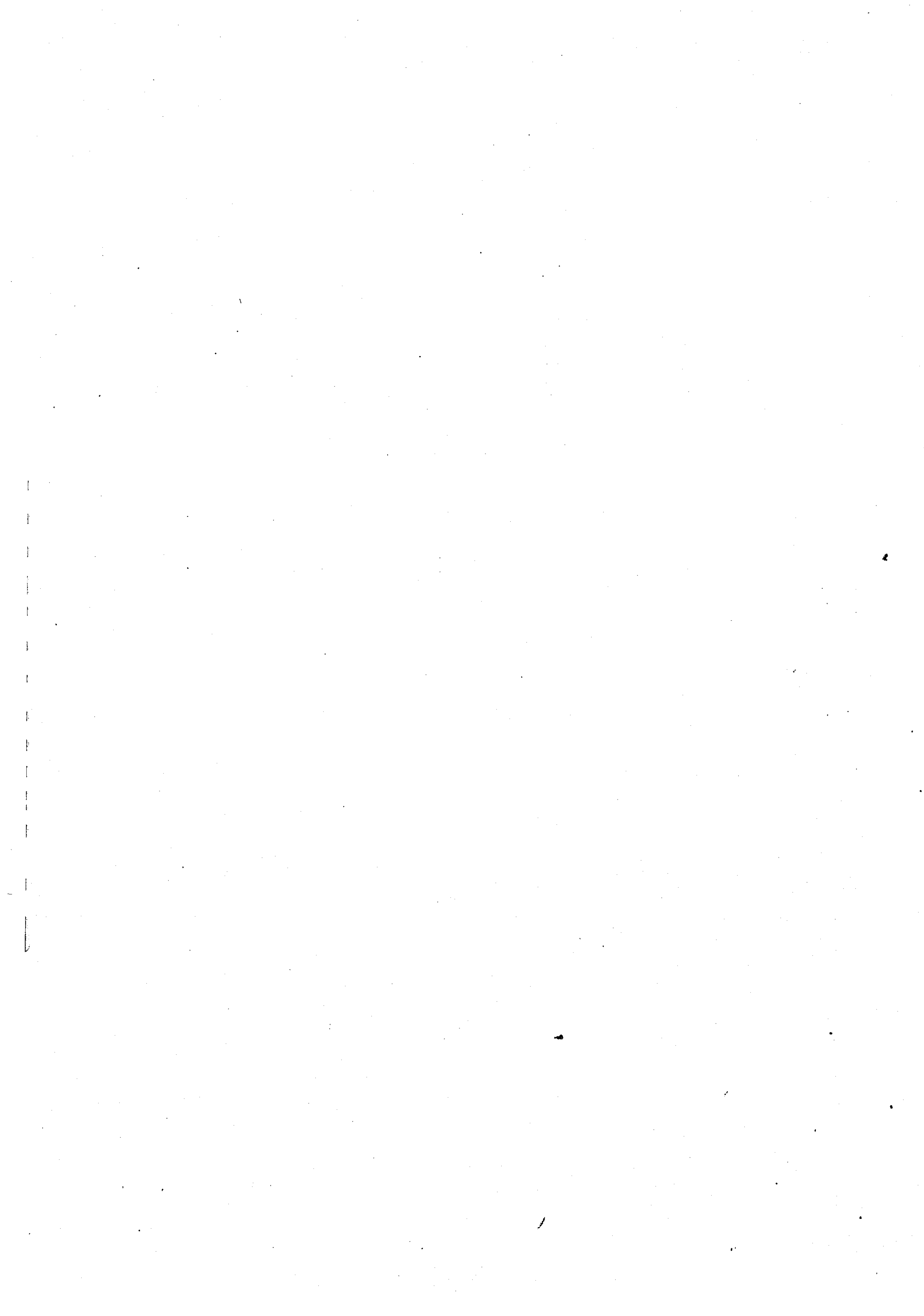
A chaque erreur est associée une valeur de niveau d'erreur.

Ces valeurs sont : #3 #7 #F.

Une erreur de niveau 3 correspond à une erreur peu importante que le compilateur a pu corriger.

Les autres erreurs ne permettent pas une exécution correcte.

A la fin de la liste de compilation, le nombre d'erreurs et le niveau de sévérité sont indiqués.



```
+-------------------------------------+
!                                     !
!      MANUEL D'IMPLEMENTATION        !
!                                     !
!  LANGAGE LP80 - METTEUR AU POINT    !
!                                     !
+-------------------------------------+
```

COMPILATEUR LP80

1. Structure générale 139
2. Les procédures de service 141
 1. L'interface système 141
  1. Gestion de la mémoire centrale 141
  2. Gestion des fichiers 142
 2. Le contrôleur général 143
  1. Rappels sur le module COMMUN 143
  2. Fonction LIRECARTE 144
 3. L'éditeur de références croisées 144
3. Le compilateur LP80 146
 1. Structure générale 146
 2. Analyse lexicographique 147
  1. Rôle 147
  2. Unités syntaxiques 147
  3. Description 148
   1. Automate de l'analyseur 148
   2. Fonctions particulières 150
 3. Gestion de la table des identifieurs 152
  1. Généralités 152
  2. Structure de la table 152
  3. Fonctions de gestion 156
   1. Recherche d'un identifieur 156
   2. Introduction d'un identifieur 156
   3. Effacement d'un identifieur 156
 4. Analyse syntaxique 157
  1. Description de la syntaxe par un automate à pile 157
  2. Description externe de la syntaxe 157
   1. Description du méta-langage 157
   2. Exemple 158
  3. Structure interne de la table des automates 159
  4. Description du mécanisme 162
 5. Fonctions sémantiques 163
  1. Paramètres généraux des fonctions sémantiques 164
  2. Sémantique des données 164

1. Structure du code objet 164
2. Implémentation de la sémantique des données 165
3. Sémantique des déclarations particulières 167
1. Déclaration des constantes 167
2. Déclaration des procédures 168
3. Déclaration des fonctions assembleurs 168
4. Déclaration des conditions 168
5. Déclaration des registres et des bases 169
4. Sémantique des liaisons externes 169
1. Traitement du segment programme 169
2. Traitement des procédures externes 170
3. Traitement des blocs de données 170
4. Gestion des références externes 170
5. Sémantique des instructions du langage 171
1. Structure du code objet 172
2. Traitement des étiquettes 172
3. Traitement des affectations 175
4. Evaluation des expressions conditionnelles 176
5. Evaluation des expressions arithmétiques 178
6. Traitement des déclarations implicites 180
6. Mécanisme de TRACE 181
7. Fonctions sémantiques générales 184

ACCOMPAGNATEUR LP80

1. Mécanisme de chargement dynamique 187
2. Interface Système 193
3. Fonctions de traitement 198
1. Buts 198
2. Principes généraux 198
3. Traitement des ordres TRACE 199
4. Traitement des ordres SNAP 200
5. Traitement des erreurs programme 201
6. Fonctions de mise au point 202

COMPILATEUR LP80

1. Structure générale

Le compilateur LP80 présente une structure modulaire permettant de répondre aux impératifs initiaux:

- indépendance système
- réentrance possible

Il se compose de 4 modules ayant chacun des fonctions spécifiques liées à leur dépendance plus ou moins grande du système hôte.
Dans la mesure où le compilateur serait à transporter sur un autre système, un nombre réduit de modifications seraient à apporter.
L'interface système serait à réécrire entièrement tandis que le contrôleur pourrait être conservé dans sa forme actuelle si le Sytème de Gestion de Fichiers était de nature semblable à celui de SIRIS8.
Les 4 modules du compilateur sont:

- L'interface système assure toutes les fonctions systèmes nécessaires au compilateur (gestion mémoire , gestion de fichiers). C'est le seul module réellement système dépendant.
  Il comporte environ 250 lignes source.
- Le contrôleur général assure trois fonctions distinctes:
  + l'analyse des options de compilation
  + le séquencement des compilations enchainées.
  + la gestion des fichiers sources selon le standard de SIRIS8 (mise à jour , compactage , décompactage)

  Seule cette dernière fonction est en partie dépendante de l'organisation des fichiers.
  Le module comporte environ 500 lignes source.
- L'éditeur de références croisées assure le tri puis

l'édition des identifieurs d'un programme compilé.
Il comporte environ 350 lignes sources
- Le compilateur enfin, assure l'analyse syntaxique et les fonctions sémantiques correspondantes. Il produit du code objet utilisable par l'éditeur de liens de SIRIS8.
Il comporte environ 7000 lignes source.

Ces 4 modules ne comportent aucune déclaration de variables. Celles ci sont déclarées dans 3 partitions de bibliothèque, selon les critères suivants:

- la table d'identifieurs standards et l'automate (qui sont générés par programme)
- les constantes (messages d'erreur , ...)
- les variables

Enfin, le système de mise au point est entièrement indépendant du compilateur.
Il est écrit en deux parties.
La première assure l'interfaçage avec le système hôte. La deuxième partie permet les différentes fonctions de mise au point.
Le metteur au point peut être employé pour la mise au point de programmes écrits en d'autres langages que LP80.

2. Les procédures de service

2.1. L'interface système

L'interface système a pour but de rendre indépendant le compilateur des problèmes liés à la demande de services systèmes (appels superviseur).
Ces services sont de deux catégories trés inégales quant à leur importance:

- la gestion de la mémoire centrale
- la gestion des fichiers

2.1.1. Gestion de la mémoire centrale

Le compilateur gère lui-même les recouvrements de variables et l'implantation de celles-ci dans la mémoire disponible.
La seule demande explicite de mémoire au système est faite par l'interface lors de l'initialisation au moyen de la macro-instruction :GP (Get Page).
La demande est volontairement faite dans le segment 0 de la mémoire virtuelle, car il n'est pas possible de procéder à une demande :GPC dans les autres segments (8 et au delaà).
En effet le système en temps partagé SIRIS8 ne permet pas l'adressage étendu (version C09), et la limitation imposée n'est pas importante.
Lors du lancement du compilateur, SIRIS8 fournit dans le registre R1 le nombre de pages disponibles en zone dynamique. Aprés déduction des pages nécessaires au SGF (buffers système) l'ensemble de l'espace est demandé.
Cet espace est remis a zéro afin d'obtenir:

- un état initial des variables
- une économie de papier et une lisibilité plus grande pour la sortie des "dumps" lors de la phase de mise au point du compilateur.

L'espace obtenu sera libéré en fin de compilation au moyen

d'un appel explicite :FP (Free Page).
La chaine d'options, dont l'adresse est fournie dans le registre R2 est recopiée dans une zone prédéterminée (UNTSTX) afin de se ramener à un cas standard quel que soit le système.

2.1.2. Gestion des fichiers

Afin de se dégager le plus possible des contraintes du sytème, chaque fichier potentiellement utilisable reçoit un numéro d'identification.
Toute demande concernant un fichier comporte donc ce numéro.
Ces demandes sont de 3 types:

- initialisation (OPEN)
- échanges (GET ou PUT)
- fin d'échange (CLOSE)

Elles sont satisfaites par des fonctions de l'interface qui travaillent toutes de manière similaire au moyen de paramètres standards:

- R0 : numéro de voie à l'appel
  code d'erreur au retour
- R1 : adresse octet de la zone mémoire concernée
- R2 : longueur de la zone en octets

Quelle que soit l'organisation du fichier et les subtilités (oh combien peu évidentes !) du SGF de SIRIS8 l'interface permet de se ramener à un protocole d'échange simplifié.

Le point d'entrée COPYF assure la gestion des partitions de bibliothèque appelées par des ordres /COPY. Ces ordres ne s'appliquent qu'à des partitions de langage source compressé.
L'interface tolère une profondeur d'imbrication de 9 niveaux. Une redéfinition de constante permet d'augmenter ou de diminuer ce nombre.
COPYF assure la sauvegarde du fichier courant puis la recherche de la partition demandée. Si la recherche n'est

pas satisfaite, un code d'erreur est renvoyé.
Le paramètre d'entrée est l'adresse octet de la chaine de caractères donnant le nom de la partition. (registre R0).

2.2. Le contrôleur général

Comme cela a été dit précedemment, il permet l'analyse des options, l'enchainement des compilations et assure la gestion des fichiers source.
Les deux premiers points ne présentent aucune particularité.
Notons seulement qu'il a été jugé préférable d'arrêter le traitement lors de la rencontre d'une erreur dans la chaine d'options.
Par contre, une erreur irrécupérable dans la compilation d'un premier programme n'empêche pas la compilation des programmes suivants.

2.2.1. Rappels sur le module COMMUN de SIRIS8

Ce module qui est théoriquement employé et employable par tous les compilateurs implémentés sur le système, assure différentes fonctions. Il devrait présenter une interface simple et tendre vers une standardisation des services possibles.
Il n'en est rien et l'écriture peu structurée de ce composant profondément modifié depuis sa première élaboration sans doute fort ancienne, ne permet pas de considérer son emploi comme facile et souhaible.
Ecrit en mode B, il ne peut pas être employé facilement en mode C.
Le principal intérêt pour l'utilisateur est le mécanisme de mise à jour des fichiers source. Il consiste à permettre la correction simple de fichiers de type compressé au moyen de commandes de mise à jour (Voir manuel utilisation).
Ce mécanisme, trés souple, a été implémenté dans le contrôleur général de LP80, ce qui permet de se passer

totalement du module commun "standard" tout en assurant les mêmes services.

2.2.2. La fonction LIRECARTE

Cette fonction renvoie dans l'emplacement mémoire BUFFER un enregistrement de langage source quels que soient les fichiers concernés. Cet enregistrement peut éventuellement résulter de la mise en oeuvre du mécanisme de mise à jour.
Les enregistrements sont toujours considérés de longueur variable et la longueur est renvoyée dans la variable LNGBUFFER. La longueur doit être comprise entre 0 et 255, mais seuls les 100 premiers caractères apparaîtront sur la liste de compilation élaborée par le compilateur.
La ligne source provient soit d'un fichier de type éditeur, soit de la mise à jour d'un fichier compressé. Si une demande de lecture en bibliothèque a été faite, la ligne proviendra alors de la partition concernée. Si l'utilisateur en a fait la demande, les lignes qui ne sont pas issues d'une bibliothèque sont rangées dans un fichier éditeur et/ou compressé.

2.3. L'éditeur de références croisées

L'éditeur travaille à partir de deux fichiers qui sont construits durant la compilation. Le premier fichier (défini par la constante FILECRF1) contient des enregistrements de longueur variable.
Chaque enregistrement contient des blocs de longueur variable qui définissent chacun des identifieurs.
Les informations qui figurent dans ce bloc sont les suivantes:

- un pointeur d'accès au fichier des références
- le type et le sous-type de l'identifieur
- le numéro de la ligne source dans laquelle

l'identifieur a été déclaré

- la profondeur de déclaration
- la base associée à l'identifieur
- le déplacement par rapport à la base
- la chaine de caractères de l'identifieur

Dans une première phase les blocs sont chargés en mémoire centrale. Ils sont ordonnés par ordre alphanumérique croissant en utilisant la technique d'un arbre binaire.
Dans une deuxième phase, l'arbre est parcouru, et chaque identificur est imprimé. L'ensemble des références est ensuite obtenu dans le fichier de numéro FILECRF2, puis ordonné par numéro de ligne croissant, et enfin imprimé.

3. Le compilateur LP80

3.1. Structure générale

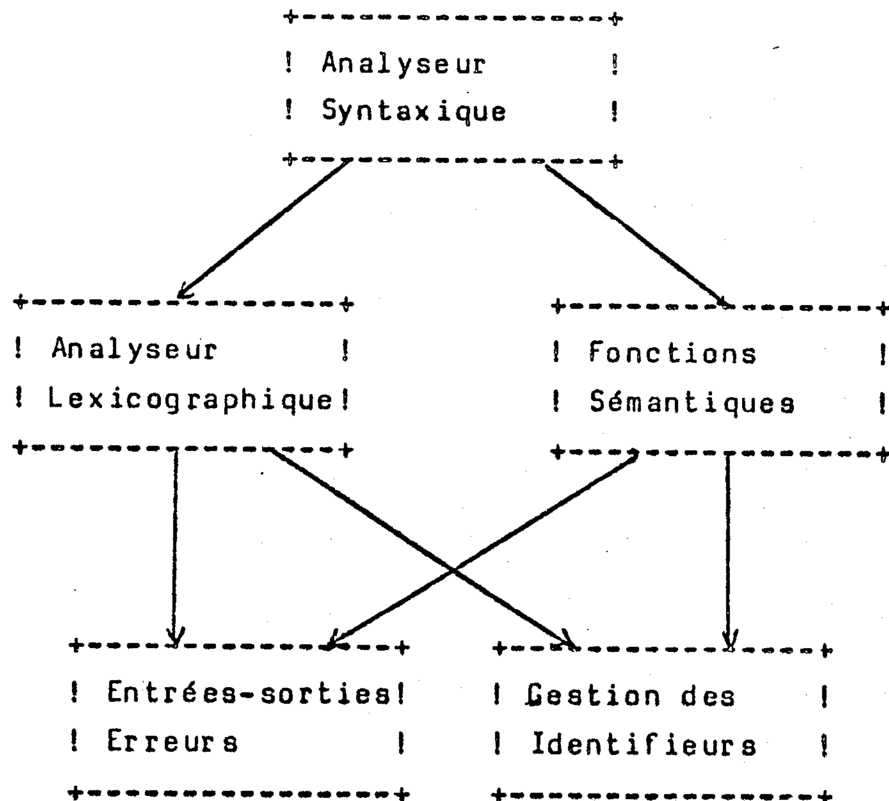
L'ensemble des procédures qui composent le module COMPILATION est organisé autour de l'analyseur syntaxique. Celui-ci fait appel à deux groupes de fonctions:

- celles de l'analyseur lexicographique
- les fonctions sémantiques

Ces fonctions utilisent elles-même deux autres groupes de procédures:

- les procédures de gestion de la table des identifieurs
- les procédures d'entrée-sortie et de traitement des erreurs

La structure peut donc être schématisée ainsi:



La description du compilateur s'articule donc autour de ces différentes parties.

3.2. L'analyseur lexicographique

3.2.1. Rôle

L'analyseur lexicographique a pour but de découper la succession des éléments symboliques qui proviennent de la lecture d'un enregistrement en UNITES SYNTAXIQUES.

3.2.2. Les unités syntaxiques

L'analyseur lexicographique reconnait les unites suivantes:

- les identifieurs, suite de caractères alphanumériques commençant par un caractère alphabétique et pouvant comporter les caractères _ (souligné) ou ' (apostrophe).
- les symboles qui sont groupés en trois classes:
  + les symboles simples
  + les symboles doubles
  + les symboles triples
- les nombres décimaux, suite de caractères numériques terminée éventuellement par le caractère L qui indique alors qu'il s'agit d'une valeur longue (double-mot).
- les nombres hexadécimaux, suite de caractères commençant par le caractère £ (dièse) suivi de caractères numériques ou des lettres A,B,C,D,E,F,. Ils peuvent être terminés par le caractère L qui indique qu'il s'agit d'une valeur longue.
- les nombres flottants , composés d'une mantisse suivie éventuellement d'un exposant, exprimés par un nombre décimal, et pouvant se terminer par la lettre L afin d'indiquer qu'il s'agit d'une valeur longue.
- les chaines, suite de caractères quelconques, commençant et se terminant par le caractère " (guillemet). Si ce caractère doit figurer dans la chaine, il doit être doublé.

3.2.3. Description

L'analyseur se compose de deux parties:
La première permet d'éliminer les blancs et les commentaires et de reconnaître les unités syntaxiques de base, à savoir:

- les identifieurs
- les nombres entiers
- les nombres flottants
- les chaines

La deuxième partie n'est exécutée que dans le cas où l'unité syntaxique de base est un identifieur.
Dans ce cas, un appel à la procédure RCHID permet de déterminer si cet identifieur est standard, déclaré ou non.
La première partie est construite au moyen d'un automate simple.

3.2.3.1. L'automate de l'analyseur lexicographique

Cet automate est décrit par une matrice de transition de 256 octets; il comporte en effet 16 états numérotés de 0 à F et 16 classes de symboles de base.
La méthode employée est coûteuse en emplacements mémoire puisqu'elle prend en tout:

Nombre d'états * Nombre de classes = Nombre d'éléments

C'est cependant la méthode la plus rapide qui à un instant donné, détermine l'état d'arrivée en un seul coup, ceci sans mécanisme itératif. Elle est donc parfaitement adaptée à un analyseur lexicographique, puisque chaque caractère en entrée transite par ce mécanisme qui doit être rapide, et que de plus le nombre de classes est réduit.
Chaque caractère dans la chaine d'entrée est traduit au moyen d'une table en une valeur comprise entre 0 et 15.

Les classes déterminées sont les suivantes:

0 caractères invalides, blancs et guillemets
1 marqueur de fin de ligne (xFF)
2 lettres A B C D F
3 lettres G à Z sauf L
4 lettre E
5 lettre L
6 chiffres 0 à 9
7 £ (dièse)
8 .
9 + -
A :
B =
C < >
D autres symboles de base
E _ (souligné) et ' (apostrophe)
F " (guillemets)

Le mécanisme de l'automate est trés simple. Lors de chaque appel de l'analyseur, l'état courant est mis à zéro. Un caractère est obtenu, qui par traduction donne le numéro de classe. On accède alors à un élément de la matrice, l'état courant donnant le numéro de colonne et le numéro de classe déterminant le numéro de ligne.
Chaque élément est un octet dont la signification est la suivante:

- les quatre premiers bits donnent le numéro de l'état suivant
- les quatre bits suivants donnent le numéro de l'action sémantique à entreprendre

Dans le cadre précis du problème posé, on obtient donc:

- la rapidité maximum
- une occupation mémoire réduite

Les limitations du mécanisme sont évidentes: impossibilité d'avoir plus de 16 états, 16 classes et 16 actions sémantiques.

3.2.3.2. Fonctions sémantiques spéciales

Deux fonctions sémantiques de l'analyseur sont un peu particulières par les traitements qu'elles opèrent. Quelques explications s'imposent donc:

1 - Lecture des lignes sources.
La rencontre d'un caractère marqueur de fin de ligne entraine l'appel de la fonction externe LIRECARTE. Une ligne source est obtenue, et un caractère marqueur de fin de ligne est introduit en fin. Le premier caractère est testé afin de déterminer si une commande au compilateur est demandée. Si c'est le cas, le traitement correspondant est effectué, en particulier pour l'ordre /COPY où un appel à la procédure externe COPYF permet d'obtenir la partition demandée.
Si l'impression du source est demandée, la ligne est préparée en tenant compte des contraintes liées à l'origine et son incidence sur la numérotation.

2 - Traitement des commentaires et des espaces.
Quoique l'automate soit rapide, il était intéressant d'accélérer le parcours des caractères non syntaxiquement utiles (blancs et commentaires).
Au lieu de parcourir la chaine en entrée caractère par caractère, un emploi judicieux de l'instruction TTBS associé à une table de traduction bien adaptée permet d'optimiser le traitement.
Rappels sur l'instruction TTBS:

Comme toutes les instructions de traitement des chaines en mode C, l'instruction TTBS nécessite l'emploi de la base 0 et d'un couple de registres. La base contient la longueur de la chaine à tester et en poids fort (bits 0-7) un masque. Le registre pair contient l'adresse de la table et le registre impair l'adresse de la chaine à tester. Apres le lancement de l'instruction, chaque caractère de la chaine

est examiné jusqu'à ce que le produit logique de l'entrée courante de la table avec le masque de la base soit différent de zéro. L'instruction s'arrète alors et le registre impair pointe sur le dernier caractère testé.
Dans le cas présent, la table de transcodage est la même que celle employée pour déterminer les classes. Tous les caractères ont pour correspondance une valeur comprise entre 0 et 15 sauf le marqueur de fin de ligne (xF1) et le caractère (perluet) (xF0).
Le masque associé à l'instruction TTBS aura pour valeur:

xF0 si on veut sauter tous les caractères sauf fin de ligne et perluet

x0F si on veut sauter les caractères blancs seulement

Après qu'un début de commentaire ait été décodé, le masque xF0 est employé jusqu'à la rencontre d'un caractère perluet marquant la fin de commentaire.

3.3. Gestion de la table des identifieurs

3.3.1. Généralités

Tous les identifieurs standards et déclarés figurent dans une table unique qui se compose d'un ensemble de blocs descripteurs chainés entre eux. Chaque bloc descripteur est de longueur variable suivant la longueur de l'identifieur. Cette table est une image de la structure de blocs du langage. Lorsque l'on sort d'un bloc tous les identifieurs déclarés dans ce bloc sont effacés. Seuls les mots clés du langage (appelés identifieurs standards) sont présents durant toute la compilation car ils appartiennent à un bloc englobant tous les autres. En fait les identifieurs standards sont répartis dans deux niveaux:
- le niveau le plus englobant qui contient les mots clés du langage proprement dit
- un niveau supérieur qui contient les mots clés des macro-instructions SIRIS8

Le mécanisme de recherche tient compte de cette particularité, ce qui permet de ne pas considérer les mots clés des macro-instructions comme des mots réservés.

3.3.2. Structure de la table des identifieurs

Quatre tables sont employées pour la gestion des identifieurs:

1 - HASH

C'est une table de 256 entrées correpondant aux différents résultats possibles du mécanisme de "hash-coding". Chaque entrée est un pointeur qui permet d'accéder à la tête de liste des identifieurs de même hash-code.

2 - PTSOM

C'est une une table de NBPROF2 éléments permettant de repérer les limites d'implantation des indentifieurs d'une profondeur de bloc donnée. NBPROF2 est une constante de

compilation qui est égale à la constante NBPROF + 2. Cette dernière indique la profondeur maximum possible pour l'utilisateur.

3 - NBPROFELEMENT

C'est une table de NBPROF2 éléments permettant de connaitre le nombre d'identifieurs déclarés dans une profondeur déterminée.

4 - TABLID

C'est la table des identifieurs proprement dite. Elle se compose de deux parties:

- la première partie, statique, non modifiable est la partie qui contient les identifieurs standards (langage et macro-instructions) c'est à dire les profondeurs -1 et -2. Elle est construite par le générateur de tables.
- la deuxième partie est implantée dynamiquement en mémoire libre; elle évolue au fur et à mesure des déclarations faites.

Les deux parties sont construites selon un même schéma, le mécanisme d'accès y est le même, les blocs descripteurs de la première partie sont de taille plus réduite.

Description d'un bloc d'identifieur standard

```
+----------------+--------+--------+
! ptr de chainage ! s/type ! type   !
+----------------+--------+--------+
! mot d'informations diverses      !
+--------+-------------------------+
! long.  !  identifieur standard   !
+--------+ . . . . . . . . . . . . .!
!                                  !
/                                  /
/                                  /
+----------------------------------+
```

Description d'un bloc d'identifieur déclaré

```
+----------------+--------+--------+
! ptr de chainage ! s/type ! type   !
+----------------+--------+--------+
! déplacement     !////////! base   !
+----------------+--------+--------+
! numéro de trace ! num. decl ou ref!
+----------------+--------+--------+
! numéro de ligne ! ptr de X. ref   !
+--------+-------+-----------------+
! long   !  identifieur             !
+--------+ . . . . . . . . . . . . .!
!                                   !
/                                   /
/                                   /
+----------------------------------+
```

Ces deux blocs sont décrits au moyen d'une même DUMMYDATA (IDENTIFIEUR) basée par B5. L'organisation générale des tables est donnée par le schéma ci-dessous.
Les chainages sont relatifs au début de TABLID ce qui implique que la deuxième partie soit située à une adresse supérieure à celle de la première partie, mais que le déplacement maximum soit inférieur à x7FFF, celui-ci étant codé sur un demi-mot.
Cette contrainte qui permet de réduire substantiellement la taille mémoire occupée peut cependant être levée sans problème.

```
     NBPROF2 éléments
+---+---+---+---+---+---+//-+---+---+---+---+---+
!   !   !   !   !   !   !   !   !   !   !   !   !  NBPROFELEMENTS
+---+---+---+---+---+---+//-+---+---+---+---+---+

+---+---+---+---+---+---+//-+---+---+---+---+---+
!   !   !   !   !   !   !   !   !   !   !   !   !  PTSOM
+---+---+---+---+---+---+//-+---+---+---+---+---+
  .   .   .   .
  .   .   .   .               TABLID - partie 1
  .   .   .   .        +-------------------------+
  .   .   .   .        !                         !
  .   .   .   .        !                         !
  .   .   .   .        !                +---+    !
  .   .   .   .        !                !   !    !  Id standard
  .   .   .   .        !                +---+<..!.....
  .   .   .   .        !                         !    .
  ..............>! - - - - - - - - - - - -!    .
      .   .   .        !                         !    .
      .   .   .        !                         !    .
      .   .   .        !                         !    .
      .   .   .        !                         !    .
      .   .   .        !                         !    .
      ...........>+-------------------------+    .
          .   .                                       .
          .   .           TABLID - partie 2           .
          .   .        +-------------------------+    .
          .   .        !                +---+...!.....
          .   .        !                !   !    !  Id profondeur 0
          .   .        !                +---+<..!.....
          ........>!- - - - - - - - - - - -  !    .
              .        !                         !    .
              .        !                +---+...!.....
              .        !                !   !    !  Id profondeur 1
              .        !                +---+<..!..........+-----+  HASH
              .        !                         !           !     !
              .        !                         !           +-----+
              ....>!- - - - - - - - - - - -  !
                       !                         !
                       /                         /
                       /                         /
                       !                         !
                       +-------------------------+
```

3.3.3 Les fonctions de gestion de la table des identifieurs

1 - RCHID

Cette fonction permet de rechercher un identifieur jusqu'à une profonfondeur donnée.

2 - COPID

Cette fonction permet d'introduire dans la table des identifieurs, l'identifieur présent dans la zone UNTSTX (unité syntaxique). Elle reçoit et range en outre le type et le sous-type de l'identifieur. Le numéro courant de la ligne source est introduit dans le bloc descripteur de l'identifieur ainsi que la profondeur courante afin de permettre éventuellement la gestion des références croisées. Les autres entrées sont initialisées de manière standard (généralement zéro).
A la sortie de la procédure, la base B5 pointe sur le bloc descripteur.

3 - NETTOY

Cette fonction assure l'effacement des identifieurs déclarés dans un bloc lors de la fermeture de celui-ci. Pour chaque entrée de la table HASH un parcours des chaines d'identifieurs est effectué jusqu'à ce que le pointeur de chainage indique que le prochain identifieur appartient à un bloc englobant. (consultation de PTSOM)
Lorsqu'un identifieur effacé correspond à une étiquette, il y a vérification de la définition de celle-ci dans le cas où des références ont été effectuées. Une erreur peut éventuellement être détectée et signalée.

3.4. Analyse syntaxique

3.4.1. Description de la syntaxe par un automate à pile

Le langage LP80 est défini par une grammaire "Régular Context Free" dont la propriété principale est de permettre l'analyse de n'importe quelle phrase du langage par un ensemble d'automates d'états finis qui s'appellent récursivement.
L'analyse est du type "top-down".
Le graphe des différents automates (à l'exception des macro-instructions) est donné en annexe 1.

3.4.2. Description externe de la syntaxe

La syntaxe est décrite de manière externe au moyen d'un méta-langage spécialement construit à cet effet. Ce langage, compilé au moyen d'un programme lui-même écrit en LP80, permet de fournir différentes tables ainsi que certaines déclarations de constantes utilisées par le compilateur.
Essentiellement on peut noter qu'il s'agit:
- de la table des identifieurs standards ainsi que des constantes associées à la gestion des identifieurs
- d'une table décrivant la syntaxe sous forme d'un ensemble d'automates.

3.4.2.1. Description du méta-langage

Le langage de description de l'automate est spécialisé pour la génération des tables du compilateur LP80. Toutefois moyennant quelques adaptations sommaires, il pourrait permettre la résolution d'autres problèmes. Le générateur est écrit en LP80. Il travaille en une seule passe avec chaine de reprise pour les références en avant. La description complète de l'automate du langage (automate généré par lui-même) est faite en annexe 5.

On reconnait deux parties:

1 - déclaration des identifieurs standards.

Ils peuvent être déclarés comme symboles terminaux ou bien être regroupés au sein d'une classe (par exemple la classe TYPE comporte les symboles BYTE, SHORT, WORD, etc ...).

La déclaration d'un identifieur permet de lui associer éventuellement une valeur de sous-type (STYPID) comprise entre 0 et 255 et une valeur d'emploi varié (INF1) pouvant tenir sur un mot.

2 - description des automates.

Les différentes transitions possibles (voir 3.4.3) sont décrites simplement au moyen des quatre phrases:

SI .... SINON
APPELER
ALLER_EN
RETOURNER

La simplicité du langage ne nécessite pas d'explications plus approfondies, l'exemple suivant permettant de comprendre .

3.4.2.2. Exemple de description d'automate

Nous donnons ici la description de l'automate DCL5 déclaration de booléens. Cet automate est appelé lorsque l'automate PROGRAMME ou INSTRUCTION rencontre l'unité syntaxique LOGICAL.

```
AUTOMATE   &******************** DCL5    *********
    DCL5
            SEM£16 SI "IDNDECL" ALLER_EN L1 SINON
            SEM£01 RETOURNER
    L1
            SEM£19 SI =          ALLER_EN L2 SINON
                   SI SYN        ALLER_EN L3 SINON
            SEM£18 ALLER_EN L6
    L2
            SEM£A7 SI "VALOGIC" ALLER_EN L6 SINON
            SEM£01 ALLER_EN L6
    L3
            SEM£1D SI "NBENT"   ALLER_EN L6 SINON
            SEM£1E SI "IDVAR"   ALLER_EN L4 SINON
            SEM£1E SI "IDLOGIC" ALLER_EN L4 SINON
            SEM£1E SI "IDPTR"   ALLER_EN L4 SINON
            SEM£1E SI "IPILE"   ALLER_EN L4 SINON
            SEM£01 RETOURNER
    L4
                   APPELER VAR4 ALLER_EN L5
    L5
            SEM£1F ALLER_EN L6
    L6
            SEM£03 SI ,          ALLER_EN DCL5 SINON
                   RETOURNER
```

3.4.2.3. Structure interne de la table des automates

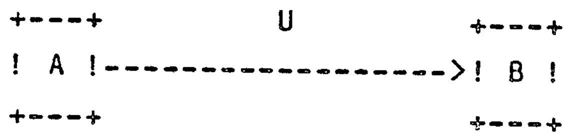
La table décrivant les automates se compose de N éléments où l'on a:

$$N = 1 + (3*T)$$

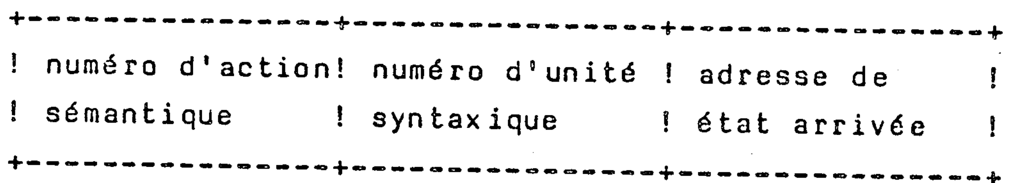
T étant le nombre total de transitions entre états.
Chaque élément est codé sur un demi-mot. Le premier élément de la table est un élément fictif. La description d'une transition nécessite 3 éléments du tableau même si cette transition est un appel d'automate ou une sortie d'automate.

Les trois éléments d'une transition prennent les valeurs suivantes:

1er cas : transition de type conditionnel SI .... SINON



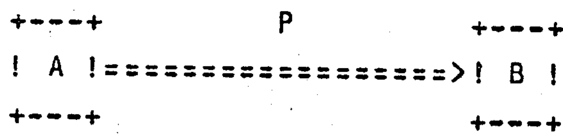
Une progression de l'état A vers l'état B est effectuée si l'unité syntaxique courante est U. Dans ce cas une action sémantique peut être entreprise.



numéro d'action sémantique	numéro d'unité syntaxique	adresse de état arrivée

Le numéro d'unité syntaxique est attribué automatiquement par le générateur. Il a une valeur strictement positive. L'adresse de l'état d'arrivée est un déplacement relatif dans la table des automates. Si aucune action sémantique n'est à entreprendre, le numéro est nul.

2ème cas : transition de type appel d'automate APPELER



Un appel impératif de l'automate P est effectué. Au retour de celui-ci l'état courant sera l'état B. Une action

sémantique peut être entreprise avant l'appel.

numéro d'action sémantique	adresse de l'automate	adresse de état d'arrivée

L'adresse de l'automate est la valeur négative du déplacement relatif de l'automate dans la table.

3ème cas : branchement inconditionnel ALLER_EN

```
+---+                  +---+
! A !----------------->! B !
+---+                  +---+
```

Une transition impérative de A vers B est effectuée. Une action sémantique peut être entreprise.

numéro d'action sémantique	valeur zéro	adresse de état d'arrivée

4ème cas : sortie d'automate RETOURNER

```
+---+
! A !- - - - ->
+---+
```

Un retour vers l'automate appelant est effectué. Une action sémantique peut être entreprise aprés le retour.

numéro d'action sémantique	valeur zéro	valeur zéro

Le générateur construit une partition (compressée) de nom LP80AUTOMATE qui est appelée dans le source du compilateur. Cela permet de modifier éventuellement la syntaxe du langage sans être obligé de modifier le compilateur.
Seule une nouvelle compilation est nécessaire.

3.4.2.4. Description du mécanisme d'analyse

Un pointeur (PTRETAT) permet de connaître l'état courant dans l'automate. Un appel à l'analyseur lexicographique permet d'obtenir une unité syntaxique. Un balayage séquentiel de la table permet de savoir si cette unité est valide à partir de l'état origine. Dans ce cas l'action sémantique est éventuellement entreprise (si la valeur n'est pas nulle) et un branchement vers l'état d'arrivée est effectué. La non reconnaissance d'une unité syntaxique peut entrainer plusieurs types d'action:
- appel d'un automate
- branchement inconditionnel
- sortie de l'automate

Une action sémantique de traitement d'erreur est eventuellement appelée. Dans ce cas l'unité syntaxique non reconnue est généralement sautée et l'état courant est positionné sur l'état qui a le plus de chances possibles de permettre la continuation de l'analyse. Cet état est choisi à priori, de manière statique, comme étant le mieux adapté.
Il n'existe donc pas de mécanisme universel de récupération d'erreurs, mécanisme qui pourrait consister par exemple à rechercher le prochain symbole standard et à déterminer en fonction de critères divers quel pourrait être l'état où

l'analyse doit reprendre. Le faible coût d'une compilation et le fait qu'en tout état de cause le code produit ne serait certainement pas correct, ont conduit à prévoir un mécanisme statique qui répond toutefois aux besoins les plus fréquents.
Par exemple la rencontre d'un identifieur non déclaré alors qu'une variable est attendue conduit à la déclaration d'une variable de type mot et à la continuation de l'analyse.

3.5. Les fonctions sémantiques

Pour chaque transition dans l'automate, il existe un numéro de règle sémantique. Ce numéro peut être nul. Dans ce cas aucune action sémantique ne doit être entreprise. Dans le cas contraire un appel de la fonction SEMANTIQUE est effectué. Cette fonction se compose uniquement d'une instruction CASE sur le numéro d'action sémantique qui lui est fourni en paramètre. On peut distinguer deux groupes d'actions:

- le premier concerne le langage LP80 proprement dit
- le deuxième n'est employé que pour les macro-instructions SIRIS8

Les deux groupes sont nettement séparés. Chaque entrée de l'instruction CASE est commentée afin d'indiquer clairement quel est le but de l'action entreprise. Dans le cas le plus fréquent, des appels à des fonctions sémantiques générales sont effectués.
Nous ne décrivons pas ici les fonctions sémantiques des macro-instructions SIRIS8. Elles ne représentent qu'une faible partie de l'ensemble, et de plus n'offrent aucune difficulté, les fonctions réalisées étant trés simples.
Les fonctions sémantiques du langage peuvent être classées en plusieurs groupes.

3.5.1. Paramètres généraux des fonctions sémantiques

Les paramètres généraux des fonctions sémantiques sont constitués des deux registres de base B4 et B5.
La base B5 permet d'adresser l'entrée dans la table des identifieurs, qui correspond à l'unité syntaxique courante.
Cette entrée est décrite par le bloc fictif IDENTIFIEUR.
La base B4 est utilisée pour accéder au descripteur d'une unité syntaxique précédemment obtenue.
Par exemple:

```
ARRAY 25 SHORT TABLE1,TABLE2;
         ↑          ↑
         B4         B5

R1 := TABLE1 (R3) (4);
      ↑           ↑
      B4          B5
```

3.5.2. Sémantique des données

Les fonctions sémantiques des données permettent de:
- remplir la table des identifieurs
- générer éventuellement le code objet nécessaire afin que le déplacement en mémoire corresponde au déplacement introduit dans la table des identifieurs
- initialiser éventuellement ces variables avec les valeurs indiquées par le programmeur

Le paramètre essentiel de ces fonctions sémantiques est le pointeur PTRDON qui contient en cours de compilation le déplacement (longueur en mots) de la zone des données.

3.5.2.1. Structure du code objet généré

Pour comprendre en détail la structure du code objet généré, il faut se reporter à la brochure CII-HB
NORMES DE PROGRAMMATION - Chapitre 2

qui décrit en détail le langage binaire objet.
Les directives utilisées par la sémantique des données sont les suivantes

- définition d'origine
cette directive indique à l'éditeur de liens l'emplacement à partir duquel la génération de code devra se faire.
- chargement absolu
cette directive permet de charger de 1 à 16 octets avec une valeur déterminée, à partir de l'emplacement courant
- répétition de chargement
cette directive est employée lorsque plusieurs éléments de mémoire doivent être chargés par la même valeur (le cas le plus fréquent est l'initialisation de tableaux)
- chargement avec translation
cette directive est employée lorsqu'une variable doit être initialisée avec l'adresse d'un élément de mémoire.

3.5.2.2. Implémentation de la sémantique des données

Les différentes fonctions employées sont:

COPID
PILEDCL
INCPTRDON
ALIGNDON
GENERDON

1 - Fonction PILEDCL

Cette fonction permet de générer un pile. Les paramètres d'appel sont:

R1 ::= indicateurs de débordement
B4 ::= adresse du bloc d'identifieur de la pile
NEL ::= nombre d'éléments de la pile

Un appel à la fonction ALIGNDON permet de forcer l'alignement sur une frontière de double mot (adresse paire).
Aprés vérification de la taille de la pile (maximum x7FFF)

le descripteur de pile est généré par deux appels à GENERDON. La pile est ensuite générée (NEL mots de valeur nulle) par appel de GENERDON. Le pointeur de données est mis a jour par INCPTRDON.

2 - Fonction INCPTRDON

Cette fonction met à jour le pointeur PTRDON et procède éventuellement à la génération d'octets afin d'assurer la cohérence entre le pointeur et les données.
Les paramètres sont:

NEL ::= nombre d'éléments déclarés
LEL ::= longueur de chaque élément

L'incrémentation du pointeur est déterminée par la formule:

(LEL * NEL - 1) / 4 + 1

3 - Fonction ALIGNDON

Cette fonction assure l'alignement du pointeur de données sur une adresse paire. Il s'en suit qu'un mot de remplissage peut éventuellement être généré.

4 - Fonction GENERDON

La fonction GENERDON admet deux paramètres principaux:
- déclaration sans initialisation
- déclaration avec initialisation

Un certain nombre de paramètres définissent de plus le type et la longueur de la variable et de l'initialisation.
La fonction GENERDON utilise les fonctions sémantiques de génération du code objet:

GENERCODE , ENREGISTRE , ENREGISTRINIT

Dans le cas d'une génération de données sans initialisation, le compilateur procède à l'initialisation par la valeur zéro.
Dans le cas d'une demande d'initialisation il faut considérer divers facteurs. En particulier l'initialisation par adresse nécessite un chargement translatable. Lors de l'initialisation d'un tableau d'octets, il faut veiller à ce

que tous les éléments soient initialisés malgré les problèmes soulevés par l'initialisation répétitive au moyen de chaines de caractères.

5 - Fonction INITDCL

Cette fonction permet l'initialisation d'une variable simple au moyen d'un appel à la fonction GENERDON.

6 - Fonction GENERTESTDUMMY

Cette fonction détermine le type d'un bloc de données. L'identifieur est introduit dans la table des identifieurs (COPID) avec le type suivant:

Bloc	Type
Fictif	Variable
Commun	Externe
Global	Externe

Dans le cas d'un externe, il y a déclaration d'une section correspondante (DSECT ou CSECT)

7 - Fonction REENTRANCE

Cette fonction est appelée lorsqu'un identifieur reçoit une affectation. Elle permet de positionner un indicateur dans l'entrée correspondante de la table des identifieurs. Cet indicateur est employé par les références croisées.

3.5.3. Sémantique des déclarations particulières

3.5.3.1. Déclaration des constantes

Tout nombre entier apparaissant dans un programme LP80 peut être remplacé par un identifieur de constante préalablement déclaré. Lors de la déclaration, celui-ci est introduit dans la table des identifieurs (COPID) et la valeur numérique entière qui lui est associée et qui peut être le résultat d'une expression arithmétique est introduite dans les emplacements INF1-INF2.

3.5.3.2. Déclaration des procédures

Le nom de la procédure est introduit dans la table des identifieurs (COPID). Le déplacement par rapport à la base programme du début de la procédure est mis dans DEPLID alors que le registre de retour, déclaré ou implicite, est préservé dans STYPID.
Un branchement à l'étiquette fictive L1 est généré en tête afin de sauter le corps de la procédure. Un appel à la fonction ETIQUETTE est effectué pour cela. En fin du corps de procédure,il y a génération d'un branchement indirect sur le registre de retour. L'étiquette fictive L1 est ensuite définie.

3.5.3.3. Déclaration des fonctions assembleur

Un certain nombre d'identifieurs d'instructions sont prédéclarés (voir Annexe 2). Lors de la déclaration d'une fonction assembleur, l'identifieur est introduit dans la table des identifieurs (COPID) et la valeur correspondant au code opération est rangée dans INF1. Le code opération est vérifié, ce qui permet de déterminer en particulier si l'instruction est à opérande immédiat. Dans ce cas, un indicateur est positionné dans le sous-type afin de permettre de s'assurer du bon emploi ultérieur de l'instruction.

3.5.3.4. Déclaration de condition

L'identifieur est rangé dans la table (COPID) et l'entrée correspondante est renseignée à deux niveaux.
INF1 reçoit la valeur du code opération nécessaire (x68 ou x69) suivant que l'on teste une condition satisfaite ou non satisfaite.
STYPID reçoit la valeur du masque de test du code condition.

3.5.3.5. Déclaration des registres et des bases

Les registres, les bases et les variables prédéclarées doivent apparaître dans les références croisées si celles-ci sont demandées.
La table des identifieurs standard n'étant pas modifiable, il est nécessaire de déclarer ces identifieurs au du début d'un programme (ouverture implicite du bloc 0).
Les procédures IDENT_REGISTRES et STANDARD permettent d'effectuer ce traitement

3.5.4. Les fonctions nécessaires à l'édition des liens

Elles ont pour but de permettre à l'éditeur des liens de résoudre les références entre les différents segments. Leur rôle est indépendant des autres fonctions sémantiques. Les directives suivantes sont générées par le compilateur LP80.

3.5.4.1. Traitement des SEGMENT PROCEDURE ou MAIN

Un appel à la procédure DEBSEG permet de procéder à un ensemble de déclarations. Celles-ci sont opérées par des appels aux fonctions ENREGISTRE et GENERCODE.
Dans l'ordre on a:
- génération du début du module objet
- déclaration d'une section de contrôle non standard pour les données
- déclaration d'un section de contrôle non standard pour les informations nécessaires à la trace
- déclaration du nom de segment précisé ou de l'identifieur "MAIN".
- définition de ce nom comme étant la section de contrôle standard
- déclaration éventuelle du nom externe CDEBUG si l'option

DB a été précisée.
Certaines de ces déclarations ne sont pas obligatoirement utiles par la suite (données, trace). Il a cependant été jugé préférable de les déclarer systématiquement afin de faciliter leur emploi éventuel, le numéro de déclaration étant alors toujours le même.

3.5.4.2. Traitement des EXTERNAL PROCEDURE

Une déclaration du nom de référence externe primaire est effectuée. L'identifieur est introduit dans la table et le numéro de déclaration est préservé dans l'entrée correspondante (NDCLID).

3.5.4.3. Traitement des GLOBALDATA ou COMMONDATA

Un appel à la procédure GENERTESTDUMMY permet de déclarer le nom externe et de définir celui-ci comme étant une section fictive ou une section de contrôle non standard. Une DUMMYDATA ne donne lieu à aucune déclaration. En fin de déclaration de bloc la longueur et la protection de celui-ci sont générées.

3.5.4.4. Gestion des références externes

Les procédures et les blocs externes peuvent être référencés dans le corps du programme soit par des appels pour les procédures soit par des chargements d'adresse (dans une base ou un registre) pour les blocs ou les procédures.
Le chargement des bases qui permettent l'adressage de blocs (GLOBALDATA ou COMMONDATA) est pris en charge par le compilateur. Le programmeur doit seulement veiller à ne pas modifier intempestivement les bases concernées, leur emploi n'étant pas interdit. Le compilateur doit charger correctement les bases lors de chaque entrée dans un segment. Ces entrées peuvent être:

- l'entrée principale (premier bloc) d'un segment
- une entrée secondaire (ENTRY) d'un segment
- une entrée implicite au retour d'une procédure externe

Dans ces trois cas, un appel à la fonction INITBASES permet de générer la séquence d'instructions suivante:
- chargement du registre de base de la section standard au moyen de l'instruction LIAI
- Chargement si nécessaire des registres permettant l'adressage des blocs de données par l'instruction LBR.

Dans ce cas, il faut disposer d'un opérande contenant l'adresse de la section considérée. Cet opérande est généré dans un vecteur d'adresses situé en fin de section standard. Ce vecteur est géré par la fonction GENERADREXT qui fournit pour chaque appel le déplacement de l'opérande dans le vecteur. Ce vecteur est situé à un emplacement défini par la référence avant numéro UN.
Les instructions de chargement doivent donc être translatable par rapport à cette référence avant (fonction GENERINST). Dans le corps du programme, la demande explicite d'une adresse externe (appel de procédure par exemple) passe par le même mécanisme.
En fin de segment, le vecteur d'adresses est généré au moyen d'une directive de chargement avec translation par rapport aux identifieurs externes.

3.5.5. Sémantique des instructions du langage

Les actions sémantiques liées au traitement des instructions ont pour but de prendre les renseignements utiles à la génération dans la table des identifieurs et de produire le code objet correspondant.
Le paramètre essentiel pour ces fonctions sémantiques est le compteur d'emplacement PTRPROG qui indique le déplacement par rapport à la base programme. Sa valeur initiale par défaut est 0. Une valeur différente peut être obtenue par les spécifications de segment.

3.5.5.1. Structure du code objet

Le code objet est toujours généré dans la section de contrôle standard dont le numéro de déclaration est zéro. Les directives suivantes sont employées:

- définition d'origine

cette directive indique à l'éditeur de liens l'emplacement à partir duquel la génération de code doit se faire.

- chargement absolu

cette directive est employée pour générer la plupart des instructions, l'adressage basé ne nécessitant aucune translation sauf dans le cas des références en avant.

- chargement translatable

cette directive est employée uniquement pour les références avant (étiquettes réelles ou fictives et références aux constantes -numériques et adresses- générées en fin de segment).

- définition de référence en avant

cette directive est générée chaque fois qu'une étiquette déja référencée apparaît. L'expression associée contient la valeur de PTRPROG.

3.5.5.2. Traitement des étiquettes

1 - Etiquettes fictives

La génération du code correspondant à certains traits du langage entraine la manipulation d'étiquettes dites fictives. Ces étiquettes qui ne sont pas accessibles par un nom sont employées chaque fois qu'un branchement implicite est nécessaire. C'est le cas lors de la génération des instructions suivantes:

LOOP, IF THEN ELSE, FOR, REPEAT, WHILE, CASE OF, PROCEDURE, :DCB, ainsi que pour l'évaluation des expressions conditionnelles.

Il existe deux étiquettes fictives dénommées L1 et L2.

Par exemple:

```
1 étiquette:      LOOP instruction
     ===>         L1 : instruction
                       GOTO L1
2 étiquettes:     FOR RA UNTIL RB DO instruction
     ===>         L1 : comparer RA et RB
                       GOTO L2 si RA > RB
                       instruction
                       incrémenter RA
                       GOTO L1
                  L2 :
```

La récursivité des automates entraine que la gestion de chacune des étiquettes se fasse dans une pile liée au pointeur de niveau général PP. Pour des raisons de programmation, les deux étiquettes L1, L2 sont gérées dans une pile de double-mots. L'emploi de L2 ne se fait que si L1, première étiquette utilisable, est déja employée. Chaque partie du double-mot peut contenir plusieurs valeurs selon l'état de l'étiquette qui lui est associée.

- 0 : l'étiquette fictive n'existe pas
- <0 : la valeur indiquée est le numéro de référence en avant changé de signe. C'est le cas lorsqu'une référence à l'étiquette a eu lieu avant sa définition.
- >0 : le déplacement par rapport à la base programme de l'étiquette. C'est le cas lorsque la définition a eu lieu avant la référence.

La manipulation des étiquettes fictives passe par la procédure ETIQUETTE, quatre cas étant possibles:

- branchement à une étiquette fictive non apparue
- apparition d'une étiquette fictive déja référencée
- branchement à une étiquette fictive déja apparue

- apparition d'une étiquette fictive non référencée

2 - Traitement général des étiquettes

Le traitement général de toutes les étiquettes, réelles ou fictives est assuré par une procédure unique ETIQUETTE. En plus de fonctions précédemment vues, elle permet:

Pour les étiquettes réelles:

- déclaration
- référence
- branchement
- définition

La procédure est appelée de plus lors de l'ouverture (BEGIN) et la fermeture (END) de chaque bloc afin de pouvoir gérer les piles nécessaires aux instructions CYCLE et EXIT.

Il faut disposer de quatre piles de travail:

DEBBLOC: qui contient la valeur de PTRPROG lors de l'ouverture du bloc

NUMBLOC : qui contient la valeur de l'étiquette numérique associée au bloc

PERMEXIT : qui indique la possibilité de sortie du bloc. Cette sortie est interdite lorsqu'il s'agit du bloc principal d'une procédure.

FINBLOC: qui contient éventuellement le numéro de référence en avant, utilisé par une instruction EXIT et qui devra être défini lors de la fermeture.

La procédure ETIQUETTE dispose des procédures de service suivantes:

INCREFAV: qui incrémente et teste le compteur de références en avant. En cas de dépassement une erreur est signalée.

RECHBLOC: qui permet de rechercher dans la pile NUMBLOC le premier bloc ayant un numéro déterminé (instructions CYCLE et EXIT). Si la recherche n'est pas satisfaite, une erreur est signalée.

GENERBRANCH: qui génère une instruction de branchement. Le code opération (x68 ou x69) et le masque de test du code condition sont passés en paramètres.

3 - Cas particulier du CASE OF

La définition syntaxique du CASE OF, issue de LP70 ne permet pas de déterminer la fin de cette instruction dans le cas d'une imbrication directe de plusieurs CASE. Dans ce cas, la fin est établie au moyen d'une action sémantique consistant à dépiler d'un niveau lorsque toutes les instructions prévues dans l'appel ont été générées. Si cela se produit en dehors d'une imbrication, une erreur est signalée.

4 - Cas particulier des points d'entrée

Les points d'entrée sont gérés de la même manière que les étiquettes. Les seules différences apparaissent lors de la déclaration et la définition d'un ENTRY.
Lors de la déclaration, il faut procéder à la déclaration du nom externe.
De même, lors de la définition, une définition de l'externe est nécessaire. Un appel à la procédure de génération de la séquence de chargement des bases est effectué (INITBASES).

3.5.5.3. Traitement des affectations

On distingue les affectations simples et les affectations multiples. Les deux peuvent porter sur deux classes différentes d'éléments:
- affectation du résultat d'une expression conditionnelle à un ou plusieurs booléens.
- affectation du résultat d'une expression arithmétique à une ou plusieurs variables.
L'ordinateur IRIS80 ne permettant pas un transfert de mémoire à mémoire, il est toujours nécessaire de disposer d'un registre de travail. Toutefois, ce registre qui doit être explicite dans le cas d'une expression arithmétique, est déterminé implicitement dans les affectations booléennes. Les 4 bits du code condition (dans le PSD) peuvent être chargés ou déchargés au moyen des instructions

LCFI, LCF, STCF. Ce jeu réduit d'instructions est suffisant pour traiter les expressions booléennes.
Si l'assignation est simple, le problème est trivial, puisqu'il suffit de procéder au chargement d'un booléen ou d'un registre avec le résultat de l'expression.
Une affectation multiple implique de conserver les différents renseignements utiles sur les variables à charger, l'affectation ne pouvant se faire qu'aprés l'évaluation de l'expression.
Pour cela, un ensemble de piles est employé:
- numéro du registre de base
- numéro de registre d'index
- type d'adressage (direct ou indirect)
- type de l'opérande (BYTE, SHORT, ...)
- déplacement par rapport à la base
- adresse dans la table des identifieurs, ce renseignement étant utilisé par la TRACE si nécessaire.

Chaque rencontre du symbole := entraine le remplissage du niveau de pile par une action sémantique et le passage au niveau supérieur par l'automate. Ceci entraine que lors de la détection du début d'expression, il est indispensable de redescendre d'un niveau pour y prendre le registre opération dans le cas d'une expression arithmétique.
A la fin de l'évaluation de l'expression, l'automate dépile les différents niveaux, et il suffit de procéder au chargement de chaque variable au moyen d'une instruction STORE (STB,STH,STW,STD) ou STCF pour les booléens.
Dans le cas où la variable à charger est un registre, un appel à la procédure GENERTESTINST permet de ne pas procéder sur l'affectation d'un registre à lui même.

3.5.5.4. Evaluation des expressions conditionnelles

L'évaluation d'une expression conditionnelle revient à évaluer chacune des conditions simples et à combiner leur resultat en fonction des opérateurs ET et OU.

1 - Evaluation d'une condition simple

Il s'agit de positionner éventuellement le code condition puis de le tester au moyen d'une instruction de branchement conditionnel.

Ce positionnement peut être fait de différentes manières:

- lors d'une comparaison à zéro par MTB, MTH, MTW
- lors du test d'un booléen par LCF
- dans les autres cas par CB, CH, CW, CD, CI

Le masque de test intervenant dans l'instruction de branchement est directement lié à l'opérateur de relation ou à la condition déclarée. Dans le cas du test d'un booléen il est implicite.

2 - Evaluation d'une combinaison de conditions

Les deux seules possibilités étant le ET et le OU, et leur emploi étant exclusif l'un de l'autre, le problème est assez simple.

On génère:

- pour le ET (Cndt1 AND Cndt2)

```
          évaluation de Cndt1
          branchement en L2 si Cndt1 non satisfaite
          évaluation de Cndt2
          branchement en L2 si Cndt2 non satisfaite
     L1 : VRAI
          . . .
     L2 : FAUX
```

- pour le OU (Cndt1 OR Cndt2)

```
          évaluation de Cndt1
          branchement en L1 si Cndt1 satisfaite
          évaluation de Cndt2
          branchement en L2 si Cndt2 non satisfaite
     L1 : VRAI
          . . .
     L2 : FAUX
```

Dans le cas où l'expression se réduit à une condition simple on considère qu'il s'agit d'un ET .

3 - Emploi du résultat de l'évaluation
Les étiquettes L1 et L2 correspondent aux resultats VRAI et FAUX. Dans le cas d'une affectation de valeur à un booléen, des instructions LCFI sont générées aprés chaque étiquette afin d'obtenir le chargement correct du registre opératoire implicite (C.C. du PSD).
Dans les autres cas (WHILE, IF) des instructions sont générées aprés l'une et/ou l'autre des étiquettes.

3.5.5.5. Evaluation des expressions arithmétiques

La planche 13 de l'annexe 1 donne le graphe d'une expression. Les différents opérateurs sont de trois classes distinctes:
- adresse
- décalage
- arithmétique et logique

Le traitement effectué est spécifique à chacune des classes. La génération de code est effective chaque fois qu'un terme de l'expression est reconnu, c'est à dire qu'il y a évaluation de gauche à droite sans possibilité de parenthésage. Le registre opératoire (ou le couple) permettant d'évaluer l'expression est le registre (ou le couple) situé à gauche du dernier symbole := .

1 - Opérateur adresse
Deux cas sont considérés selon que l'identifieur est un externe ou non.
- dans le premier cas un appel à la fonction GENERADREXT permet d'obtenir le déplacement de la constante adresse utilisable par rapport à la référence avant numéro un.
Une simple instruction LW ou LBR est donc générée.
- dans le deuxième cas le type de la variable intervient

(fonction CSTADR). De plus un index peut être employé. La encore deux cas sont possibles:
+ la variable est synonyme d'une quantité dont l'adresse est inférieure à x3FFF. Une instruction LI peut donc être employée avec comme valeur immédiate l'adresse exprimée en fonction du type de la variable. Si un index est précisé, une instruction AW suit.
+ la variable est basée. Dans ce cas une instruction LVAW pouvant être indexée si l'élément est de type mot est générée. Pour les autres types, une instruction SHIFT suivie éventuellement d'un AW pour le registre index est nécessaire.

2 - Opérateur de décalage

Le code opération et le mode de décalage sont directement obtenus à partir de l'opérateur et des opérandes.

3 - Opérateurs arithmétiques ou logiques

On distingue en tout 14 classes possibles d'opérations (chargement, addition, multiplication, union, ...) pouvant s'appliquer à 8 classes possibles de variables. Ces deux paramètres sont fournis à la procédure CODOPRAT et permettent d'accéder à une matrice 14X8 donnant les codes opérations des instructions à générer. Ce code peut ne pas exister (par exemple addition d'une variable de type BYTE). Dans ce cas une erreur est signalée et une instruction inopérante est générée.
Il faut noter que les registres de base ne possèdent pas d'instruction autre que le chargement (LBR). La plupart des opérations sont donc impossibles. Il serait possible de procéder dans ce cas à la génération d'une séquence d'instructions afin d'avoir de plus grandes possibilités.
Par exemple l'expression

```
        BO := R4 + 30;
```

pourrait être générée sous la forme:

```
        LBR   BO,R4
```

```
XW    R0,B0
AI    R0,30
XW    R0,B0
```

ce qui permet de ne pas modifier la valeur du registre intermédiaire implicitement employé. Toutefois, tous les cas n'auraient pas pu être résolus. L'option prise est donc de signaler une erreur.

3.5.5.6. Traitement des déclarations implicites

Le compilateur doit procéder à la réservation implicite de certaines valeurs utilisées dans les expressions arithmétiques. Ces valeurs doivent être générées lorsque aucune instruction immédiate existe ou bien que la valeur ne tient pas dans le champ immédiat d'une instruction. La procédure NBRCHN permet de déterminer ces cas. La réservation de constante éventuelle est effectuée par la fonction GENERCST.
Elle reçoit en paramètres l'adresse d'un double-mot contenant la constante et un indicateur définissant la longueur de celle-ci (mot ou double-mot). Elle gère une table (TABCST) de mots dont la longueur dépend de la constante LTABCST. Cette table contient toutes les constantes déja existantes. Une recherche séquentielle permet de trouver le déplacement d'une valeur déja introduite et dans le cas contraire d'insérer la nouvelle valeur.
L'utilisation de ces constantes nécessite une translation par rapport à la référence avant numéro deux.
En fin de compilation, la table TABCST est générée sur une frontière de double-mot et la référence avant est définie.

3.5.6. Mécanisme de trace

3.5.6.1. But, généralités

Le mécanisme de trace permet de suivre l'évolution d'un programme en donnant des indications sur les valeurs successives d'éléments de mémoire, d'appel de procédure ou de passage sur des étiquettes. Des messages sont imprimés par un système de mise au point appelé lors de l'exécution. L'option DB qui peut être donnée lors de l'appel du compilateur permet de déterminer si les ordres de trace (TRACE, NOTRACE, SILENCE, NOSILENCE) et de SNAP doivent être pris en compte. En son absence l'analyse syntaxique subsiste mais aucune action sémantique est entreprise. Cela permet, sans modifier le source d'un programme d'obtenir des versions avec ou sans action de mise au point.

3.5.6.2. Déclarations TRACE et NOTRACE

Lors de la première déclaration de TRACE un appel à la procédure OPENTRC permet d'initialiser les tables utilisées par les fonctions de trace. Ces tables sont dimensionnées par la constante NVARTRC (valeur actuelle 64).

CHNVAR : tableau d'octets permettant de chainer les éléments

INDXVAR : numéro du registre index associé éventuellement à la variable

DEPLVAR : valeur du déplacement éventuellement associé à la variable

PTRC : déplacement du bloc descripteur de l'élément dans la section de contrôle d'informations pour la trace.

La déclaration de trace entraine pour les identifieurs concernés le positionnement de la valeur TRCID dans l'entrée correspondante de la table des identifieurs. Cette valeur donne le numéro d'entrée dans les tableaux précédents.

De-plus un bloc descripteur de l'élément est généré dans la

section de contrôle des traces (fonction DECLTRACE).
Ce bloc est de la forme:

```
+--------+--------------------------+
! silence! adr. élément suivant     !
+--------+--------------------------+
! type   ! déplacement / base       !
+--------+--------------------------+
! long.  ! identifieur              !
+--------+ . . . . . . . . . . . . .!
!                                   !
/                                   /
/                                   /
+-----------------------------------+
```

L'indicateur de silence est positionné dynamiquement, lors de l'exécution, par les instructions SILENCE et NOSILENCE. Il permet de supprimer l'impression des messages.
L'adresse de l'élément suivant est employée par les mêmes instructions lorsque toutes les variables sont concernées (ALL). C'est un simple chainage avant.
"Type" permet de définir l'élément (octet, demi-mot, procédure, ...)
Le déplacement par rapport à la base est utilisé lorsque l'élément est accédé avec un déplacement complémentaire.
Enfin la chaine de caractères définissant l'identifieur est introduite avec en tête la longueur.
La procédure TRACEVAR gère le mécanisme. Elle est appelée:
- lors d'une déclaration de TRACE
- lors d'une déclaration de NOTRACE
- lors de la rencontre d'un élément à tracer

3.5.6.3. Rencontre d'un élément à tracer

Il faut générer dans ce cas un appel à l'accompagnateur chargé de sortir les messages. Cet appel (construit par GENERTRACE) est constitué de 2 mots:

- le premier est une instruction CAL4 permettant de dérouter le programme vers l'accompagnateur
- le deuxième est l'adresse du bloc descripteur de l'élément tracé

```
+--------+----+--------------------+
!  CAL4  !flag! numéro de ligne    !
+--------+----+--------------------+
!  adresse du bloc descripteur     !
+----------------------------------+
```

La zone "flag" prend la valeur 1 dans le cas général. Lors du retour d'une procédure, elle vaut 2.
Le mécanisme adopté facilite la sauvegarde du contexte (voir chapitre 4).

3.5.6.4. Instructions de silence

Ces instructions permettent de modifier dynamiquement l'indicateur de silence figurant dans le bloc descripteur d'un ou plusieurs éléments. La séquence d'appel générée est similaire à la séquence de trace. La valeur prise par "flag" indique s'il s'agit d'une demande de silence ou de fin de silence et si elle concerne tous les éléments ou un seul.

3.5.7. Fonctions sémantiques générales

3.5.7.1. Fonctions de gestion du code objet

Le code objet généré est lié au système SIRIS8. Afin de se rendre indépendant le plus possible de celui-ci, les fonctions de génération du code objet sont systématiquement effectuées par des procédures spéciales.
Il est évident que le compilateur est fortement conditionné par la structure du langage objet.
La méthode adoptée doit cependant permettre de passser éventuellement sans trop de difficultés à un autre type de langage objet.
Il semble même possible de transformer le compilateur afin qu'il puisse produire des modules exécutables.
Les procédures employées sont les suivantes:

GENERCODE, ENREGISTRE, RECORD

1 - GENERCODE
Elle détermine les ordres à générer en fonction des paramètres fournis lors de l'appel. Par exemple lors d'une demande de chargement absolu d'un mot, la fonction procède à la génération de l'octet de contrôle x44 suivi des 4 octets du mot à charger.

2 - ENREGISTRE
Elle charge dans une section de contrôle donnée les octets passés en paramètres. Pour cela une définition d'origine peut être nécessaire si un changement de section a lieu.

3 - RECORD
Les octets en provenance de la procédure ENREGISTRE sont accumulés dans une zone de travail. Lorsque la zone est pleine, elle est vidée sur le fichier de sortie FILECO.

4 - DUMPCODE

Si la commande CODEON est positionnée, la fonction permet d'obtenir une liste en hexadécimal du code généré au fur et à mesure de sa génération.

3.5.7.2. Gestion des références croisées

Une procédure spécialisée assure l'ensemble des opérations utiles à l'édition des références croisées. Les fonctions réalisées sont les suivantes:

- initialisation du mécanisme
- lors de chaque référence à un identificateur, introduction dans le fichier références du numéro de ligne courante et chainage aux références précédentes (chainage inversé).
- lors de l'effacement d'un identifieur, introduction de celui-ci dans le fichier déclarations
- en fin de compilation, introduction des derniers articles dans les fichiers références et déclarations

ACCOMPAGNATEUR LP80

La modularité d'écriture adoptée pour le compilateur a été conservée pour le metteur au point.
Celui-ci doit satisfaire trois classes de services:
- gestion des TRACE et des SNAP demandés lors de la compilation
- récupération et analyse sommaire des déroutements lors de l'exécution d'un programme
- gestion conversationnelle ou non de primitives de mise au point

Ces trois services ont été groupés au sein d'un même mécanisme, non par nécessité puisqu'ils sont logiquement disjoints les uns des autres, mais parce que le recouvrement de fonctions nécessaires à chacune des classes permettait de réduire l'occupation en mémoire.
On distingue trois parties dans le metteur au point:
- chargement dynamique (SDEBUG)
- interface système
- traitement des fonctions

Les deux dernières parties forment le module CDEBUG qui peut être lié statiquement (sans passer par le mécanisme SDEBUG) lors de l'édition des liens.

1. Le mécanisme de chargement dynamique

1.1. Buts

Ce mécanisme a été introduit bien aprés le module CDEBUG afin de faciliter l'utilisation de celui-ci. Les contraintes suivantes impliquaient en effet de prévoir l'emploi du metteur au point lors de la compilation:

1 - Le système SIRIS8 permet de disposer des différentes protections d'accès à la mémoire avec obligation pour certaines tables de services système (FPT) d'être implantées en protection écriture. Or le système de points d'arrêt implique de modifier dynamiquement certaines instructions qui sont générées par défaut en protection écriture. Il était impératif de lever les protections des segments de programme dont on voulait faire la mise au point. Cela justifiait donc d'obliger le programmeur de se lier le metteur au point, une intervention au niveau de la compilation étant obligatoire en tout état de cause.

Cela entrainait:

- une lourdeur de l'emploi du metteur au point
- une modification de l'implantation en mémoire des différentes sections de contrôle, d'où certains effets de bord possibles.

Le hasard a permis d'avoir connaissance d'un service système non officiellement existant, mais bien réel et utilisable. Ce service (CAL1) permet de modifier la clé de protection d'une page. Il était donc possible de contourner la difficulté précédente.

2 - Le fait de ne pas vouloir modifier l'implantation d'un programme en mémoire entraine la nécessité de disposer du metteur au point dans un emplacement différent de la zone programme.

Il était hors de question d'introduire une modification du moniteur SIRIS8 afin d'y inclure au moins une partie de l'accompagnateur.

La seule zone mémoire disponible restait donc la zone dynamique commune.

Le module SDEBUG a donc pour but de lancer l'exécution d'un programme aprés avoir implanté le metteur au point en zone dynamique commune.

Pour cela on distingue deux phases de traitement.

1.2. Translation du metteur au point.

La première phase de SDEBUG va permettre la translation du module CDEBUG. Cela implique la translation de toutes les adresses absolues qui figurent dans ce module.
On peut distinguer deux types d'adresses:
- les adresses implicites générées en fin de segment
- les adresses figurant dans les données

L'évolution inévitable du produit et la maintenance aisée de celui-ci proscrit l'emploi de "bidouillages" afin de retrouver les mots à transformer.
Une écriture soignée du module CDEBUG permet de réduire au maximum les adresses du premier type, puisqu'une seule est indispensable. Elle correspond à l'adresse de la GLOBALDATA CDEBUGDATA qui contient les données utiles à la mise au point. La structure figée d'un module compilé par LP80 est telle que l'on connait alors les deux premières instructions:

```
LIAI B1,-1
LBR  B2,=V(CDEBUGDATA)
```

Le mot à modifier est donc situé à l'adresse d'implantation du module CDEBUG augmentée du déplacement figurant dans l'instruction LBR.
Il suffit alors de remplacer la valeur de ce mot par la nouvelle adresse de CDEBUGDATA.
La translation des adresses de la GLOBALDATA ne pose aucun problème puisque les variables initialisées sont nommées explicitement.

ETAT INITIAL

```
+-------------------------+
!                         !
!                         !
!      MONITEUR           !
!                         !
+-------------------------+
!      SDEBUG             !
+-------------------------+
!      CDEBUG             !
+ - - - - - - - - - - - - +
!      CDEBUGDATA         !
+-------------------------+
!                         !
!                         !
/                         /
/                         /
!                         !
+-------------------------+
```

TRANSLATION

```
+-------------------------+
!                         !
!                         !
!      MONITEUR           !
!                         !
+-------------------------+
!      SDEBUG             !
+-------------------------+
!    /CDEBUG/             !
+ - - - - - - - - - - - - +
!    /CDEBUGDATA/         !
+-------------------------+
!                         !
!                         !
/                         /
/                         /
!                         !
+-------------------------+   nouvelle limite
!   CDEBUGDATA            !   zone dynamique
+-------------------------+
!   CDEBUG                !
+-------------------------+
!                         !
+-------------------------+
```

Le mécanisme de translation consiste donc à:
- obtenir de la place pour le code
- translater CDEBUG dans la zone obtenue
- obtenir de la place pour les données
- translater CDEBUGDATA dans la zone obtenue
- procéder à la translation des adresses
- mettre le code en protection écriture

1.3. Modification du module à accompagner

Le traitement est trivial, le chargeur initiateur de SIRIS8 ne vérifiant pas l'adresse de lancement. Il suffit donc de transformer le module chargeable (assigné sur l'étiquette LM) afin de le faire démarrer non pas à l'adresse prévue, mais sur le point d'entrée d'initialisation du metteur au point en zone dynamique commune.
L'adresse de départ est utilisée lors de l'initialisation afin de relancer l'exécution au bon emplacement (sauf contre indication de l'utilisateur).
Il faut noter qu'il est possible de faire exécuter ainsi des programmes écrits aussi bien en mode C qu'en mode B. Dans ce cas cependant, l'exécution se fera en mode C restreint, les bases étant préchargées avec les valeurs suivantes:

B0	B1	B2	B3	B4	B5	B6	B7
0000	4000	8000	C000	10000	14000	18000	1C000

et les requêtes d'accès aux bases interdites.

ETAT LORS DU LANCEMENT

```
+------------------------+
!                        !
!                        !
!       MONITEUR         !
!                        !
+------------------------+
!                        !
!      PROGRAMME         !
!      UTILISATEUR       !
+------------------------+
!                        !
!                        !
/                        /
/                        /
!                        !
+------------------------+   nouvelle limite
!     CDEBUGDATA         !   zone dynamique
+------------------------+
!     CDEBUG             !
+------------------------+
!                        !
+------------------------+
```

1.4. Extensions possibles et prévues

L'analyse plus poussée du module initial peut permettre de gérer une table de noms symboliques qui seraient utilisés ensuite par les requêtes de mise au point.
Toutefois l'espace de travail nécessaire étant trop important dans le cas de grands modules (cas où la mise au point est plus difficile, et donc le metteur au point plus indispensable), la gestion efficace de tables de noms symboliques, éventuellement en relation avec la compilation,

ne peut être envisagée que sur un système offrant à l'utilisateur plus de mémoire.
La version C10 de SIRIS8 pourra sans doute permettre cela.

2. Interface système

Lors de l'écriture du module SDEBUG, ce segment a été réuni au segment de traitement afin de réduire les adresses externes et faciliter ainsi la translation.
Il est cependant logiquement distinct.

2.1. Buts

L'interface a pour buts:

- d'offrir un certain nombre de services système tels que:
  + arrêt de la tâche
  + obtention date et heure
  + détermination des pages mémoire accessibles
  + modification de la protection des pages
  + gestion des échanges
- de permettre la récupération des déroutements et de certaines interruptions
- de les récupérer et de les transmettre de manière standard aux fonctions de traitement
- de donner le contrôle à la procédure de traitement lors du premier appel (initialisation).

2.2. Initialisation

Lors de l'initialisation (premier appel), l'interface procède à deux traitements distincts.

1 - Analyse des noms d'étiquettes logiques (OPL) associées à la tâche afin de déterminer les fichiers sur lesquels les échanges pourront avoir lieu.
Les noms DBSI, DBLO, DBLS sont recherchés. La rencontre de

l'étiquette DBSI entraine la recopie des cartes de commande qui existent dans le fichier ainsi désigné sur un espace de travail temporaire. Cela est indispensable dans la mesure ou l'utilisateur veut pouvoir employer des fichiers de cartes, le système SIRIS8 ne permettant pas l'ouverture simultanée de plusieurs symbionts d'entrée.

2- La deuxième partie consiste à initialiser le mécanisme de mise au point, tant au niveau système qu'au niveau utilisateur.
Les services :INT et :TRAP sont utilisés. Les déroutements ainsi récupérés permettent de donner le contrôle à une fonction formattant un bloc de données (contexte utilisateur). Ce bloc est ensuite fourni à la procédure de traitement CDEBUGLP80.
Lors du premier appel, un contexte fictif est construit, et la procédure de traitement est appelée, avec demande d'initialisation (R0 = 0).

2.3. Récupération des déroutements

Certaines difficultés liées aux services offerts par SIRIS8 ont dues être contournées.

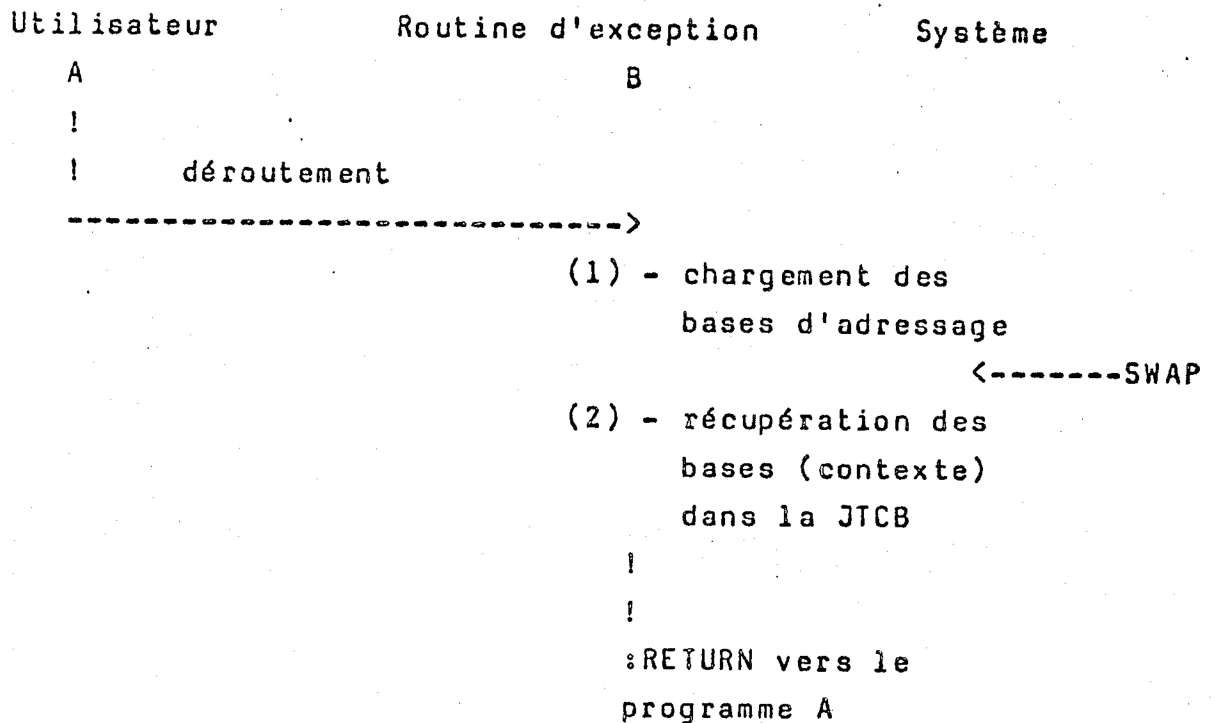
1 - Difficultées liées aux déroutements utilisateur
Lors de l'exécution d'une routine d'exception (:TRAP ou :INT) il est impossible de récupérer des déroutements. Les ATTENTION2 ne provoquent un nouveau déroutement qu'à la fin de la routine et les déroutements programme entrainent un abandon de la tâche.
Or il est intéressant d'interrompre une requête de mise au point lorsque son exécution est trop longue (affichage d'une trop grande zone mémoire par exemple). De même, l'utilisateur pouvant demander des références mémoire interdites, il est utile, plutôt que de vérifier systématiquement si les références sont possibles, de

pouvoir récupérer les erreurs éventuelles.
La solution adoptée consiste à faire travailler l'accompagnateur au niveau de l'utilisateur, c'est à dire à n'effectuer que le travail de formattage du contexte au niveau de la routine d'exception. Il est dès lors possible de récupérer les erreurs d'accès à la mémoire et de pouvoir interrompre une liste trop longue. Cependant il devient possible d'être interrompu durant la restauration du contexte utilisateur. SIRIS8 n'offrant aucune primitive permettant de se protéger, il faut vérifier lors de chaque déroutement si celui-ci ne s'est pas produit durant la séquence critique. Dans l'affirmative, l'ATTENTION est ignorée et le restauration du contexte reprise.

2 - Difficultés liées aux déroutements système

Lors de l'exécution d'une tâche en T.S., celle-ci dispose d'un quantum de temps à la fin duquel elle est obligatoirement interrompue et vidée sur espace secondaire (swap-out).
Le mode C ayant été introduit aprés coup, une erreur grave s'est introduite dans ce mécanisme.
Il n'existe en effet qu'une zone de sauvegarde unique pour les bases d'une tâche suspendue. Cela conduit immanquablement à l'écrasement des valeurs successives des bases lorsqu'une tâche est interrompue pour fin de quantum dans une routine d'exception.

Illustration:



Si un SWAP intervient entre les instants (1) et (2) du programme B les bases de A sont écrasées. Ce phénomène étant totalement asynchrone est peu facile à mettre en évidence.
La solution apportée consiste à n'exécuter en début de B que des instructions sans référence mémoire. Aucune base n'est donc utilisée jusqu'à ce que les bases devant être employées ultérieurement par le metteur au point (B1,B2) soient sauvées dans des registres. Il est alors possible de forcer l'initialisation des bases d'adressage au moyen d'un point d'entrée.

Les déroutements récupérés qui doivent être traités par l'accompagnateur possèdent un numéro de désignation. Si le déroutement se produit durant la mise au point, c'est la valeur négative de ce numéro qui est transmis en paramètre. Dans le cas normal, c'est le numéro qui est transmis dans R0.

Le contexte, reconstitué à partir des éléments dispersés par SIRIS8 est donné dans un bloc contenant:

- 2 double-mots pour le PSD (seul le premier est utile sous SIRIS8)
- 16 mots pour les registres généraux
- 8 mots pour les bases

2.4. Fonctions de service

Les fonctions d'arrêt de la tâche et d'obtention de l'heure sont fournies par les primitives système :RETURN et :TIME.

La détermination des pages mémoire accessibles se fait au moyen de l'analyse de la table des pages du segment zéro.

Le primitive SIRIS8 permèttant de modifier la protection d'accès d'une page n'étant pas officielle, son usage n'a pas à être dévoilé davantage.

Enfin la gestion des entrées-sorties est triviale. Le seul point à noter est la notion de voie d'échange. Les fonctions d'aide à la mise au point utilisent des voies spécialisées (dialogue, grandes impressions, commandes, etc ...) Cinq voies existent logiquement et permettent de satisfaire les besoins. Ces voies sont les suivantes:

En entrée:

- lecteur de cartes logique
- console logique

En sortie:

- imprimante logique
- console logique
- périphérique logique

Physiquement ces voies peuvent exister ou non et être disponible ou non. Logiquement, les échanges sont toujours satisfaits en fonction de la configuration (DBSI, DBLO, DBLS) et du mode de travail (BATCH ou T.S.).

3. Les fonctions de traitement: CDEBUGLP80

3.1. Buts

Ces fonctions ont pour but d'assurer les services suivants:

- système de TRACE
- sortie des SNAP
- analyse sommaire des erreurs programme
- requêtes de mise au point

3.2. Principes communs de fonctionnement

Le problème principal à résoudre est la communication entre le programme utilisateur et le système de mise au point. La contrainte principale est que le contexte utilisateur ne doit pas être modifié. De plus, il est utile de pouvoir utiliser le metteur au point sans que cette utilisation soit prévue lors de la compilation du programme. Le mécanisme le plus simple à mettre en oeuvre, et qui a été adopté consiste à provoquer un déroutement. Ce déroutement entraine la sauvegarde par le système du contexte de travail. L'initialisation dans l'interface d'une primitive de récupération permet d'obtenir le contrôle et de procéder aux traitements nécessaires.
L'instruction CAL4 est employée pour provoquer le déroutement.
Les différents champs de l'instruction servent à communiquer au metteur au point le type de traitement à effectuer.

```
+--------+----+-------------------+
!  CAL4  !Flag! Valeur immédiate  !
+--------+----+-------------------+
```

La zone registre (Flag) de l'instruction donne la fonction à exécuter selon les valeurs suivantes:

0	Trace
1	Retour de procédure
2	Demande de silence général
3	Fin de silence général
4	Demande de silence d'un élément
5	Fin de silence d'un élément
6	Snap simple (2 adresses)
7	Snap simple (adresse + longueur)
8	Snap avec contexte (2 adresses)
9	Snap avec contexte (adresse + longueur)
A	Point d'arrêt

3.3. Traitement des appels de trace

Deux fonctions sont à assurer pour la gestion des traces:
- positionnement d'un indicateur de silence pour un nombre quelconque d'éléments parmi ceux succeptibles d'être tracés. Dans ce cas aucun message n'est imprimé.
- sortie de l'état d'un élément au moment de l'appel. Dans ce cas un message spécifique est imprimé si l'indicateur de silence le permet.

3.3.1. Traitement des demandes de silence

Le mot situé derrière le CAL4 d'appel donne l'adresse d'un bloc descripteur. Suivant la valeur de la zone "flag", le premier élément de la chaine, ou tous les éléments sont positionnés à la valeur voulue.

3.3.2. Traitement des sorties de trace

Tous les messages imprimés comportent en tête le numéro de ligne dans le programme source. L'utilisateur peut ainsi retrouver rapidement l'instruction à l'origine du message.

plusieurs cas sont à considérer selon le type de l'élément concerné.

1 - Pour les procédures le message est le suivant:

NNNN : APPEL DE nom de la procédure
ou
NNNN : RETOUR DE nom de la procédure

2 - Pour les étiquettes le message est le suivant:

NNNN : PASSE PAR nom de l'étiquette

3 - Pour les piles on obtient les valeurs hexadécimales de tous les registres

4 - Pour les variables, le message dépend de l'instruction d'affectation. La forme générale est:

NNNN : nom de la variable := valeur

Le nom de la variable peut être suivi du registre index (et de sa valeur) ainsi que du déplacement complémentaire. La valeur est TRUE ou FALSE pour les logiques et 1, 2, 4, ou 8 octets en hexadécimal selon la longueur des variables arithmétiques.

Le traitement de trace n'est effectué que si l'indicateur de silence le permet.

3.4. Traitement des ordres SNAP

En fonction des valeurs du (ou des deux) mots suivant le CAL4, les limites inférieures et supérieures de la zone a visualiser sont calculées. Les procédures SNAPPER et DUMPER permettent de formatter des lignes donnant en hexadécimal le

contenu de la mémoire avec éventuellement la valeur du PSD et des registres.

3.5. Traitement des erreurs programme

En fonction du code d'erreur transmis en paramètre par l'interface, un message est constitué. Il indique en clair le type d'erreur et la valeur hexadécimale de l'instruction fautive.

DEROUTEMENT SUR L'INSTRUCTION XXXXXXXX message en clair

Les messages possibles sont les suivants:

Code	Message
32	PAGE HORS TABLE
34	SEGMENT HORS TABLE
36	INSTRUCTION PRIVILEGIEE ET INEXISTANTE
38	INSTRUCTION INEXISTANTE
3A	PAGE NON PRESENTE
3B	INSTRUCTION INEXISTANTE ET VIOLATION PROTECTION
3C	ADRESSE INEXISTANTE
3E	INSTRUCTION PRIVILEGIEE EN MODE ESCLAVE
3F	VIOLATION PROTECTION MEMOIRE
41	INSTRUCTION NON CABLEE
42	DEBORDEMENT DE PILE
43	DEBORDEMENT EN VIRGULE FIXE
44	ERREUR VIRGULE FLOTTANTE
45	ERREUR DECIMALE
49	CAL2
4A	CAL3
4C	INTERRUPTION EXTERNE

Deux types d'action sont entreprises selon que la tâche s'exécute en BATCH ou en T.S..

- en BATCH, un "dump" complet de la mémoire est imprimé et

la tâche est arrêtée sauf si des commandes sont données dans le fichier DBSI.

- en T.S. le contrôle est donné à l'utilisateur qui peut alors émettre des requêtes de mise au point.

3.6. Traitement des requêtes de mise au point

Les fonctions des différentes requêtes ont été données au chapitre 11 du manuel langage.

Le traitement est trés simplifié par le fait que la syntaxe des requêtes est la même pour toutes. La procédure TRAITE-REQUETE assure le décodage des requêtes et la gestion du dialogue avec l'utilisateur. Chaque message recu est décomposé en quatre éléments:

- le premier caractère de la requête
- la première valeur hexadécimale
- le séparateur (=, +, -, /, *)
- la deuxième valeur hexadécimale

Les erreurs éventuelles sont détectées à deux niveaux:

- soit lors de l'analyse (valeur hexadécimale erronnée)
- soit lors de l'utilisation (nombre de valeurs)

Toutes les adresses de mémoire sont relatives à une origine. La valeur effective d'une adresse est donc donnée par:

Exemple:

```
(ORIGINE 4000)

AFFICHER 237
               !
               !                   +------------+
               !                   !   x4000    !
               !                   +------------+
               !-------> + <-----------!
                         !
                         V
                        4237
```

3.6.1. Principe des points d'arrêt

Le mécanisme des points d'arrêt consiste à permettre de suspendre l'exécution d'un programme en un point quelconque et de donner le contrôle au metteur au point.
Afin de standardiser, un CAL4 est employé à cet effet.
L'instruction désignée par l'utilisateur est donc remplacée par un appel moniteur de ce type. Les instructions étant de longueur fixe, le problème est assez simple.
L'instruction désignée et son adresse sont préservées dans une table de 32 entrées. Le CAL4 qui remplace l'instruction comporte en valeur immédiate le numéro de l'entrée.
Lors de l'exécution d'un CAL4 de type point d'arrêt, le metteur au point peut vérifier que l'adresse du déroutement correspond à l'adresse dans la table. En cas de non conformité, aucune action n'est entreprise et le déroutement est traité comme une erreur programme.
Dans le cas normal l'instruction est remise en place, l'entrée dans la table est libérée, et le contrôle est donné à la procédure TRAITE_REQUETE aprés que le message suivant ait été imprimé:

STOP EN XXXXXX ORG. + DEPL. DE XXXXXX

L'indication du déplacement par rapport à l'origine courante permet à l'utilisateur de retrouver facilement l'emplacement du point d'arrêt dans son programme.

3.6.2. Requêtes de traitement

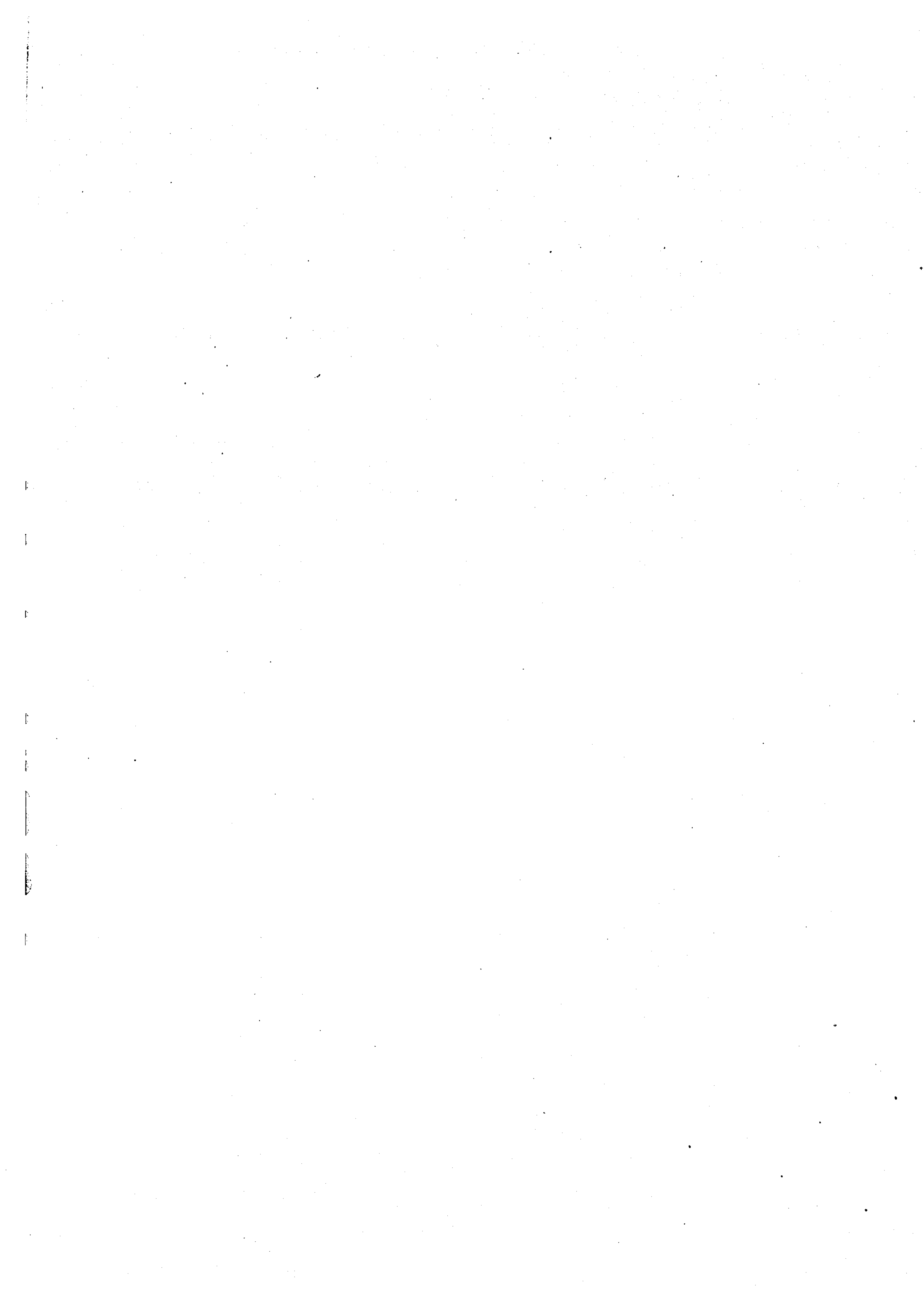
On peut distinguer trois classes de requêtes:
- requêtes à références mémoire
- requêtes à références contexte (registres, bases, PSD)
- requêtes sans références

Ces dernières se contentent d'offrir à l'utilisateur des

possibilités de renseignements divers:

+ obtention de l'heure
+ liste des points d'arrêt et de l'origine courante
+ calculatrice en hexadécimal

Cette dernière possibilité facilite grandement la mise au point, le calcul mental en hexadécimal n'étant pas particulièrement aisé.



C O N C L U S I O N

Au terme de cette réalisation, il est utile de dresser un bilan et d'entrevoir les perspectives possibles des deux produits.
On peut tout d'abord constater que le développement de LP80 et du metteur au point n'aura pas été inutile.
Plusieurs projets importants ont en effet adopté ce langage comme moyen de programmation. Il serait certes prétentieux de croire que cela est du aux qualités intrinsèques du langage.
Sur ce point, il faut bien admettre que LP80 n'est qu'un surensemble de LP70 et que les possibilités nouvelles qu'il offre ne sont certainement pas à même d'entrainer à elles seules le choix de ce composant.

Le mode C est un fonctionnement de l'IRIS80 qui autorise l'expression plus claire et facilite la manipulation des structures de données.
LP80 a eu et a toujours l'avantage d'être le seul traducteur apte à permettre l'emploi de ce mode au niveau machine.
Il était donc normal que les quelques projets de développement de logiciel qui ont eu à faire un choix de langage se décident pour LP80.

Il est par ailleurs paradoxal de constater que les utilisateurs de LP80 pour l'écriture de produits de moindre importance ont obéi à des motivations tout à fait différentes dans le choix du langage. Le mode C n'étant pas pour eux une nécessité, ce sont plutôt certains avantages

que l'on pourrait presque qualifier de "gadgets" qui ont emporté leur décision.
Le metteur au point peut à la limite être rangé dans cette catégorie.
Sa facilité de mise en oeuvre, la simplicité de ses commandes ont intéressé des utilisateurs qui, pour des raisons diverses, n'avait pas cru devoir adopter LP80.

Ainsi aura-t-on vu l'accompagnateur LP80 aider à la mise au point de programmes écrits dans des langages directement concurrents.

A propos de l'implémentation des deux produits, quelques remarques s'imposent.

Certaines options de départ qui tenaient compte de la structure et des possibilités du système GEMAU ont été abandonnées.
En particulier l'utilisation des fichiers est restée de conception classique alors qu'il aurait été plus intéressant de valider certaines possibilités d'un système proposant des espaces archivables importants.

Cet aspect n'a cependant pas été totalement ignoré, et la structure modulaire proposant des protocoles d'interface bien déterminés permettrait d'implanter rapidement le compilateur et le metteur au point sur un système plus évolué que SIRIS8.

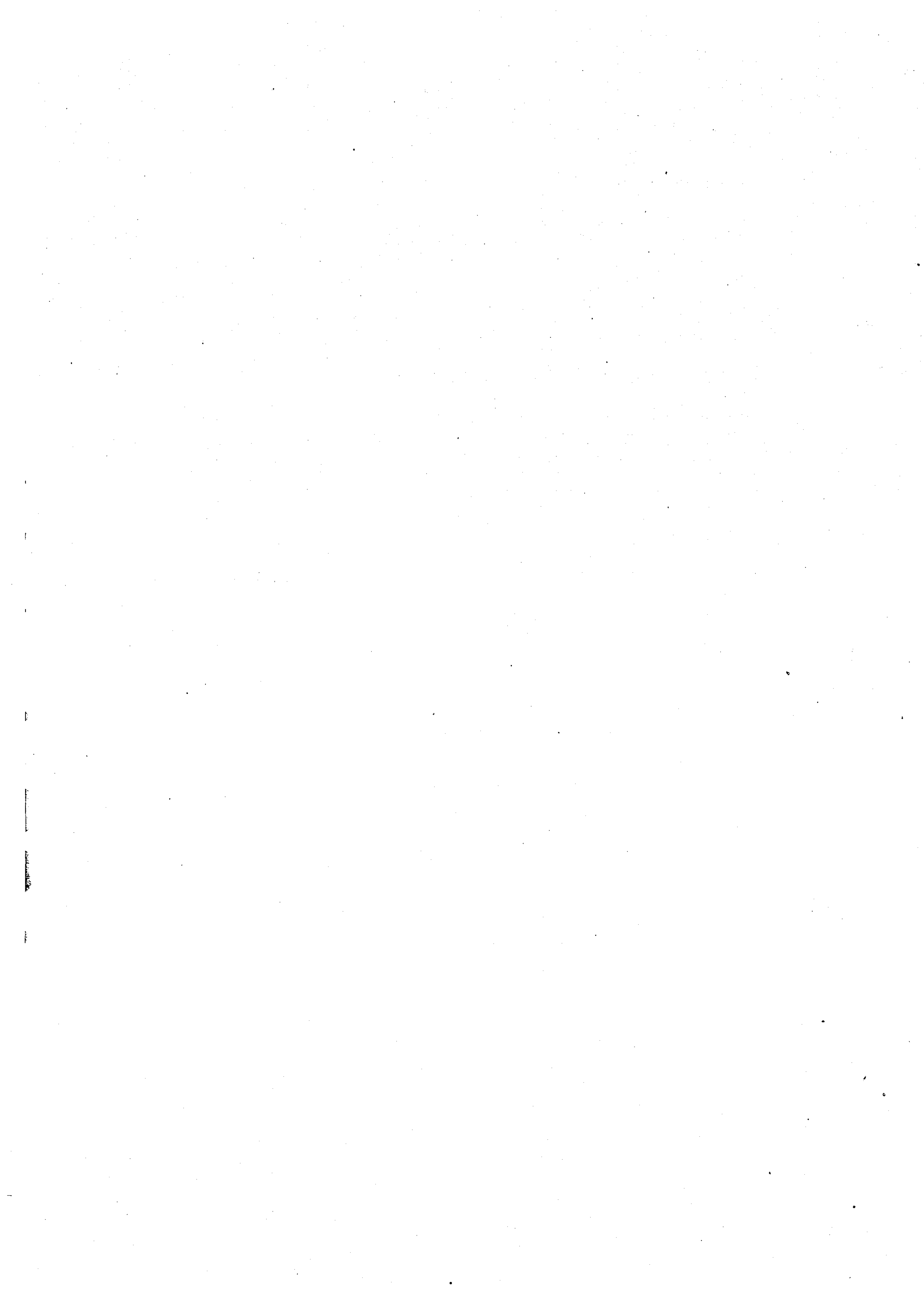
Concernant l'utilisation du mode C sur ce système, il est apparu que ce mode de traitement -qui certes n'est pas officiellement employable- n'avait sans doute pas fait l'objet d'une étude bien approfondie.
Certaines surprises lors des appels de primitives système et les problèmes soulevés par la modification intempestive des bases en T.S. ont permis d'apprécier ce point.

Le metteur au point ne comporte pas actuellement certains traits qui rendraient son emploi encore plus

souple. Nous avions initialement l'intention de produire enfin un ensemble cohérent entre le compilateur et son metteur au point.
Il était tout à fait possible, étant donné que nous avions la maitrise d'oeuvre des deux produits, de fournir par l'intermédiaire de l'éditeur des liens des renseignements exploitables lors de la mise au point (d'une certaine manière la table des symboles).
Nous avions pour cela étudié les possibilités de mise au point en terme de langage initial. L'éditeur de liens de SIRIS8 offre bien certaines possibilités de transmission de symboles, mais il ne tient pas compte des contraintes de la structure de blocs d'un langage.
L'aboutissement du système P3S et un avenir moins flou pour la gamme IRIS80 auraient sans doute permis, nonobstant les problèmes du C.I.C.G., de mener à terme cette idée.

La dure réalité nous a obligé à rester à un niveau de développement qui ne présente guère d'originalité.
Du moins offre-t-il des services non négligeables.

ANNEXES



A N N E X E 1

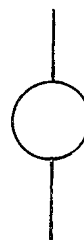
GRAPHE DU LANGAGE

1. NOTATIONS

Les transitions sont normalement représentées par des flèches horizontales.

- transition ordinaire unité syntaxique ⟶
- transition vide σ ⟶
- transition PUSH (appel d'automate) nom de l'automate ⟹
- transition PULL (sortie d'automate) - - - - - - - - - ->

Les états sont représentés par des cercles prolongés éventuellement par des traits verticaux.



2. ABREVIATIONS

idndecl	: identifieur non déclaré
nbent	: nombre entier
nbflot	: nombre flottant
chaine	: chaîne de caractères
idreg	: identifieur de registre
idbase	: identifieur de base
type	: BYTE, SHORT, WORD, LONG, REAL, LONGREAL
idglb	: GLOBALDATA, DUMMYDATA, COMMONDATA
idvar	: identifieur de variable
idlogic	: identifieur de logique
idptr	: identifieur de pointeur
ipile	: identifieur de pile
valogic	: TRUE, FALSE
idext	: identifieur de procédure externe
relop	: opérateur de relation : <,>, <= , >=
shop	: opérateur de décalage
logop	: opérateur booléen : AND, I, XOR

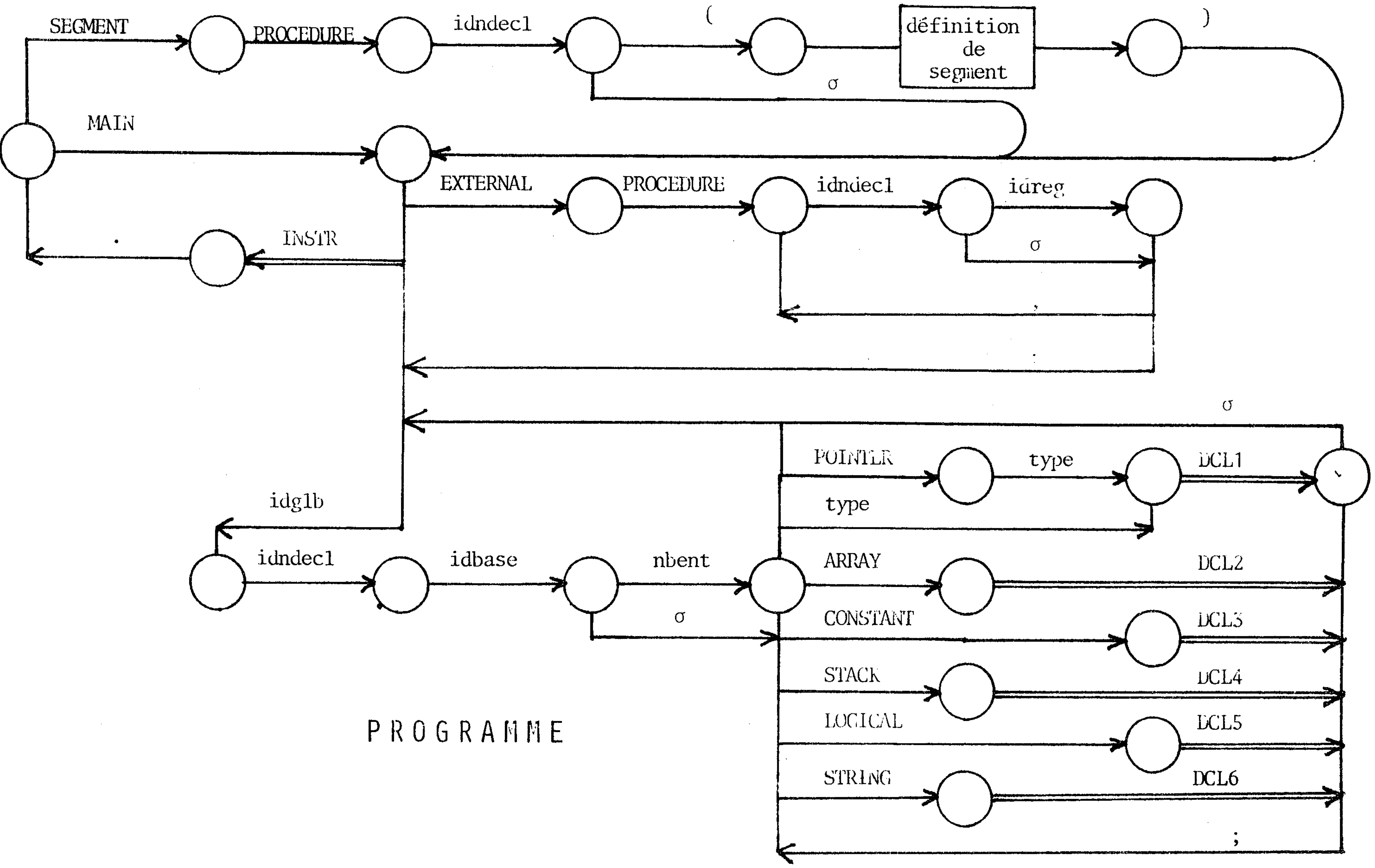
3. LISTE DES AUTOMATES

PROGRAMME

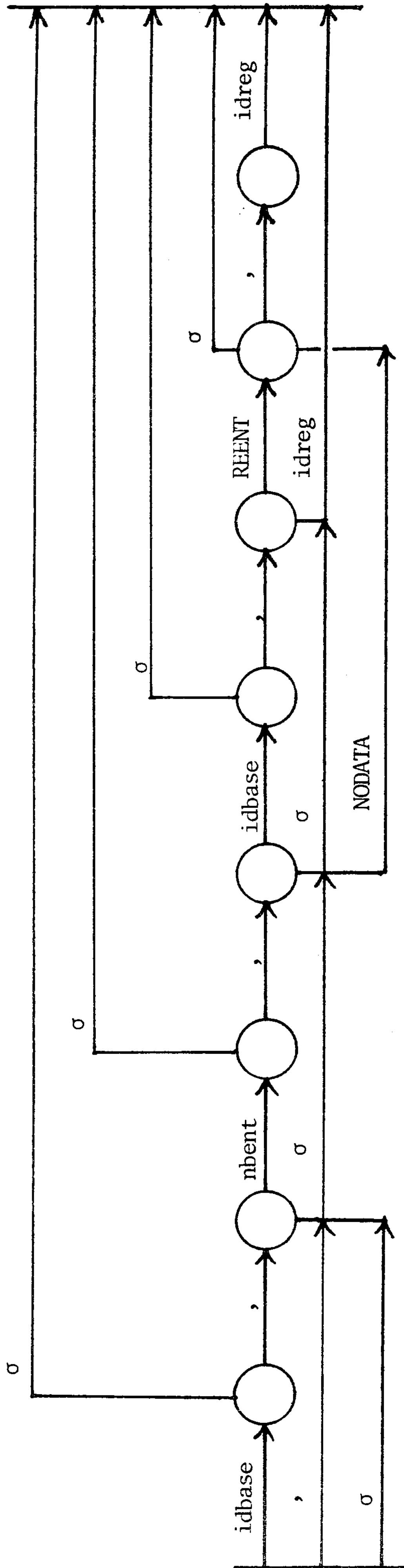
DCL1	déclaration de variable ou de pointeur	01
DCL2	déclaration de tableau	03
DCL3	déclaration de constante	04
DCL4	déclaration de pile	06
DCL5	déclaration de logique	06
DCL6	déclaration de chaîne	07
VAR	variable	07
CSTE	constante adresse ou numérique	08
ASSREG	assignation simple de registre	09
ASSREG1	assignation de registre d'index	10
ASSREG2	assignation de registre	10
EXPLOGIC	expression booléenne	10
ASSVAR	assignation de variable, base, registre	11
CNDT	condition	12
EXPCNDT	expression conditionnelle	12
EXPR	expression numérique	13
INSTR	instruction composée	14
	instructions d'assignation	14
	déclarations	15
	instructions de contrôle	17
	instructions de branchement	18
	instructions diverses	18

NOTA BENE : Les automates concernant les macro-instructions SIRIS8 ne sont pas décrits.

PLANCHE 1



PROGRAMME



DEFINITION DE SEGMENT

PLANCHE 3

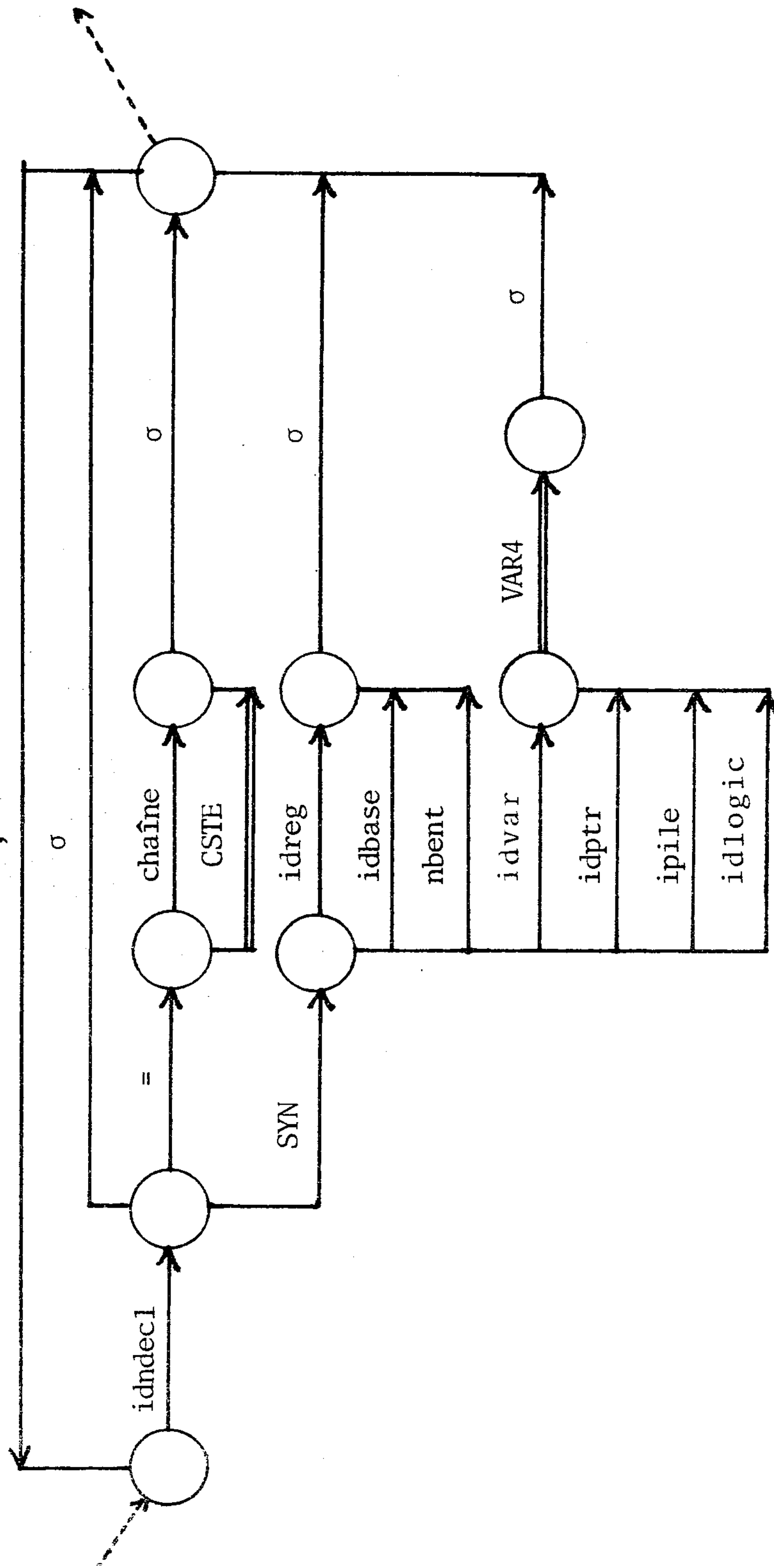
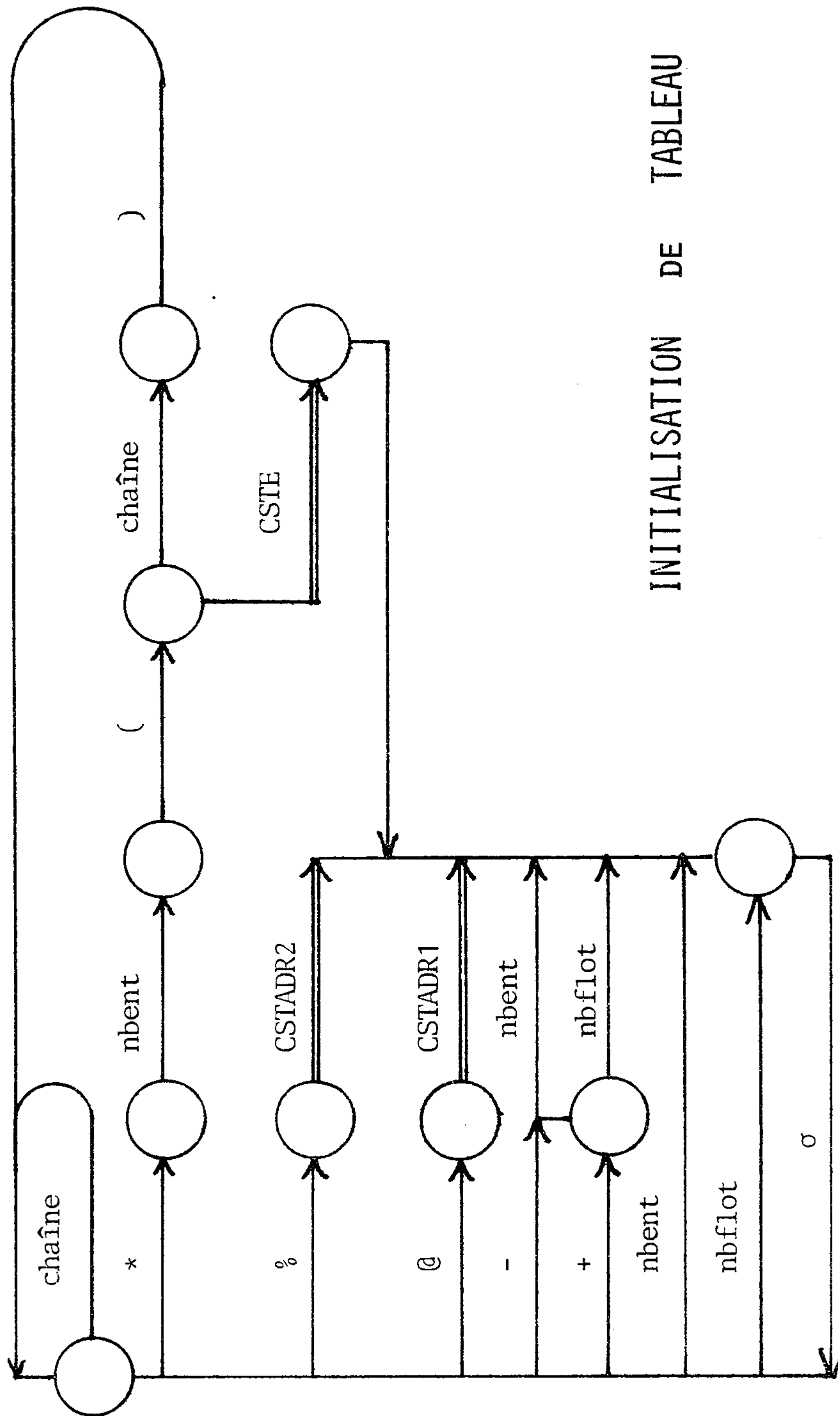


PLANCHE 4

DCL2

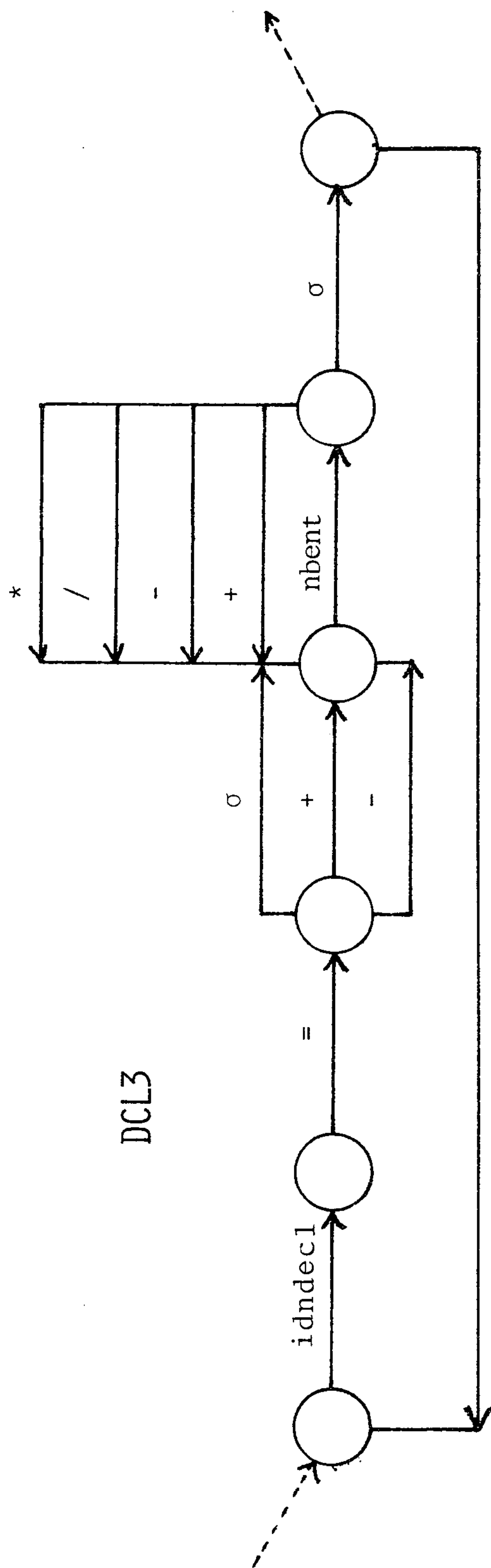
nbent
type
POINTER
type
idndecl
=
σ
Initialisa-
tion de
tableau
σ
σ
SYN
nbent
LOGICAL
idndecl
SYN
idvar
VAR4
σ
idptr
ipile
idlogic
σ
σ
=
valogic
σ
*
nbent
)
(
valogic

PLANCHE 5



INITIALISATION DE TABLEAU

PLANCHE 6



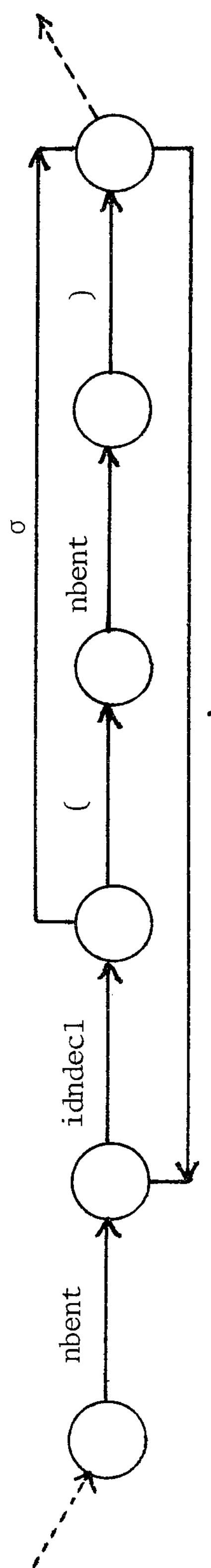
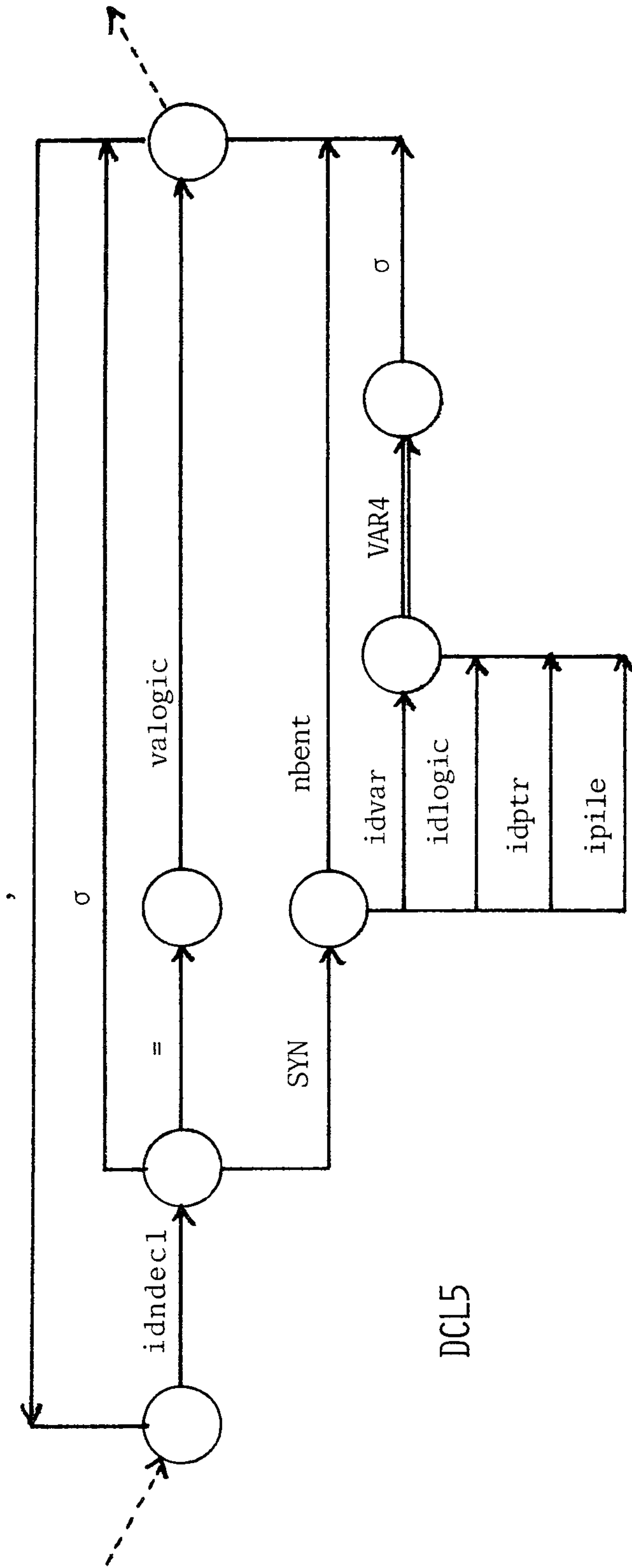
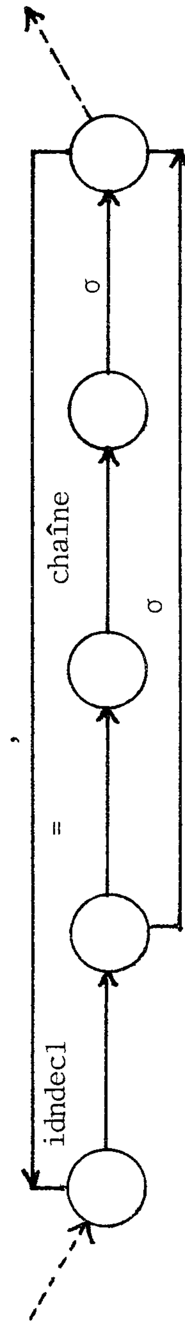


PLANCHE 7





DCL6

PLANCHE 8

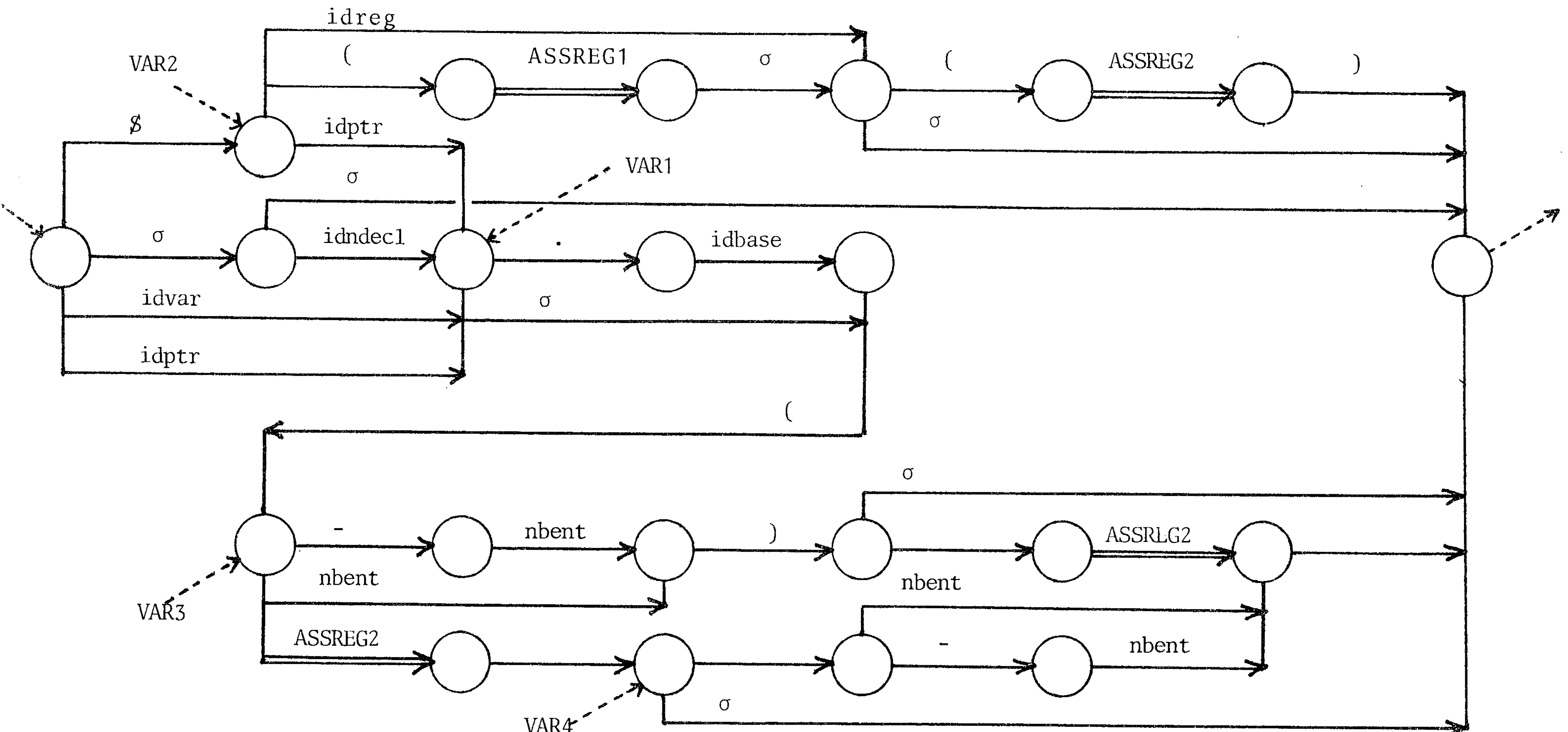


PLANCHE 9

CSTE
(CSTADR1, CSTADR2)

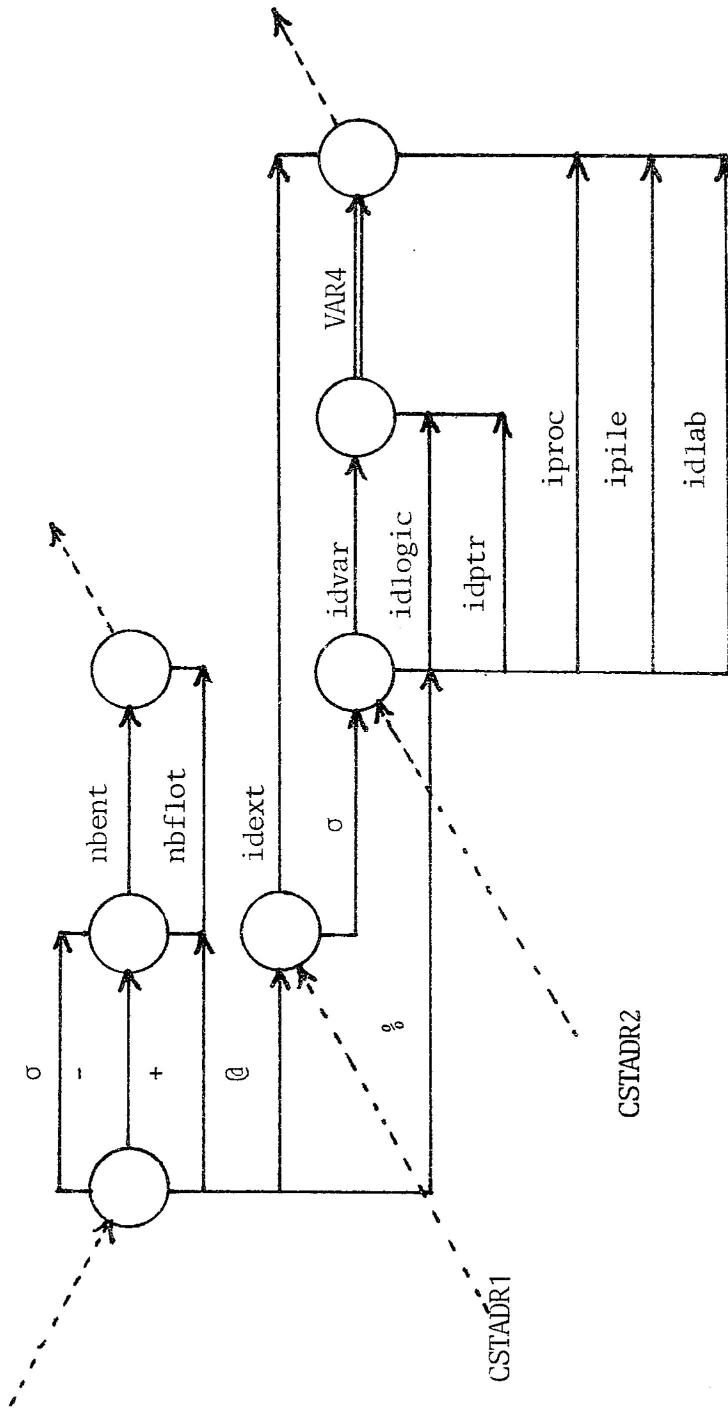
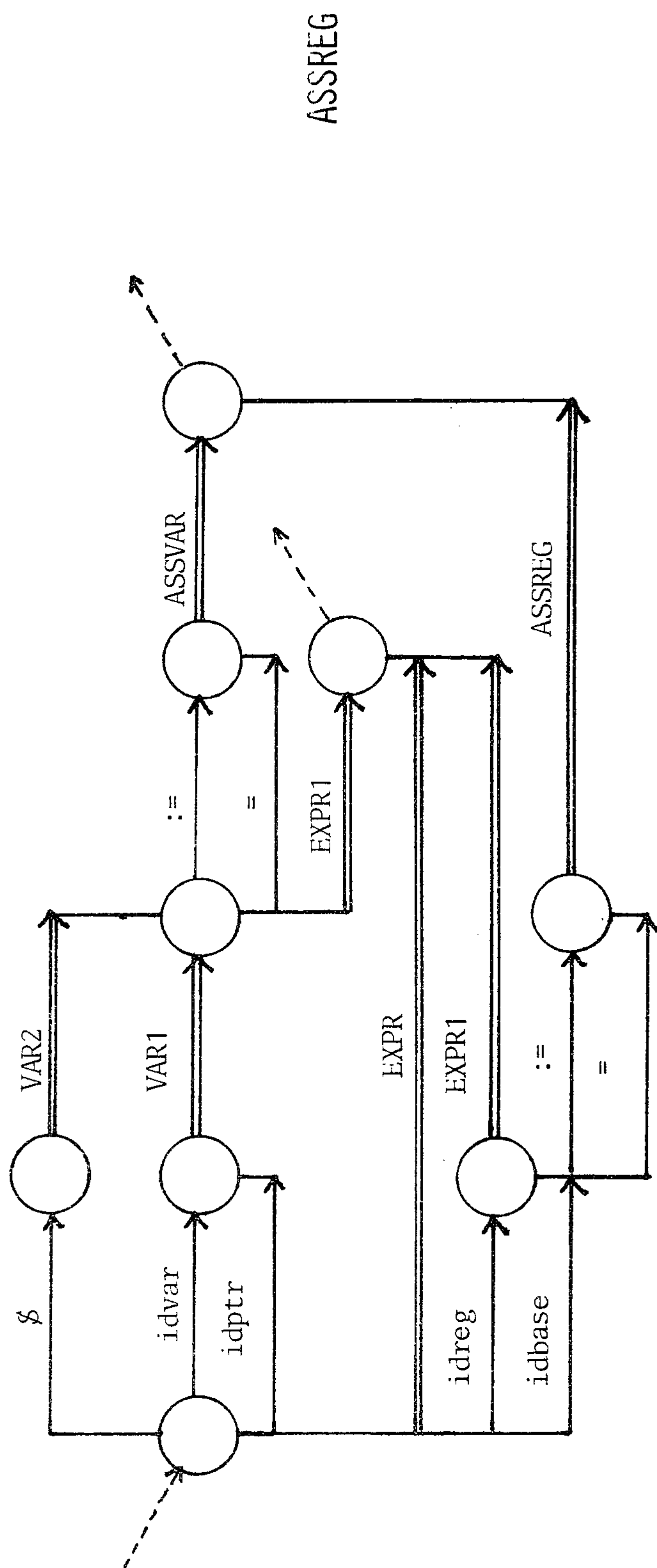
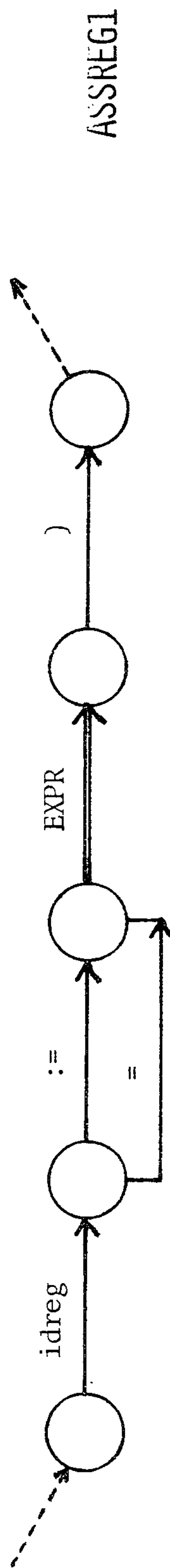


PLANCHE 10





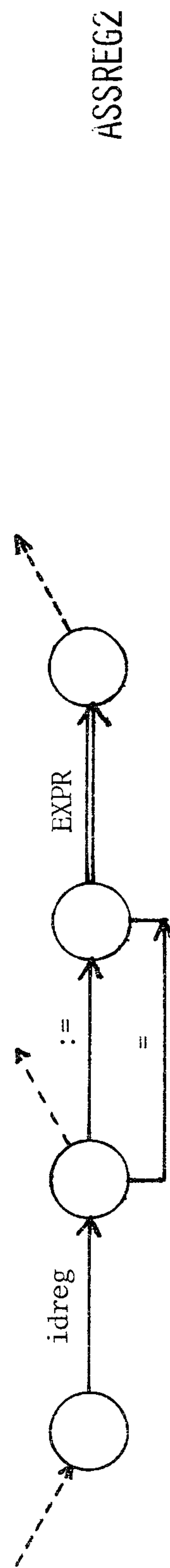
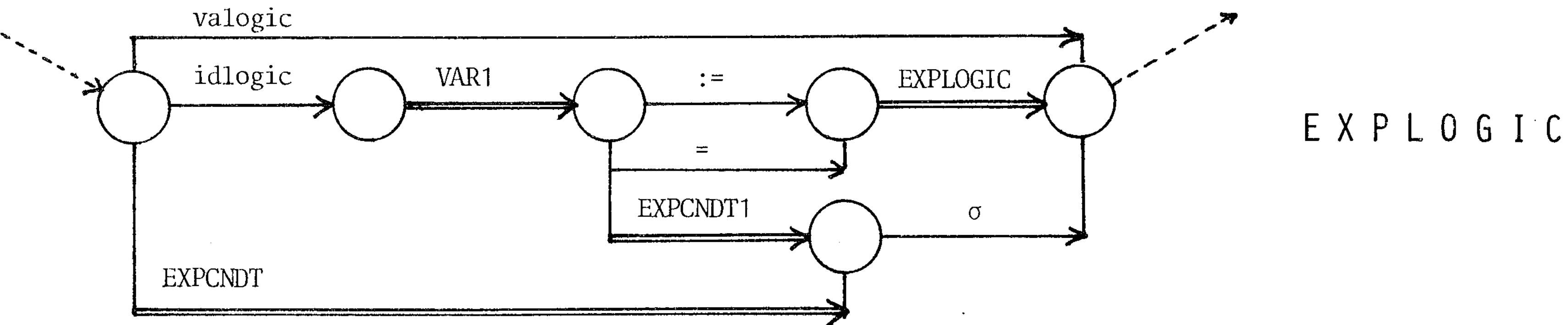


PLANCHE 11

EXPLOGIC



ASSVAR

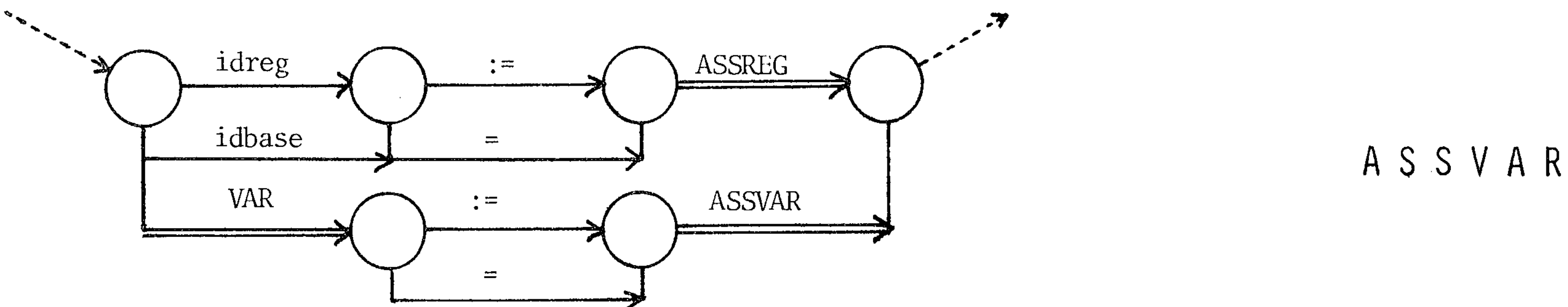


PLANCHE 12

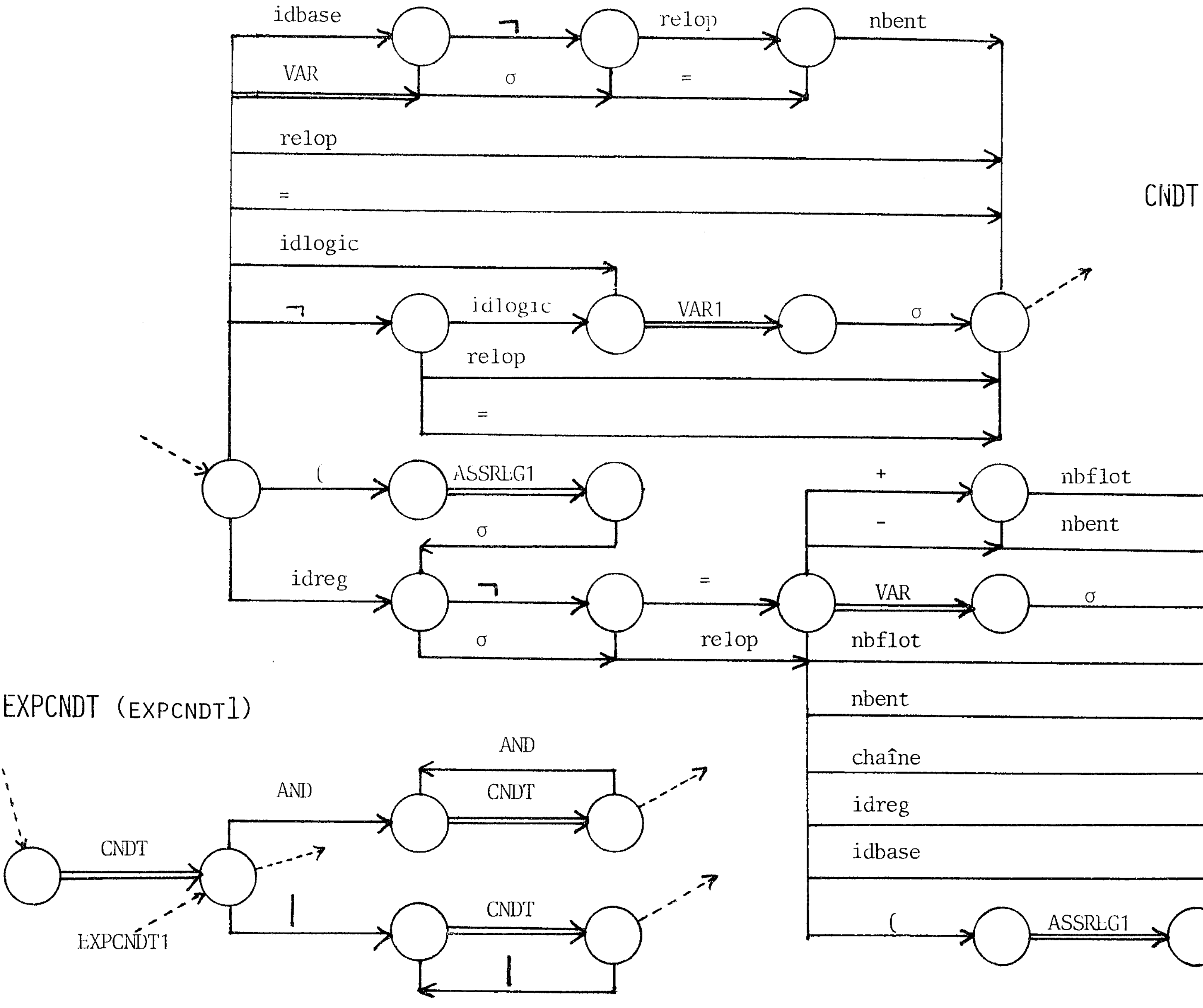
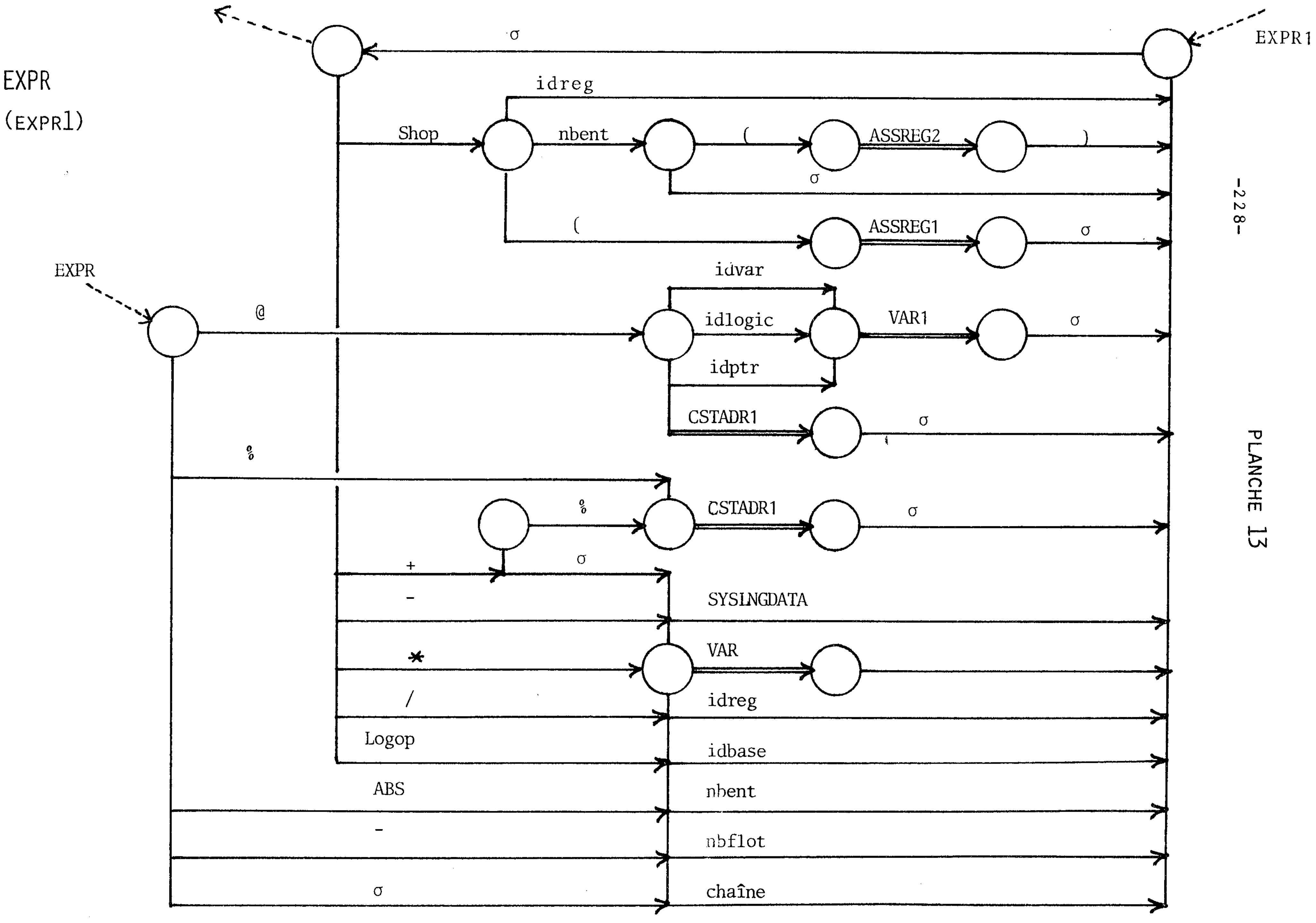


PLANCHE 13



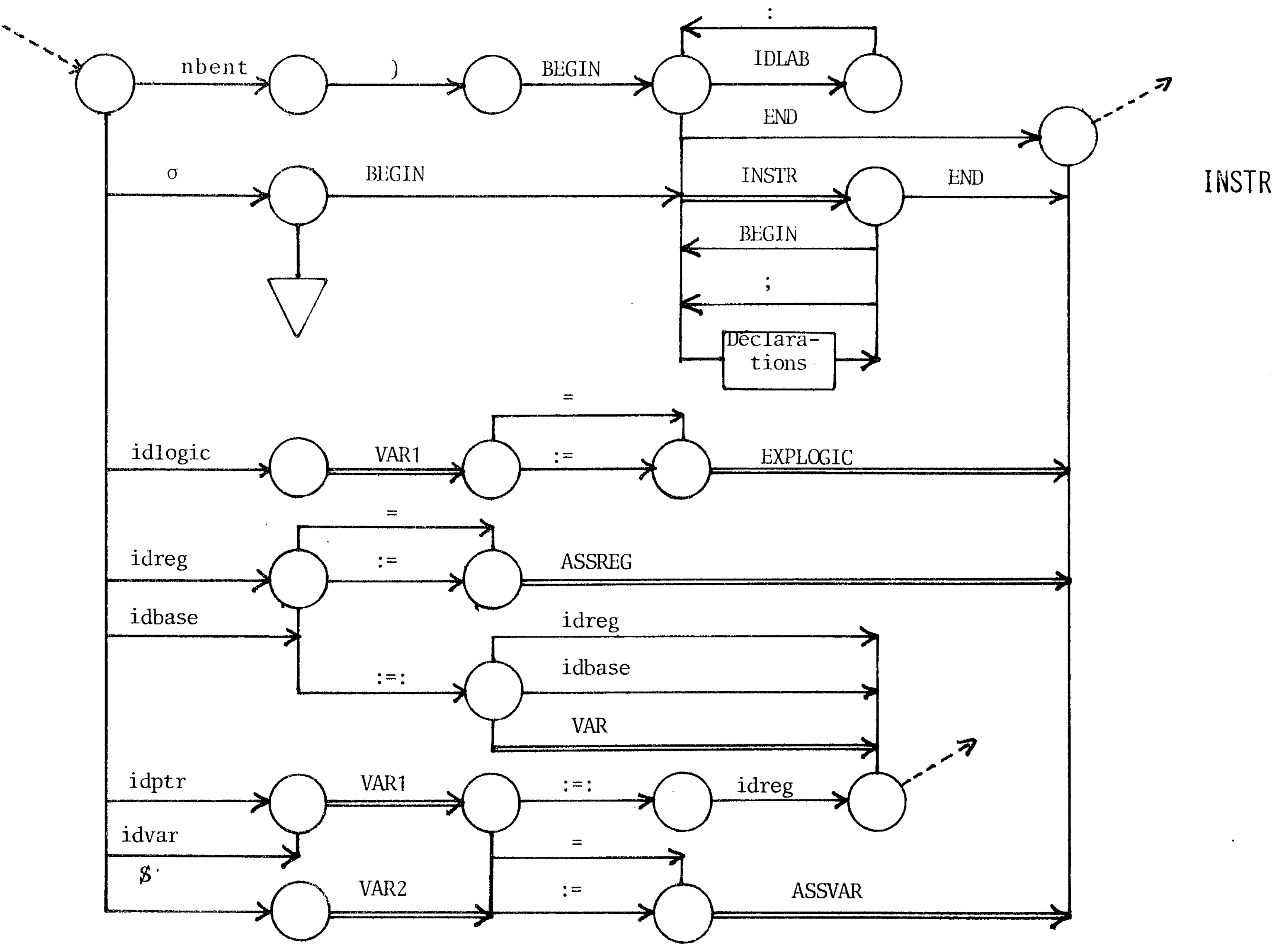
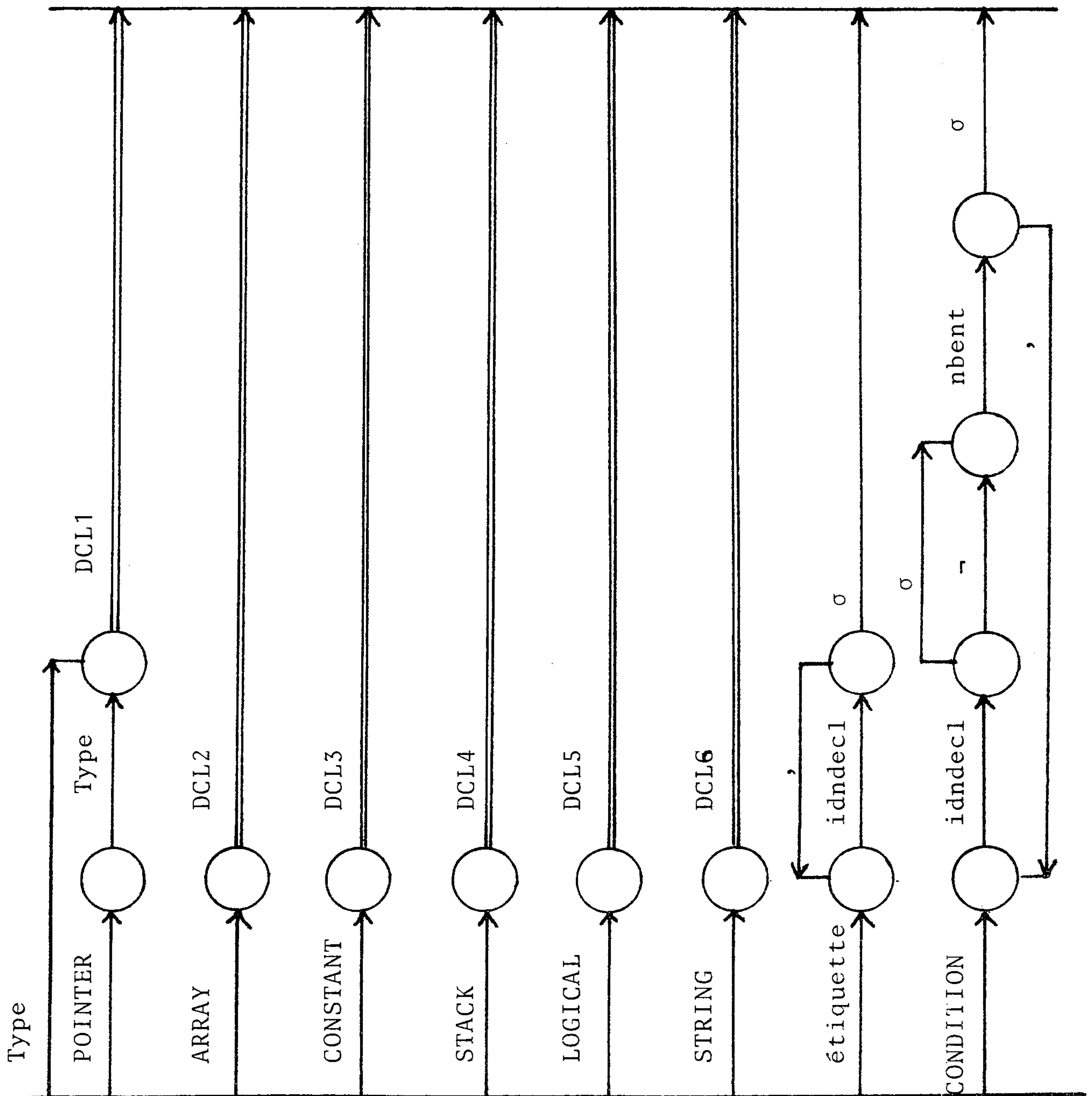

nbent
)
BEGIN
:
IDLAB
END
σ
BEGIN
INSTR
END
INSTR
BEGIN
;
Déclara-
tions
=
idlogic
VAR1
:=
EXPLOGIC
=
idreg
:=
ASSREG
idbase
idreg
:=:
idbase
VAR
idptr
VAR1
:=:
idreg
idvar
=
$'
VAR2
:=
ASSVAR

PLANCHE 15

DECLARATIONS



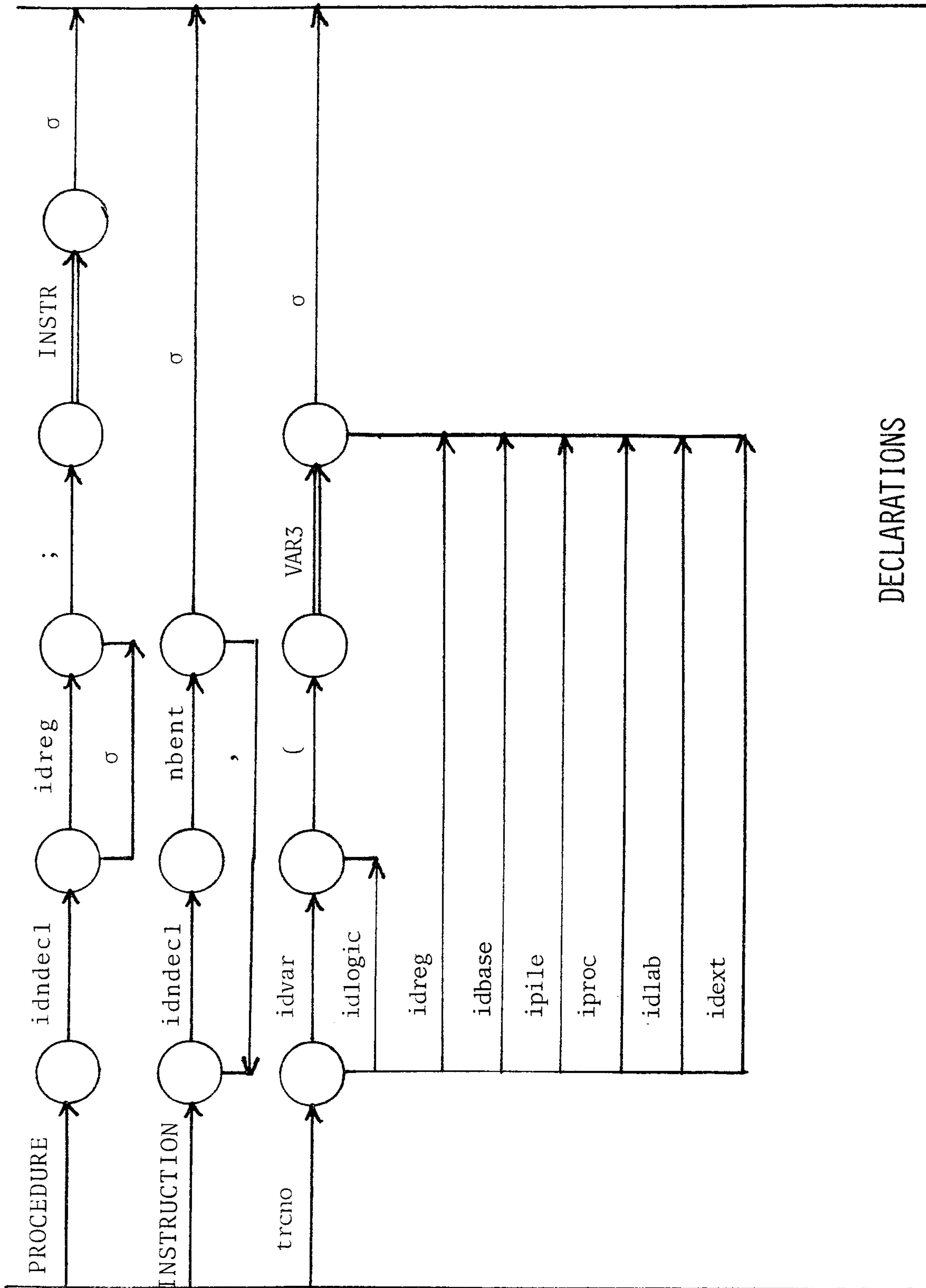

PROCEDURE
idndecl
idreg
σ
;
INSTR
σ
INSTRUCTION
idndecl
nbent
,
σ
trcno
idvar
idlogic
(
VAR3
σ
idreg
idbase
ipile
iproc
idlab
idext
DECLARATIONS

PLANCHE 17

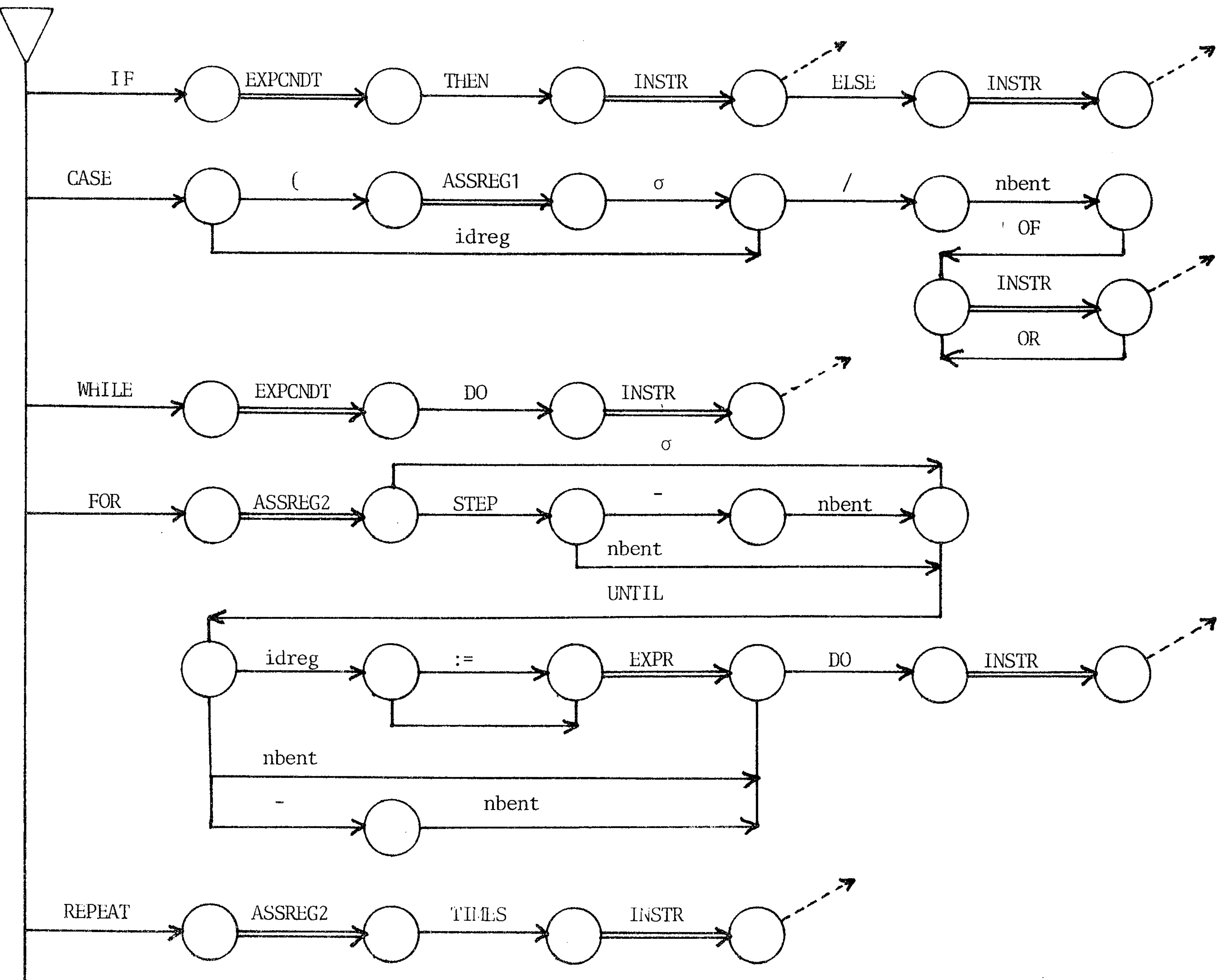


PLANCHE 18

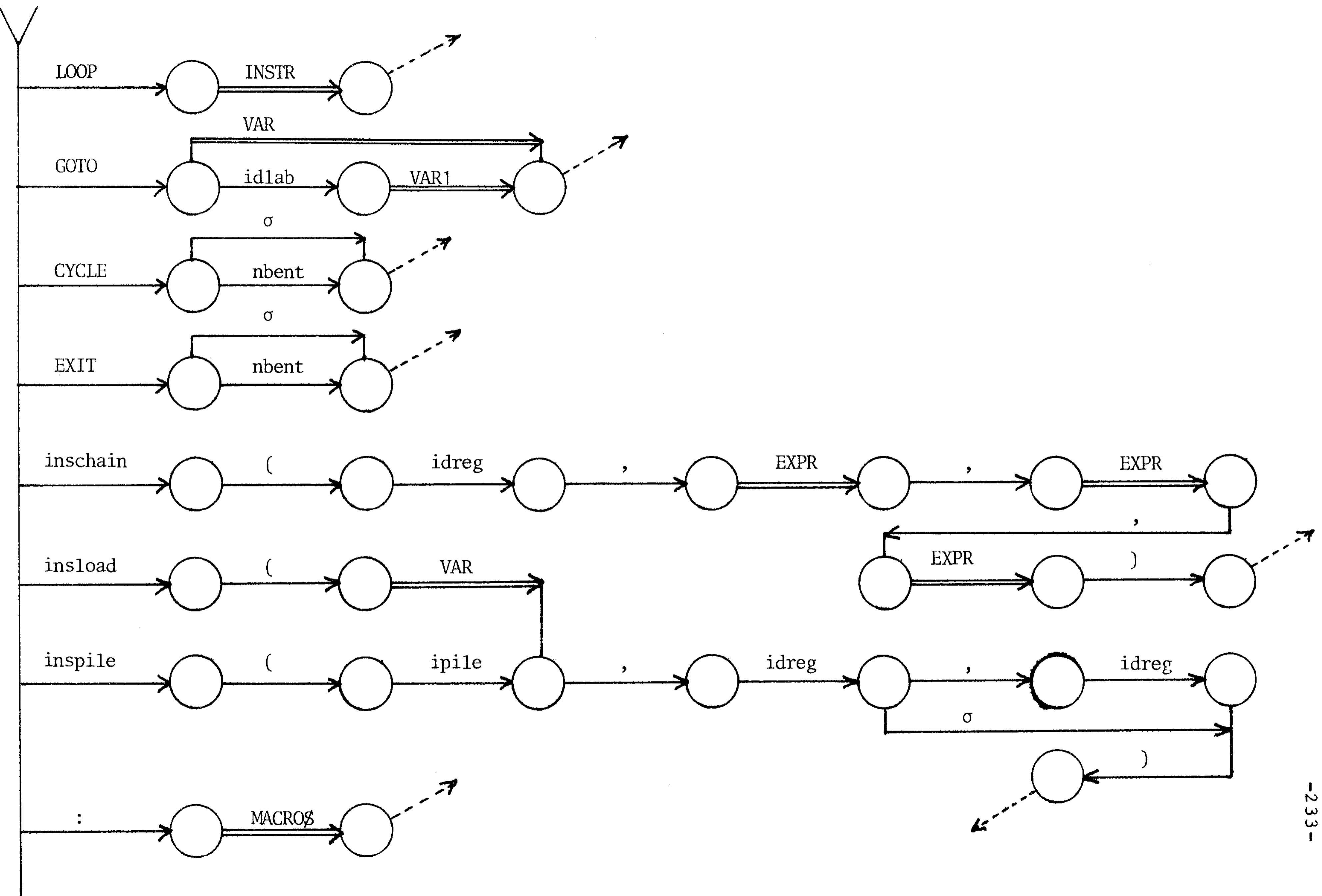
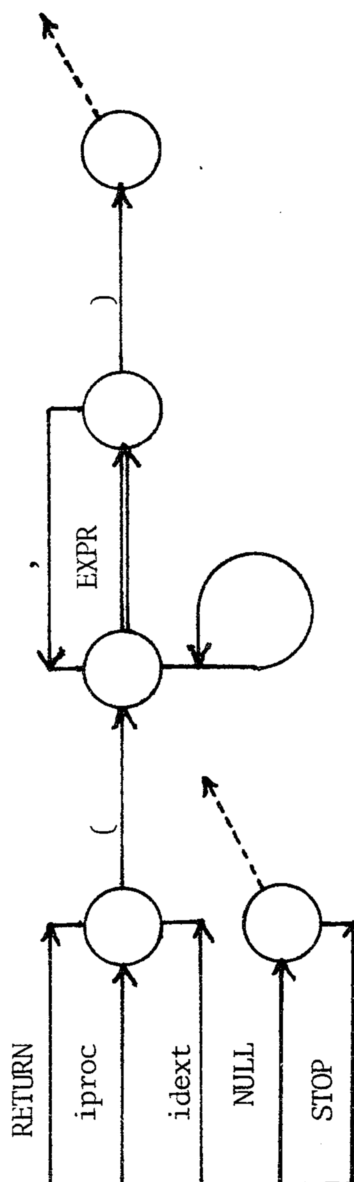


PLANCHE 19

idinst
(
idreg
idbase
nbent
+
-
nbent
,
VAR
iproc
idext
ipile
nbent
+
-
idlogic
idlab
σ
nbent
VAR1
)
silno
,
idreg
idvar
idbase
ipile
idlab
iproc
idext
idlogic
ALL
σ

PLANCHE 20



ANNEXE 2

LISTE DES FONCTIONS ASSEMBLEUR PREDECLAREES

Fonction	Code
AIO	#6E
ANLZ	#44
CAL1	#04
CAL2	#05
CAL3	#06
CAL4	#07
CBS	#60
CLM	#19
CLR	#39
CS	#45
CVA	#29
CVS	#28
DA	#79
DC	#7D
DD	#7A
DL	#7E
DM	#7B
DS	#78
DSA	#7C
DST	#7F
EBS	#63
EXU	#67
HIO	#4F
INT	#6B
LAS	#26
LBR	#27
LCF	#70
LCFI	#02
LIAI	#01
LPSD	#0E
LRA	#2C
LRP	#2F
LSTP	#2F
LVAW	#34
MBS	#61
MMC	#6F
MTB	#73
MTH	#53
MTW	#33
PACK	#76
PLS	#0C
PSS	#0D
RDIRECT	#6C
RIO	#4F
SIO	#4C
STCF	#74
TBS	#41
TDV	#4E
TIO	#4D
TTBS	#40
UNPK	#77
WAIT	#2E
WDIRECT	#6D
XPSD	#0F

A N N E X E 3

LISTE DES UNITES SYNTAXIQUES ET DES IDENTIFIEURS PREDECLARES

01	identifieur non déclaré
02	nombre entier
03	nombre flottant
04	chaine de caractères
05	identifieur de registre
06	identifieur de base
07	identifieur de variable
08	identifieur de pointeur
09	identifieur d'étiquette
0A	identifieur de procédure
0B	identifieur de procédure externe
0C	identifieur de pile
0D	identifieur d'instruction (voir annexe 2)
0E	BYTE
	SHORT
	WORD
	LONG
	REAL
	LONGREAL
0F	SLLS
	SRLS
	SLLD
	SRLD
	SLCS
	SRCS
	SLCD
	SRCD
	SLAS
	SRAS
	SLAD
	SRAD
	SLFS
	SRFS
	SLFD
	SRFD

Code	Élément
10	identifieur de condition
	<
	>
	>=
	<=
11	SILENCE
	NOSILENCE
12	identifieur de logique
13	TRUE
	FALSE
14	TRACE
	NOTRACE
15	MOVE
	COMP
	TRANS
16	PUSH
	PULL
17	LOAD
	STORE
18	GLOBALDATA
	COMMONDATA
	DUMMYDATA
19	LABEL
	ENTRY
1A	SYSPTRPROG
	SYSPTRDATA
	SYSPTRCARD
1B	RETURN
1C	AND
1D	!
1E	OR
1F	XOR
20	=
21	ABS
22	+
23	-
24	*
25	/
26	^
27	LOGICAL
28	STRING
29	MAIN
2A	SEGMENT
2B	EXTERNAL
2C	.
2D	PROCEDURE
2E	(
2F	,
30	NODATA
31	REENT
32	)
33	;
34	POINTER
35	ARRAY
36	CONSTANT
37	STACK
38	SYN
39	à
3A	%
3B	:=
3C	SYSLNGDATA
3D	$
3E	BEGIN
3F	END
40	INSTRUCTION
41	CONDITION
42	:
43	IF
44	CASE
45	LOOP
46	WHILE
47	REPEAT
48	FOR
49	GOTO
4A	EXIT
4B	CYCLE
4C	NULL
4D	STOP
4E	SNAP
4F	GEN
50	ELSE
51	:=:
52	THEN
53	OF
54	DO
55	STEP
56	UNTIL
57	TIMES
58	ALL

ANNEXE 4

AUTOMATE DE L'ANALYSEUR LEXICOGRAPHIQUE

L'automate possède 7 sorties :

1 - Identifieur
2 - Symbole
3 - Chaîne
4 - Nombre entier court
5 - Nombre entier long
6 - Nombre flottant court
7 - Nombre flottant long

<u>NOTATION</u> : FDL : caractère marqueur de fin de ligne
SS : symbole simple.

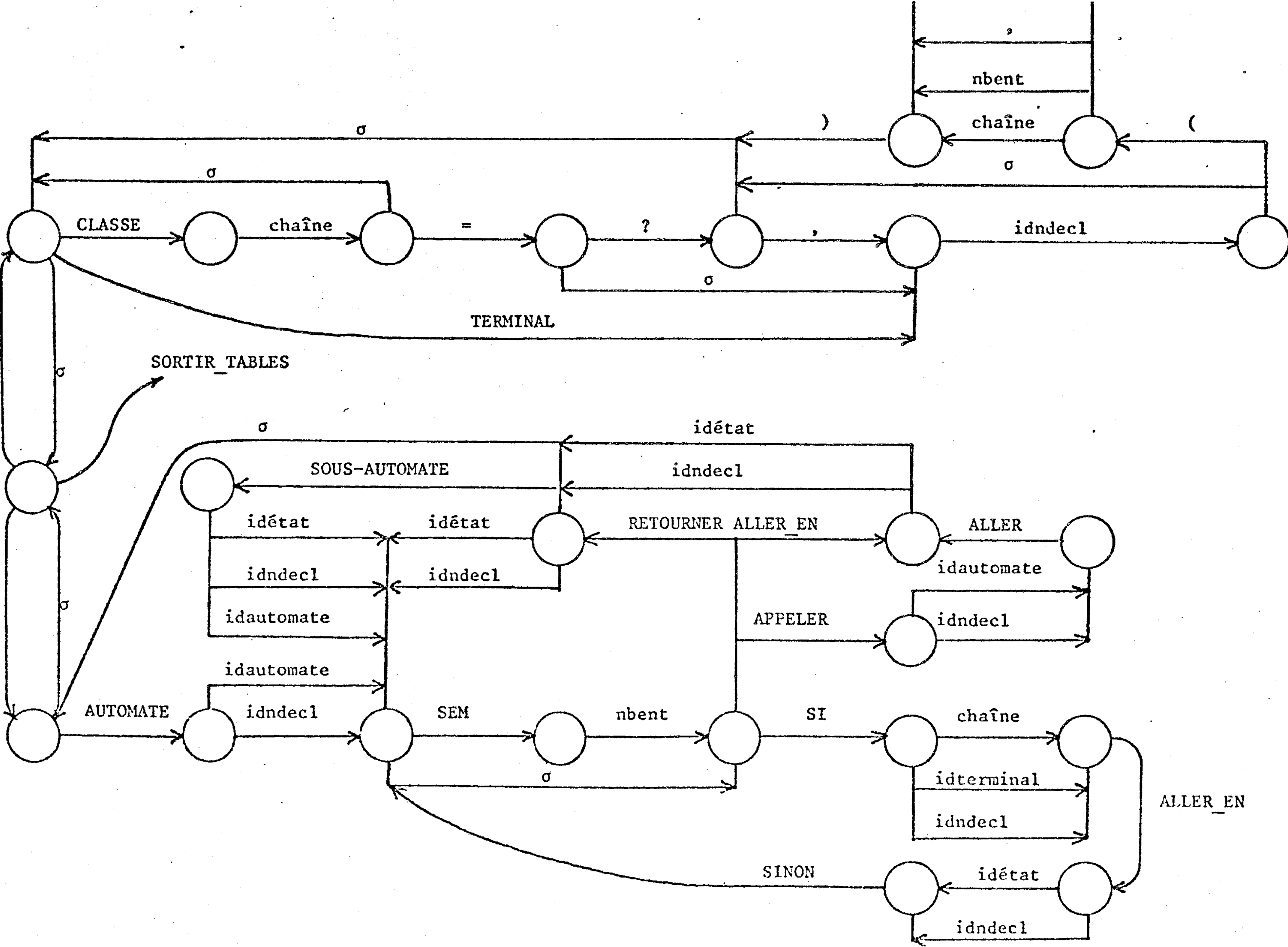
chiffre hexadécimal
tout caractère sauf "
#
lettre
FDL
L
L
chiffre
lettre, chiffre,_,
"
"
SS
chiffre
chiffre
≥ ≤
=
:
.
.
2
chiffre
chiffre ±
L
L
chiffre
:
=
0
1
2
3
4
5
6
7
8
9
A
B
C
D
E
F

A N N E X E 5

AUTOMATE DU GENERATEUR DE TABLES

NOTATION :

nbent	:	nombre entier
chaîne	:	chaîne de caractères
idndecl	:	identifieur non déclaré
idétat	:	identifieur apparu après le verbe ALLER-EN ou déclaré comme identifieur d'état.
idautomate	:	identifieur apparu après le verbe APPELER ou après les déclarations AUTOMATE et SOUS-AUTOMATE.



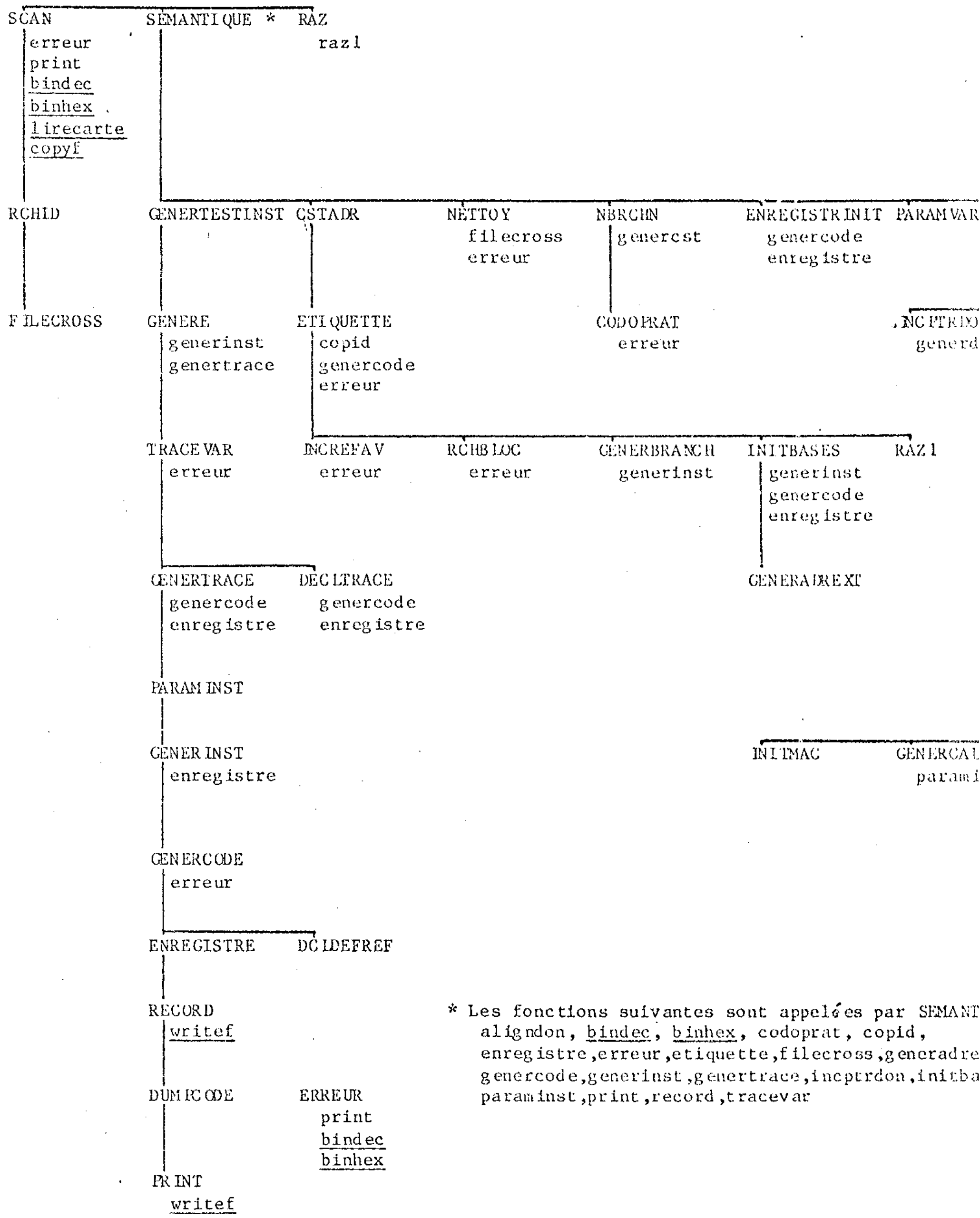
DECLARATIONS
DESCRIPTIONS
σ
CLASSE
chaîne
=
?
,
idndecl
(
)
chaîne
nbent
TERMINAL
SORTIR_TABLES
AUTOMATE
idndecl
idautomate
idétat
SOUS-AUTOMATE
RETOURNER ALLER_EN
ALLER
APPELER
SEM
nbent
SI
idterminal
ALLER_EN
SINON

A N N E X E 6

GRAPHE DE DÉPENDANCE DU SEGMENT COMPILATION

NOTATION :

Les identifieurs de procédure soulignés désignent des procédures externes au segment.
Les niveaux de profondeur des procédures correspondent aux niveaux sur le graphe. En minuscules figurent des procédures appelées, définies dans d'autres branches.



SCAN
erreur
print
bindec
binhex
lirecarte
copyf
SEMANTIQUE *
RAZ
razl
RCHID
FILECROSS
GENERTESTINST
CSTADR
NETTOY
filecross
erreur
NBRCHN
genercst
ENREGISTRINIT
genercode
enregistre
PARAMVAR
GENERE
generinst
genertrace
ETIQUETTE
copid
genercode
erreur
CODOPRAT
erreur
TRACEVAR
erreur
INCREFAV
erreur
RCHBLOC
erreur
GENERBRANCH
generinst
INITBASES
generinst
genercode
enregistre
RAZ1
GENERTRACE
genercode
enregistre
DECLTRACE
genercode
enregistre
GENERADREXT
PARAMINST
GENERINST
enregistre
INITMAC
GENERCODE
erreur
ENREGISTRE
DCLDEFREF
RECORD
writef
DUMPCODE
ERREUR
print
bindec
binhex
PRINT
writef
* Les fonctions suivantes sont appelées par SEMANT
aligndon, bindec, binhex, codoprat, copid,
enregistre, erreur, etiquette, filecross, generadre
genercode, generinst, genertrace, incptrdon, initba
paraminst, print, record, tracevar

N
on

PILEDCL
generdon
erreur

ALIGNDON

GENERDON
enregistre
genercode
erreur

ENREGISTRINIT
genercode
enregistre

STANDARDS
copid

IDENT_REGISTRE

COPID
erreur

OPENTRC

GENERCST
erreur

REENTRANCE

INITDCL
generdon

ALTREGS
erreur

DEBSEG
genercode
enregistre

GENERTESTDUMMY
copid
genercode

FONCTIONS SEMANTIQUES DES MACRO-INSTRUCTIONS SIRIS 8

OUT
nst

SAVEREGJTCB
paraminst

FINOPTION
codoprat

VALIDITE
erreur

TESTLOADREG

LOADADR
generinst

LOADADRFPI
generinst

CHAINMOT

EXCHANGARG

DECLWORD
generdon
incptrdon

IQUE:

xt,
se,

BIBLIOGRAPHIE

Normes de Programmation SIRIS7/SIRIS8 versions B09/C09
Document technique CII-HB

Procédures systèmes SIRIS7/SIRIS8 versions B09/C09
Document technique CII-HB

Langage de commande SIRIS7/SIRIS8 versions B09/C09
Document technique CII-HB

Aide à la mise au point SIRIS7/SIRIS8 versions B09/C09
Document technique CII-HB

Editeur de liens SIRIS7/SIRIS8 versions B09/C09
Document technique CII-HB

Ordinateur IRIS80
Document technique CII-HB

Langage de programmation LP70 sous SIRIS7/SIRIS8
Document technique CII-HB

Le langage PL360
Daniel SUTY - C.I.C.G.

Compiler construction for digital computers
David GRIES -

D.E. KNUTH
The Art of computer programming
Adisson Wesley Reading