DESCRIPTION ET UTILISATION

DU LANGAGE DE PROGRAMMATION

DE SYSTEMES SPIP

Daniel Thalmann

CENTRE UNIVERSITAIRE
D'INFORMATIQUE

24, rue Général Dufour
1211 GENEVE 4
Tél. 20 93 33

Avril 1977

CUI/77/2

DESCRIPTION ET UTILISATION DU LANGAGE DE PROGRAMMATION DE SYSTEMES SPIP

Daniel Thalmann

CENTRE UNIVERSITAIRE
D'INFORMATIQUE

24, rue Général Dufour
1211 GENEVE 4
Tél. 20 93 33

Avril 1977

CUI/77/2

Table des matières

		Page
1.	Introduction	1
	1.1 Les principes de SPIP	1
	1.2 Les différentes parties de SPIP	1
2.	La machine λ	2
	2.1 L'unité centrale	2
	2.1.1 Les registres et les structures	2
	2.1.2 L'accès à la mémoire	3
	2.1.3 Les modules	3
	2.2 Les entrées-sorties	4
	2.2.1 Les périphériques	4
	2.2.2 Le système d'interruption	4
	2.2.3 Le jeu d'instructions	4
3.	Champs des instructions et opérandes	4
	3.1 Les champs-mot	4
	3.2 Les champs-byte	4
	3.3 Structures considérées globalement	6
	3.4 Chaînes de caractères	7
	3.5 Modules	7
	3.6 Périphériques	8
	3.7 Mots-clé	8
4.	Les instructions simples	9
	4.1 Les instructions de transfert	9
	4.1.1 Transfert entre registres et mémoire	9
	4.1.2 Transfert avec conversion	9
	4.2 Les opérations élémentaires	10
	4.2.1 Opérations arithmétiques	10
	4.2.2 Décalage	10
	4.2.3 Complémentations logique et arithmétique	11
	4.2.4 Incrémentation, décrémentation et effacement	11
	4.3 Maniement des bytes	12
	4.3.1 Composition et décomposition	12
	4.3.2 Transfert d'un byte dans un mot	12
	4.3.3 Echange de bytes	13
	4.4 Accès aux bits	13
	4.5 Echange de pages	14
5.	Les instructions de contrôle	14
	5.1 Conditions	14
	5.1.1 Comparaison d'objets	14
	5.1.2 Test de bits	15
	5.1.3 Test de catégorie	15
	5.2 Instructions sélectives	16
	5.2.1 Instruction conditionnelle	16
	5.2.2 Instruction de choix	17
	5.3 Instructions répétitives	18
	5.3.1 Instruction "while"	18
	5.3.2 Instruction "repeat"	19
	5.3.3 Instruction "loop"	19
	5.3.4 Sorties de boucles	20

6. Le traitement des structures 21
6.1 Piles et queues 21
6.2 Tables 22
6.3 Chaînes 23
6.4 Longueur des structures 26
6.5 Libération des structures 26

7. Les entrées-sorties 26
7.1 Lecture et écriture 26
7.1.1 Transferts simples 27
7.1.2 Transferts complexes 28
7.2 Recherche 29
7.3 Test de fin de données 29
7.4 Transferts disque-mémoire 29
7.5 Entrées-sorties au niveau fondamental 30

8. Le traitement des interruptions 31
8.1 Définition d'un module de service 31
8.2 Le sauvetage et la restauration de l'état du processeur 31
8.3 L'enclenchement du système d'interruption 31
8.4 La reconnaissance des périphériques qui ont causé l'interruption 31

9. Le traitement des fichiers 32
9.1 Instructions standards 32
9.2 Instructions spécifiques 33
9.2.1 Ouverture et fermeture des fichiers 33
9.2.2 Fin de fichier et repositionnement 33

10. Les modules 34
10.1 Définition des modules 34
10.1.1 Liste des paramètres formels 34
10.1.2 Déclarations 34
10.1.3 Corps d'un module 34
10.2 Utilisation des modules 34
10.3 Prédéfinition de modules 35
10.4 Exemples de modules 36
10.4.1 Module non récursif 36
10.4.2 Module récursif 36
10.5 Branchement conditionnel 36
10.6 Modules standards 37
10.7 Les instructions stop, return-monitor et abort 38

11. Les requêtes 38
11.1 Restrictions et extensions au niveau d'un moniteur 38
11.2 Définition d'une requête 39
11.2.1 Paramètres 39
11.2.2 Paramètres, mots et bytes 39
11.2.3 Unités logiques passées comme paramètres 40
11.3 Appel à une requête 41
11.4 Requêtes permanentes 41

12. Les pseudo-instructions 41
12.1 Déclarations de structures 41
12.1.1 Type des éléments 41
12.1.2 Réservation 42
12.2 Registres rapides 42

12.3 Registres volatils 43
12.4 Objets locaux 43
12.5 Priorités d'accumulateurs 44
12.6 Options 44
12.7 Début et fin d'un programme 45
12.8 Adresse de départ 46

13. Les macros et les synonymes 46
13.1 Macros sans paramètre 46
13.1.1 Définition d'une macro 46
13.1.2 Utilisation d'une macro 46
13.2 Macros avec paramètres 47
13.2.1 Définition d'une macro 47
13.2.2 Utilisation d'une macro 47
13.3 Synonymes 48
13.3.1 Pseudo-instruction name 48
13.3.2 Pseudo-instruction sequence 50

Bibliographie

Annexe A : SPIP et O.S. 1100

Annexe B : Erreurs
B.1 Erreurs détectées par le compilateur SPIP
B.2 Erreurs détectées par les procédures de génération de code (erreurs internes)
B.3 Erreurs détectées par l'éditeur de liens
B.4 Erreurs à l'exécution

Annexe C : Limitations du compilateur SPIP

Annexe D : Objets prédéfinis

1. Introduction

L'écriture de logiciel en langage d'assemblage pour un mini-ordinateur est un travail fastidieux. Les algorithmes sont masqués par la pauvreté du langage; les modifications sont pénibles à implanter et sont souvent la source d'erreurs; on assiste ainsi à une dégradation des systèmes, écrits en langage d'assemblage; les programmes sont rarement documentés et la portabilité est inexistante.

Nous avons voulu montrer qu'il est possible d'utiliser la puissance d'une grosse machine et un langage de haut niveau adéquat pour produire un système d'exploitation pour plus d'un mini-ordinateur.

1.1 Les principes de SPIP

SPIP est un compilateur de systèmes d'exploitation implanté sur une machine hôte, en général un gros ordinateur. On écrit dans une première étape un système d'exploitation pour une machine abstraite possédant un jeu d'instructions très riche : la machine λ. Le système-objet produit par SPIP est un système d'exploitation prêt à fonctionner sur le mini-ordinateur choisi. Une simple option dans SPIP permet de sélectionner la machine-objet. A l'heure actuelle, 2 options sont implantées : option N pour Data General Nova et option P pour DEC PDP-11.

1.2 Les différentes parties de SPIP

SPIP se compose de 4 parties (voir figure 1.2) :

1. Un compilateur accepte un programme écrit dans le langage symbolique SPIP et produit comme langage-objet, un code indépendant du mini-ordinateur : le code λ.

2. Un ensemble de procédures traduit le code λ en code relogeable spécifique au mini-ordinateur choisi. Malgré la dépendance du hardware, environ 80 % des procédures restent valables pour tous les mini-ordinateurs. C'est à ce niveau que s'effectue une optimisation conséquente.

3. Un éditeur de liens produit un code absolu pour le mini-ordinateur. Une optimisation de l'occupation mémoire est encore assurée par cette partie. Le programme d'édition de liens est indépendant du mini-ordinateur choisi.

4. Un simulateur de chaque mini-ordinateur permet l'interprétation du code absolu sur la machine hôte.

NB. : on trouvera dans la bibliographie les références traitant les aspects d'organisation et d'implantation de SPIP.

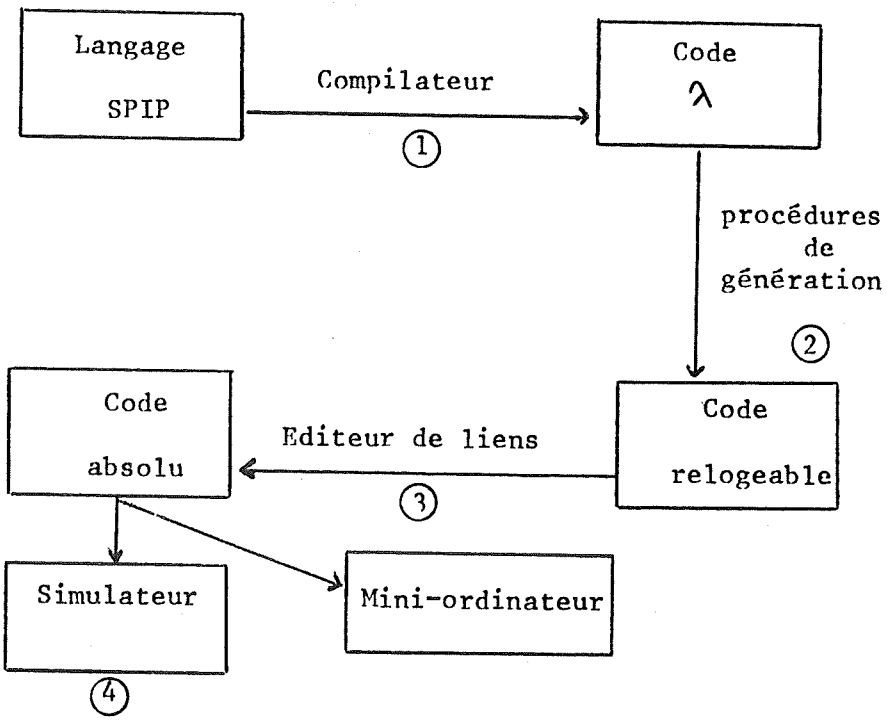


Figure 1.2

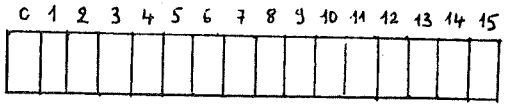
2 La machine λ.

2.1 L'unité centrale

C'est un processeur traitant des mots de 16 bits et des bytes de 8 bits. Il est connecté à une mémoire adressable indirectement, par registres d'index ou par page. Le processeur permet l'accès à divers périphériques et possède un système d'interruption.

2.1.1 Les registres et les structures

30 registres-mot de 16 bits, w1,w2,...,w30 sont utilisés comme registres d'opérandes, dans le transfert entre le processeur central et les périphériques, comme registres d'index et dans l'adressage par page.



10 registres-byte de 8 bits, b1,b2,...,b10 servent comme registres d'opérandes et dans le transfert entre le processeur central et les périphériques.

La grande puissance de la machine λ réside dans ses structures; en effet, les instructions du processeur peuvent manier directement 10 piles, s1,s2,...,s10, 10 queues, q1,q2,...,q10, 10 chaînes, c1,c2,...,c10 et 30 tables, t1,t2,...,t30. Les éléments composant ces structures peuvent être des mots et des bytes.

Nous allons définir ces 4 structures telles qu'elles sont implantées dans la machine λ.

Définitions

Une pile est une zone de mémoire dynamique, dans laquelle on peut accéder, ajouter ou retirer un élément à une de ses extrémités seulement.

Une queue est une zone de mémoire dynamique, dans laquelle on ne peut ajouter un élément qu'à une extrémité et n'en retirer un qu'à l'autre.

Une chaîne ou liste linéaire est une structure formée d'éléments constitués de 2 parties, une partie contenant l'information proprement dite, et l'autre partie un pointeur sur le prochain élément. Le premier élément de la chaîne est appelé tête de chaîne.

Une table est une zone de mémoire dans laquelle on peut accéder à n'importe quel élément par l'intermédiaire de l'adresse de l'élément appelé indice.

2.1.2 L'accès à la mémoire

Trois types d'adressage permettent d'accéder à la mémoire de la machine λ :

1. L'adressage indirect : l'adresse effective de l'opérande est supposée être le contenu de l'emplacement désigné par l'adresse spécifiée.

2. L'adressage indexé : on atteint un mot-mémoire par la valeur obtenue en additionnant le contenu d'un registre-mot et un déplacement positif.

3. L'adressage par page : on atteint un mot-mémoire par un déplacement relatif à la page spécifiée.

Les adressages peuvent être combinés.

Il faut encore signaler la possibilité d'accéder à des opérandes immédiats sans passer par un adressage.

Définition

Une page est une zone de mémoire de page-length mots. La mémoire est ainsi divisée en pages numérotées de 0 à N.

2.1.3 Les modules

Le processeur permet la définition et l'utilisation de 50 modules, m1,m2,...,m50. Les modules peuvent être définis récursivement, l'adresse de retour, les paramètres et les objets locaux sont sauvés automatiquement.

2.2 Les entrées-sorties

2.2.1 Les périphériques

Tous les transferts entre registres simples et structures avec les périphériques sont possibles.

Le disque est accessible par secteurs, les transferts se font entre page de la mémoire et secteur disque.

Le traitement de fichiers s'effectue par l'intermédiaire des unités logiques f1 à f10.

2.2.2 Le système d'interruption

La machine λ possède un système d'interruption qui permet d'interrompre le programme en cours d'exécution à la demande d'un équipement périphérique. Un branchement à un mode de traitement d'interruptions est effectué automatiquement. Des priorités d'interruptions peuvent être attribuées.

2.2.3 Le jeu d'instructions

Il sera présenté dans les chapitres suivants.

2 niveaux sont définis :

a) le niveau fondamental (moniteur)

b) le niveau secondaire (utilités, compilateurs ...)

Certaines instructions ne sont disponibles qu'à un seul niveau, cette restriction sera mentionnée au moment voulu.

Commentaires

Des commentaires peuvent être insérés dans le langage.

Forme : « ‹texte du commentaires› »

3. Champs des instructions et opérandes

Les champs des instructions du langage SPIP sont le plus fréquemment des mots ou des bytes. Ils peuvent encore être des modules, des périphériques ou des mots-clé. On distinguera 3 catégories de champs :

a) les champs sources : l'information est connue et ne peut être modifiée.

b) les champs destination : l'information peut être inconnue et est modifiée.

c) les champs source-destination (s-destination) : l'information est connue et est modifiée.

3.1 Les champs-mot

On peut donner la classification suivante :

a) Champs destination, s-destination et source

registres : w1,w2,... w30

adressage indexé : ind <déplacement> <registre>
le déplacement est un entier positif.
p.e. ind 5 w2

adressage par page : page <déplacement><numéro de page>
le déplacement et le numéro de page peuvent être des registres ou des entiers positifs.
p.e. page w1 w2 page 5 w3 page w4 0

adressage indirect : indirect <adresse>
l'adressage est indexé ou par page
p.e. indirect ind 5 w1
indirect page w4 w5

structures : piles, queues, chaînes : l'accès se fait conformément au type de structure

tables : non autorisé.

b) champs source

nombres entiers : positifs p.e. 17, 23, 52

négatifs p.e. -4, -10

octal p.e. 17b, 55b

Remarque : les nombres négatifs sont traités en complément à 2.

adresse d'un objet : l'objet est précédé du préfixe address.

p.e. address w4, address f1

Restriction : address n'est pas autorisé pour les structures.

longueur d'une structure : la structure est précédée du préfixe length (voir paragraphe 6.4)

p.e. length s2

3.2 Les champs-byte

On peut les classer de la manière suivante :

a) Champs destination, s-destination et source

registres : b1,b2,... b10

structures : comme pour les champs-mot, mais leurs composantes doivent être des bytes.

b) Champs source

caractères : tout le jeu de caractères ascii est disponible de 2 manières :

1. Par les caractères entre apostrophes précédés par les préfixes ctrl et/ou shift selon la terminologie "télétype".

   p.e. ctrl 'C' ctrl shift 'K' 'L' '$'

2. Par leur code ascii précédé de ".

   p.e. "101b "041

   NB : le code doit avoir 3 digits.

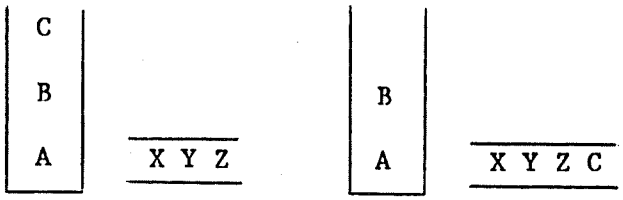
Note : quelques caractères ont un nom prédéfini (voir annexe D),

p.e. space.

3.3 Structures considérées globalement

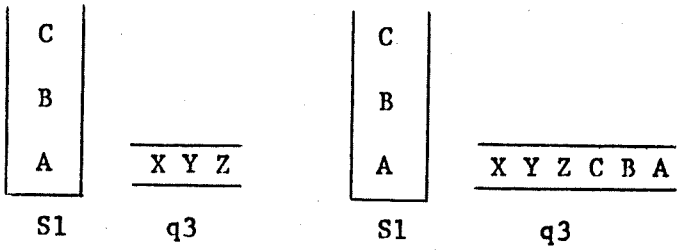
Il est possible de traiter tout le contenu d'une structure, dans ce cas elle doit être suivie d'une étoile lorsqu'il y a ambiguïté.

p.e. move(s1,q3) enlève le dernier mot sur la pile s1 et le place dans la queue q3,



avant après

move(s1*,q3*) copie le contenu de la pile s1 dans la queue q3 en respectant les règles d'accès.



avant après

3.4 Chaînes de caractères

Les chaînes de caractères sont délimitées par des apostrophes.

p.e. 'BONJOUR'

Certains caractères à l'intérieur ont une signification spéciale

' ne peut pas être utilisé.

est interprété comme un passage à une nouvelle ligne.

@ signifie que le caractère suivant est un shift caractère.

% signifie que le caractère suivant est un caractère de contrôle.

$ signifie que le caractère suivant est un ctrl shift caractère.

" signifie que les 3 caractères suivants doivent être interprétés comme un code ascii en octal.

p.e. move('MESSAGE SPECIAL#AAA@L$K,%M"101#BBB',output)

va afficher :

MESSAGE SPECIAL
AAA shift L ctrl shift K, ctrl M A
BBB

↑ caractère ascii 101

Remarque : pour imprimer ces caractères spéciaux, il faut les traiter par leur code.

3.5 Modules

On distinguera la définition et l'utilisation de modules.

a) Définition

Sont disponibles les modules m1 à m50.

b) Utilisation

Outre les modules m1 à m50, 3 modules standards peuvent être utilisés :

continue : ignore l'appel

halt : arrêt du processeur

system : donne le contrôle au moniteur du système.

3.6 Périphériques

Les périphériques d'entrée suivants sont reconnus :

input	(terminal)
cards	(lecteur de cartes)
disk	(disque)
f1 à f10	(unités logiques)

Pour la sortie, sont reconnus :

output	(terminal)
printer	(imprimante)
disk	(disque)
f1 à f10	(unités logiques)

Restrictions :

a) L'utilisation de disk est restreinte au niveau fondamental.

b) cards et printer sont reconnus à l'analyse syntaxique, mais leur utilisation suppose l'usage de drivers non implantés.

3.7 Mots-clé

La plupart des instructions SPIP ont des champs occupés par des mots-clé; ce sont des mots à signification particulière, dénotant généralement une action ou une opération.

Exemples : look, modulo, echo

4. Les instructions simples

4.1 Les instructions de transfert

4.1.1 Transfert entre registres et mémoire

L'instruction move permet le transfert d'une source à une destination.

Forme : move(<source>,<destination>)

Utilisation

a) copie d'un registre dans un autre

p.e. move(w1,w3) move(b1,b2)

b) transferts registre↔ mémoire

p.e. move(w3,ind 2 w3) move(page 2 w4, w5)

c) transferts mémoire

p.e. move(page 3 w1,ind 4 w2)

d) initialisations

p.e. move(3,w2) move('A',b1)

e) transferts d'éléments de piles et de queues

p.e. move(q2,s3) move(s4,w3) move(b1,q2)

f) transferts globaux

p.e. move('BONJOUR',q2*) move(s1*,q3*)

Ici le texte 'BONJOUR' est copié dans la queue q2, tandis que le contenu de la pile s1 est copié dans la queue q3.

4.1.2 Transfert avec conversion

Il existe une deuxième forme de l'instruction move permettant une conversion de type.

Forme : move(<source>,<type>,<destination>,<type>)

4 types sont disponibles : ascii, octal, decimal et binary.

Exemples :

a) move(w1,decimal,w2,octal)

Le contenu de w1, considéré comme un nombre décimal, est converti en octal et stocké dans le registre w2.

p.e. w1 | 0 000 000 000 011 000 | 24

w2 | 0 000 000 000 011 110 | 30

$24_{10} = 30_8$

b) move(q1,ascii,w1,binary)

Les caractères contenus dans la queue q1 sont interprétés comme une suite de bits, le nombre binaire calculé est transféré dans le registre w1.

Restrictions : les conversions ne sont pas autorisées pour des nombres négatifs.

4.2 Les opérations élémentaires

4.2.1 Opérations arithmétiques

L'instruction arithmetic permet d'effectuer une opération entre 2 opérandes, le résultat est un opérande.

Forme : arithmetic(<source>,<opérateur>,<source>,<destination>)

8 opérateurs sont disponibles :

+	: addition
-	: soustraction
*	: multiplication
/	: division
and	: et bit à bit
or	: ou bit à bit
xor	: ou exclusif bit à bit
modulo	: reste de la division

Exemples

a) arithmetic(w1,+,5,w2)

b) arithmetic(w1,and,77B,w3)

On isole les 6 bits de droite de w1, le résultat va dans w3.

4.2.2 Décalage

L'instruction shift permet de décaler un mot à gauche ou à droite d'un nombre de bits donné.

Forme : shift(<côté>,<source>,<nombre de bits>,<destination>)

Le côté est left (gauche) ou right (droite).

La source et la destination doivent être des opérandes-mot.

Le nombre de bits peut être constant ou variable, c'est un opérande source.

Exemples :

a) shift(left,w2,2,w5)

Le contenu du registre w2 est décalé de 2 bits à gauche et placé dans le registre w5.

b) shift(right,w2,w3,w2)

Le contenu du registre w2 est décalé à droite du nombre de bits contenu dans w3.

4.2.3 Complémentations logique et arithmétique

L'instruction complement permet de calculer le complément logique d'un opérande tandis que l'instruction negate fournit le complément arithmétique (complément à 2).

Formes : complement(<source>,<destination>)

negate(<s-destination>)

Exemples :

a) complement(w1,w2)

Le complément logique du contenu de w1 est placé dans le registre w2.

p.e.	w1	0 000 000 000 001 111	15
	w2	1 111 111 111 110 000	-16

b) negate(w1)

Le contenu de w1 est changé de signe

p.e.	w1	0 000 000 000 001 111	15 (avant l'instruction)
	w1	1 111 111 111 110 001	-15 (après l'instruction)

4.2.4 Incrementation, décrémentation et effacement

On peut incrémenter, décrémenter ou effacer le contenu d'un registre ou d'une adresse par les instructions increment, decrement et clear.

Formes : increment(<s-destination>)
decrement(<s-destination>)
clear(<s-destination>)

increment(w) est synonyme de arithmetic(w,+,1,w)

decrement(w) est synonyme de arithmetic(w,-,1,w)

clear(w) est synonyme de move(o,w)

Exemples :

a) increment(ind 3 w1)

Le contenu du mot d'adresse, obtenue en additionnant 3 au contenu de w1, est augmenté de 1.

b) decrement(page 2 w3)

On diminue de 1 le contenu du mot 2 de la page dont le numéro est contenu dans w3.

c) clear(w4)

Le contenu de w4 est effacé.

4.3 Maniement des bytes

Certaines instructions sont spécifiques au traitement des bytes. Elles complètent des instructions comme move aussi bien disponibles pour des opérandes-mots que pour des opérandes-bytes.

4.3.1 Composition et décomposition

On peut composer un mot, à l'aide de 2 bytes, par l'instruction compose et séparer un mot en 2 bytes, par l'instruction decompose.

Formes : compose(<source>,<source>,<destination>)
decompose(<source>,<destination>,<destination>)

Exemples :

a) compose(b3,'Z',w4)

On forme, dans w4 un mot composé du caractère contenu dans b3 et de 'Z'.

b) decompose(w5,b1,b2)

Le contenu de w5 est séparé en 2 bytes.

4.3.2 Transfert d'un byte dans un mot

L'instruction move-byte permet de transférer un byte dans un mot sans modifier la valeur de l'autre byte; l'instruction sert également à extraire un byte d'un mot.

Forme : move-byte(<côté>,<source>,<s-destination>)

Le côté désigne le byte transféré : left (gauche) ou right (droite).

Si la source est un byte, la destination doit être un mot; on introduit le byte dans le mot.

Si la source est un mot, la destination doit être un byte; on extrait le byte du mot.

Exemples :

a) move-byte(left,'K',w2)

Le caractère 'K' est placé dans le byte de gauche de w2, le byte de droite n'est pas affecté.

p.e. w2 | W | Z | avant l'instruction

w2 | K | Z | après l'instruction

b) move-byte(right,w6,b1)

Le byte de droite de w6 est transféré dans b1, w6 n'est pas affecté.

4.3.3 Echange de bytes

L'instruction swap-bytes échange les 2 bytes d'un mot.

Forme : swap-bytes(<source>,<destination>)

Exemples :

a) swap-bytes(w1,w2)

Le registre w2 contiendra les 2 bytes de w1 échangés.

p.e. w1 | A | B | w2 | S | K | avant l'instruction

w1 | A | B | w2 | B | A | après l'instruction

b) swap-bytes(s1,s1)

Les 2 bytes au sommet de la pile s1 sont échangés.

p.e. avant l'instruction

S1	E	F
	C	D
	A	B

après l'instruction

S1	F	E
	C	D
	A	B

4.4 Accès aux bits

N'importe quel bit d'un mot peut être mis à 0 ou à 1 par l'instruction oper-bit.

Forme : oper-bit(<opération>,<bit>,<s-destination>)

Si l'opération est on, le bit est mis à 1; si l'opération est off, le bit est mis à 0.

Le bit est spécifié par bit0, bit1, ... bit15.

Exemples :

a) oper-bit(on,bit3,w1)

Le bit 3 de w1 est mis à 1.

b) oper-bit(off,bit0,q2)

Le bit 0 du dernier mot placé dans la queue q2 est mis à 0.

4.5 Echange de pages

L'instruction swap-pages échange les informations de 2 pages de la mémoire, les 2 pages sont référencées par des opérandes contenant leur numéro.

Forme : swap-pages(<source>,<source>)

Exemples :

a) swap-pages(2,w2)

On échange le contenu de la page 2 et de la page dont le numéro est contenu dans w2.

b) swap-pages(w4,w5)

On échange les pages dont les numéros sont contenus dans les registres w4 et w5.

Important : l'instruction swap-pages n'est reconnue qu'au niveau fondamental.

5. Les instructions de contrôle

5.1 Conditions

Les conditions permettent de comparer des mots, des bytes et des structures considérées globalement; on peut également tester la valeur de n'importe quel bit d'un mot. Il existe enfin des conditions de fin de fichier (voir paragraphe 9) et d'interruption (voir paragraphe 8).

5.1.1 Comparaison d'objets

Forme : <source>,<connecteur>,<source>

Les 2 opérandes que l'on compare doivent être compatibles, par exemple 2 mots, 2 bytes, 2 structures-mot, etc.

Les connecteurs disponibles sont :

eq	=
ne	=
lt	<
le	⩽
gt	>
ge	⩾

Exemples :

a) w4,lt,5

On compare le contenu de w4 et 5

b) bl,ne,s3

On compare le contenu de bl et le dernier caractère placé sur la pile s3.

c) q3*,eq,'PERSONNE'

On compare le contenu de la queue q3 et la chaîne de texte 'PERSONNE'.

Inportant : lorsqu'on teste un élément ou tout le contenu d'une structure, celle-ci n'est évidemment pas réduite.

5.1.2 Test de bits

On peut tester si un bit d'un mot est à 0 ou à 1.

Forme : <bit>,<source>,<valeur>

Le bit est spécifié par bit0, bit1, ... bit15.

La valeur est 0 ou 1.

Exemples :

a) bit4,w3,1

On teste si le bit 4 du contenu de w3 est à 1.

b) bit0,s2,0

On teste si le bit 0 du dernier mot placé sur la pile s2 est à 0.

Notation : dans les paragraphes suivants, les conditions sont notées par C, les suites d'instructions par S.

5.1.3 Test de catégorie

Il est possible de tester la catégorie d'un caractère par l'un des 3 mots-clé suivants :

a) letter : désigne une des lettres 'A' à 'Z';
b) digit : désigne un des chiffres '0' à '9';
c) special : désigne tout autre caractère.

Un tel mot-clé peut se placer dans un champ d'une condition.

Exemples

a) b3,eq,digit

On teste si le contenu de b3 est un chiffre.

b) b4,gt,letter

On teste si le contenu de b4 est un caractère de code supérieur à ceux des lettres.

5.2 Instructions sélectives

5.2.1 Instruction conditionnelle

Deux formes existent :

i) if-begin(C) S if-end

Si la condition C est vérifiée, la suite d'instructions S est exécutée.

ii) if-begin(C) S_1 otherwise S_2 if-end

Si la condition C est vérifiée, la suite d'instructions S_1 est exécutée; si elle n'est pas vérifiée, c'est la suite S_2 qui est exécutée.

Exemples :

a) if-begin(w1,lt,0) negate(w1) if-end

Si le contenu de w1 est négatif, il est changé de signe.

b)
```
if-begin(w2,gt,10) decrement(w2) clear(w4)
        otherwise  increment(w2)  move(1,w4)
if-end
```

Si le contenu de w2 est supérieur à 10, il est diminué de 1 et le registre w4 est mis à 0; si le contenu de w2 n'est pas supérieur à 10, il est augmenté de 1 et le nombre 1 est placé dans w4.

c) imbrication :

```
if-begin(w1,lt,0)

      if-begin(w2,gt,3)   arithmetic(w2,*,w1,w3)

                          if-begin(w3,gt,-5)  clear(w3)

                          if-end
```

```
                otherwise   move(1,w2)
                            increment (w3)
                            if-begin(w3,gt,10)   clear(w2)
                                      otherwise   clear(w1)
                            if-end
        if-end
        move(w3,w2)
    otherwise   clear(w1)
if-end
```

5.2.2 Instruction de choix

Une sélection est effectuée en fonction de la valeur d'un mot ou d'un byte.

La forme de l'instruction est la suivante :

```
case-begin(<source>)
  of(<valeurs>)  S
  of(<valeurs>)  S
   .
   .
   .
  of(<valeurs>)  S
  other-case  S
case-end
```

Les valeurs doivent être des nombres entiers si la source est mot, des caractères si la source est byte. Les valeurs doivent être différentes. Si la source est égale à une valeur, la suite d'instructions correspondante est exécutée. Si la source n'est égale à aucune valeur, c'est la suite d'instructions précédée de other-case qui est exécutée. La clause other-case peut être omise. Si plusieurs valeurs sont placées dans un of, elles doivent être séparées par des virgules.

Exemples :

a)
```
case-begin(w1)
  of(1)  clear(w2)
  of(2,3)  increment(w2)
           clear(w3)
```

```
      of(4,6,10)  decrement(w2)
      other-case  clear(w2) clear(w3)
   case-end
b) case-begin(b3)
      of('A',ctrl 'C')  increment(w1)
      of('Z','$','W')  clear(w1)  arithmetic(w2,+,2,w2)
      of('B','C')
      other-case  clear(w2)
   case-end
```

On remarque que l'on n'exécute aucune instruction si b3 contient un 'B' ou un 'C'.

Une catégorie de caractères peut apparaître dans une clause of.

Exemple :

```
   case-begin(b1)
     of(letter,digit,'-')  increment(w1)
     other-case  increment(w2)
   case-end
```

5.3 Instructions répétitives

5.3.1 Instruction "while"

Forme : while-begin(C) S while-end

Tant que la condition C est vérifiée, la suite d'instructions S est exécutée.

Exemples :

a) factorielle de 5

```
   move(1,w1) move(1,w2)
   while-begin(w1,le,5)
        arithmetic(w2,*,w1,w2)
        increment(w1)
   while-end
```

Le résultat se trouvera dans le registre w2.

b) recherche d'un $ dans une queue :

```
while-begin(q1,ne,'$')  move(q1,b1)

while-end
```

L'instruction move est nécessaire ici pour enlever les caractères de la queue.

5.3.2 Instruction "repeat"

Forme : repeat-begin S repeat-end (C)

On répète la suite d'instructions S, jusqu'à ce que la condition C soit vérifiée. La suite d'instructions est toujours exécutée au moins une fois.

Exemples :

a) factorielle de 5 :

```
move(1,w1)  move(1,w2)

repeat-begin

     arithmetic(w2,*,w1,w2)

     increment(w1)

repeat-end(w1,gt,5)
```

Le résultat se trouve dans le registre w2.

b) transferts de tous les caractères d'une queue dans une pile jusqu'au premier espace compris :

```
repeat-begin  move(q1,S3)

repeat-end(q1,eq,space)
```

5.3.3 Instruction "loop"

Forme :

```
loop-begin(<indice>,<debut>,<fin>,<pas>,<sens>)

       S

loop-end
```

L'indice est nécessairement un registre-mot. Les limites de l'indice sont début et fin, 2 opérandes-mot. Le pas est aussi un opérande-mot. Ces 3 opérandes peuvent évidemment être variables. Le sens de la boucle est soit up (boucle ascendante), soit down (boucle descendante).

Exemples :

a) factorielle de 5 :

```
move(1,w2)
loop-begin(w1,1,5,1,up)
     arithmetic(w2,*,w1,w2)
loop-end
```

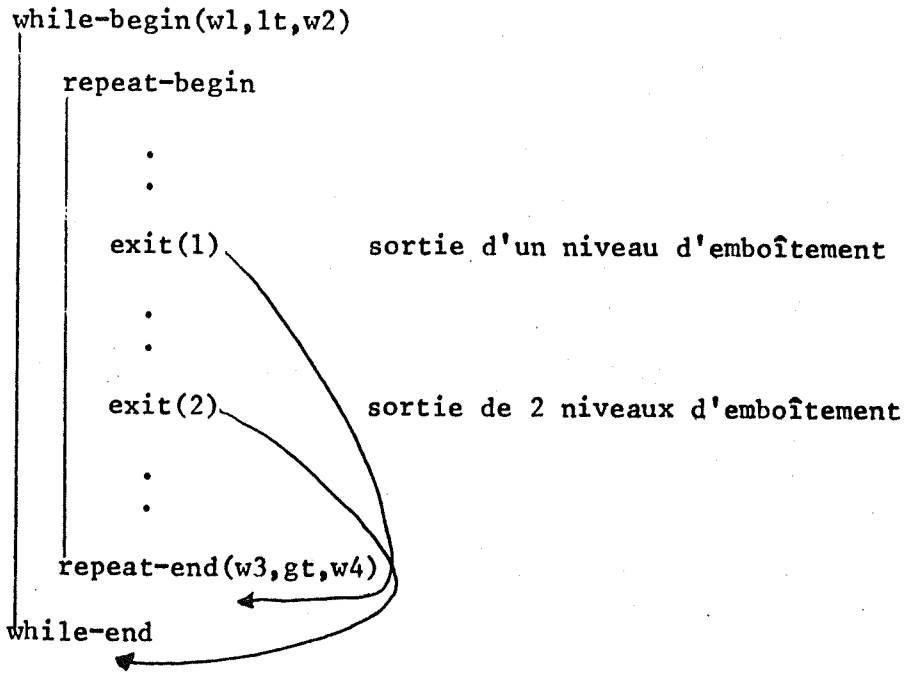
b) calcul de la somme des nombres pairs de 500 à 400 :

```
clear(w1)
loop-begin(w2,500,400,2,down)
     arithmetic(w1,+,w2,w1)
loop-end
```

5.3.4 Sorties de boucles

Il est possible de sortir de boucles emboîtées (while, repeat ou loop) par l'instruction exit. On précise comme champ le niveau de sortie des boucles.

Exemple :



De la même manière, on peut passer au pas suivant d'une boucle par l'instruction next. Le niveau doit aussi être indiqué.

Exemple :

```
loop-begin(w1,1,10,2,up)
        .
        .
        .
if-begin(w3,gt,0)  next(1)  if-end
        .
        .
        .
loop-end
```

6. Le traitement des structures

Les structures sont des collections d'éléments organisés, ces éléments peuvent être des mots ou des bytes; ce choix est fait par la pseudo-instruction declare (voir 12).

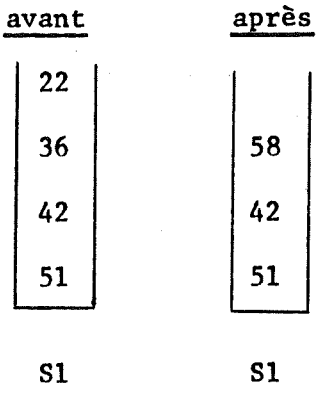
6.1 Piles et queues

Le traitement de ces deux structures s'effectue au moyen de toutes les instructions propres aux registres simples. les structures ne sont pas réduites dans les tests.

Exemples :

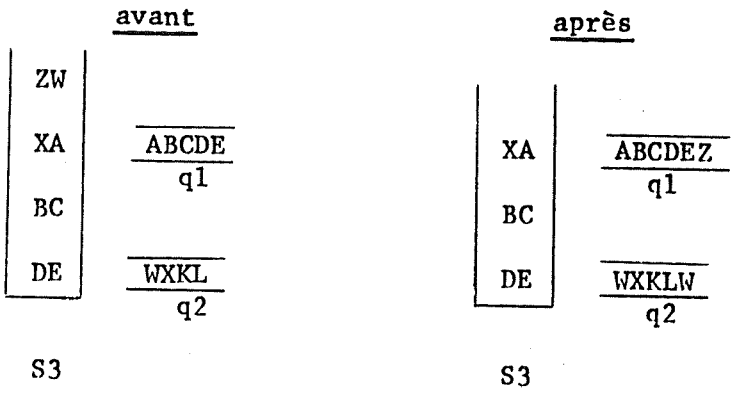
a) arithmetic(s1,+,s1,s1)

On additionne les 2 valeurs au sommet de la pile s1, le résultat est rangé à la place sur la pile.



b) decompose(s3,q1,q2)

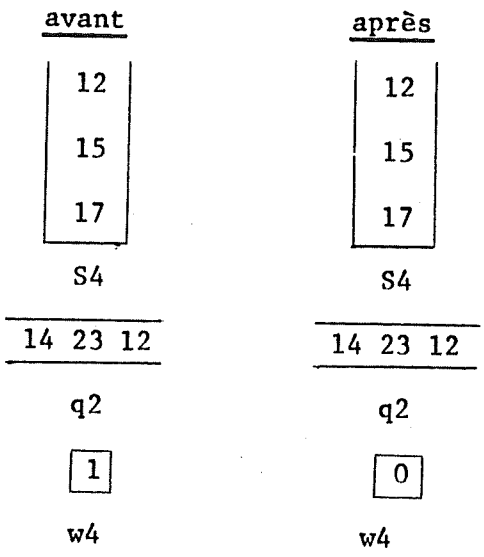
Le dernier mot sur la pile s3 est décomposé en 2 bytes qui sont placés dans les queues q1 et q2.



c) if-begin(s4,eq,q2) clear(w4)

if-end

Le contenu de w4 est effacé, si le dernier élément sur la pile s4 est égal au dernier placé dans la queue q2.



6.2 Tables

Le traitement de tables s'effectue exclusivement par l'instruction oper dont la forme est la suivante :

oper(<opération>,<table>,<indice>,<composante>)

Trois opérations sont disponibles :

out : on extrait une composante de la table.

into : on introduit une composante dans la table.

look : on cherche l'indice d'une composante dont on connaît le contenu; -1 est fourni si la recherche se révèle infructueuse.

L'indice est une opérande-source pour les opérations out et into, c'est un opérande-destination pour look.

La composante est une opérande-source pour les opérations into et look, c'est un opérande-destination pour out.

La composante doit être un mot, si la table est formée de mots, sinon elle doit être un byte. L'indice est obligatoirement un mot.

Exemples :

a) initialiser une table de 10 bytes avec des espaces :

```
loop-begin(w1,1,10,1,up)
     oper(into,t1,w1,space)
loop-end
```

b) classer une table de 100 éléments dans l'ordre croissant :

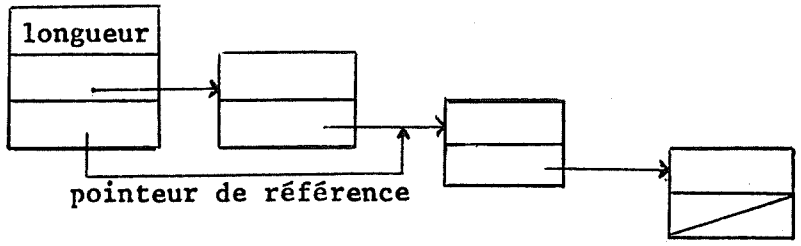
```
loop-begin(w1,1,99,1,up)
     arithmetic(w1,+,1,w3)
     loop-begin(w2,w3,100,1,up)
          oper(out,t1,w1,w4)   oper(out,t2,w2,w5)
          if-begin(w4,gt,w5)   oper(into,t1,w1,w5)
                               oper(into,t2,w2,w4)
          if-end
     loop-end
loop-end
```

c) chercher la valeur -10 dans la table t4, et placer son indice dans la pile s1 :

```
oper(look,t4,s1,-10)
```

6.3 Chaînes

Pour comprendre la manipulation des chaînes, il est nécessaire de connaître leur organisation.



Les éléments sont chaînés par un pointeur vers l'avant; un pointeur référence le prochain élément atteignable.

L'instruction de base de traitement de chaînes est l'instruction oper disponible avec 4 opérations différentes.

i) oper(extract,<element>,<chaîne>)

enlève l'élément référencé (tête) de la chaîne.

ii) oper(insert,<élément>,<chaîne>)

attache l'élément à la chaîne.

iii) oper(advance,<élément>,<chaîne>)

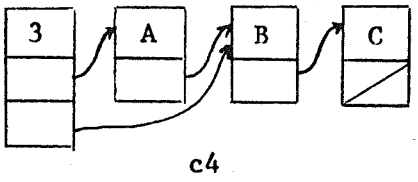
avance le pointeur de référence sur le prochain élément.

iv) oper(reset,<élément>,<chaîne>)

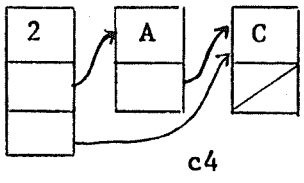
place le pointeur de référence en début de chaîne.

Exemples :

a) oper(extract,b3,c4)

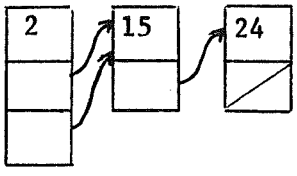


b3 avant

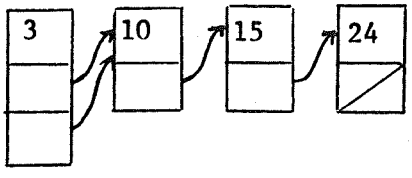


b3 après

b) oper(insert,w2,c1)

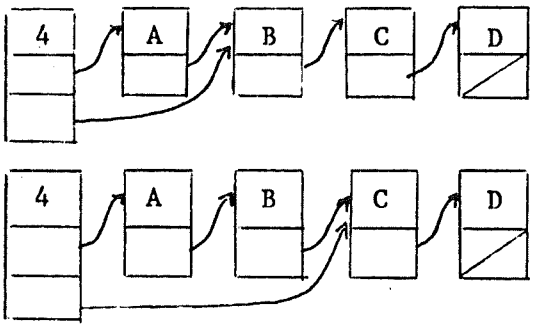


w2 avant

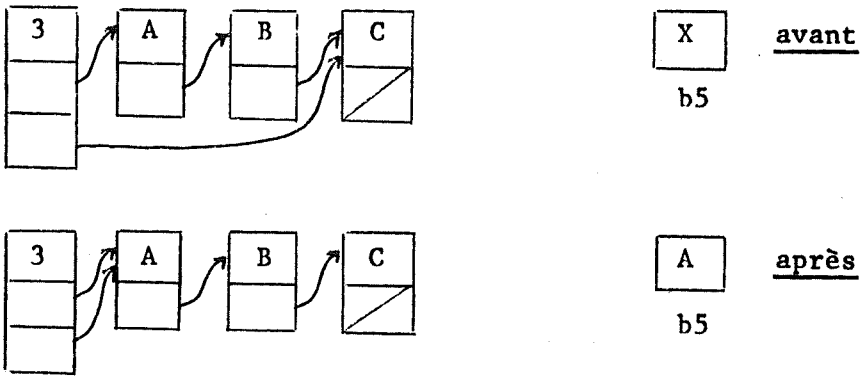


w2 après

c) oper(advance,b4,c2)



d) oper(reset,b5,c3)



Remarques :

I) Il est possible d'utiliser les chaînes dans des instructions simples.

Exemples :

a) move(c4,b2)

est synonyme de

oper(extract,b2,c4)

b) arithmetic(c1,+,2,c1)

est synonyme de

oper(extract,b$_i$,c1)

arithmetic(b$_i$,+,2,b)

oper(insert,b$_i$,c1)

II) Il n'est pas permis de comparer des éléments de chaînes.

if-begin(c1,lt,3) ...

doit être remplacé par

oper(extract,w1,c1)

if-begin(w1,lt,3) ...

6.4 Longueur des structures

La longueur courante d'une structure (nombre d'éléments) peut être connue grâce au préfixe length placé devant cette structure.

Exemples :

a) move(length q3,w2)

La longueur de la queue q3 est placée dans le registre w2.

b) clear(w1)

```
while-begin(length s1,gt,0)
     arithmetic(w1,+,s1,w1)
while-end
```

On calcule la somme des éléments de la pile s1.

Remarque : length ne peut pas être utilisé avec une table.

6.5 Libération des structures

L'instruction release permet de libérer une pile, une queue ou une chaîne.

Forme : release(<structure>)

L'instruction n'est pas valable pour une table.

L'effet de l'instruction est différent suivant la structure :

piles,queues :

la longueur de la structure est mise à 0. La place occupée par la structure n'est pas libérée (statique);

chaîne :

la place est dynamiquement libérée.

Exemple : release(c6)

7. Les entrées-sorties

7.1 Lecture et écriture

Le langage SPIP ne possède pas d'instructions spécifiques de lecture ou d'écriture, c'est l'instruction move qui effectue ces différentes actions.

Il est ainsi possible d'effectuer les transferts suivants :

registre ↔ périphérique

mémoire ↔ périphérique

périphérique ↔ périphérique

7.1.1 Transferts simples

Forme : move(<source>,<destination>)

Exemples :

a) move('A',output)
écriture du caractère 'A' sur terminal

b) move(b3,output)
écriture du caractère contenu dans le registre b3 sur le terminal

c) move(input,b2)
un caractère est lu sur le terminal et placé dans le registre b2.

Remarques :

1) L'opérande peut être un mot, dans ce cas le transfert est effectué successivement pour les 2 bytes.

On a donc les équivalences suivantes :

move(w_i,output)	decompose(w_i,b_j,b_k)
	move(b_j,output)
	move(b_k,output)
move(input,w_i)	move(input,b_j)
	move(input,b_k)
	compose(b_j,b_k,w_i)

Exemples :

a) move(41101B,output)

écriture de 'BA' sur le terminal

b) move(input,ind 3 w4)

lecture de 2 caractères et rangement à l'adresse obtenue en additionnant 3 au contenu de w4.

2) Lors d'une lecture sur le terminal, il est possible de demander l'écho en ajoutant un troisième champ echo dans l'instruction move.

p.e. move(input,b4,echo)

7.1.2 Transferts complexes

Il est possible d'effectuer des entrées-sorties avec conversion et liées directement aux structures.

Forme : move(<source>,<type>,<destination>,<type>)

Les types possibles sont : decimal, octal, binary et ascii.

Si un type est omis, il est supposé ascii; pour un périphérique, le type doit être ascii.

L'opérande source peut être une chaîne de caractères.

Exemples :

a) move(w4,decimal,output)

le contenu du registre w4 pris comme un nombre décimal est imprimé sur le terminal.

b) move(input,w6,octal)

un nombre octal est lu sur le terminal et transféré dans le registre w6.

c) move('BONJOUR',output)

écriture de la chaîne 'BONJOUR' sur le terminal.

Remarques :

1) Pour une lecture, un délimiteur (fin des caractères à lire) peut être précisé; par défaut, le délimiteur est l'espace (voir exemple b ci-dessus).

p.e. move(input,w5,binary,cr)

un nombre binaire terminé par un retour de chariot est lu sur le terminal et transféré dans le registre w5.

2) Il est possible de demander l'écho lors d'une lecture sur le terminal en spécifiant comme dernier champ le mot echo

p.e. move(input,w4,octal,cr,echo)

7.2 Recherche

Par l'instruction find on peut rechercher un byte donné, c'est-à-dire que le prochain caractère lu sur le périphérique suit le byte indiqué.

Forme : find(<périphérique>,<byte>)

L'instruction est équivalente à :

```
repeat-begin   move(<périphérique>,b↓)

repeat-end(b↓,eq,<byte>)
```

Exemple :

```
find(input,line-feed)
```

lecture de caractères sur le terminal jusqu'à la reconnaissance d'un "line-feed".

7.3 Test de fin de données

On peut tester la fin des données en provenance d'un périphérique par la condition :

eof,<périphérique>,on

"eof, périphérique ,off" étant vrai tant que cette fin n'est pas détectée.

Exemple :

```
while-begin(eof,input,off)

      move(input,s1)

while-end
```

On lit des caractères et on les empile tant qu'il y a des données.

Remarques : sur le terminal, c'est le caractère ctrl 'E' qui est reconnu comme fin de données.

7.4 Transferts disque-mémoire

Au niveau fondamental, il est possible d'effectuer les transferts suivants :

page mémoire ⟷ secteur disque

par les instructions :

fetch-sector(<secteur>,<page>) disque ⟶ mémoire

store-sector(<secteur>,<page>) mémoire ⟶ disque

Le secteur et la page sont décrits par des opérandes mot-source.

Exemples :

a) fetch-sector(w4,w5)

transfert du secteur dont le numéro est contenu dans w4 dans la page de numéro rangé dans w5.

b) store-sector(0,0)

stockage de la page 0 dans le secteur 0.

7.5 Entrées-sorties au niveau fondamental

Au niveau fondamental, les 2 instructions d'entrée-sortie principales sont i-o et disk-i-o.

Formes : i-o(<équipement>,<registre byte>)

disk-i-o(<opération>,<mot>,<mot>,<mot>)

i-o effectue un transfert entre le registre-byte et l'équipement.

disk-i-o effectue un transfert entre le disque et la mémoire. Le premier mot est l'adresse secteur, lesecond l'adresse mémoire et le troisième doit être un registre de status en cas d'erreur. (0 = pas d'erreur).

L'opération into correspond à une lecture du secteur, out à une écriture.

Les opérations i-o et disk-i-o n'assurent aucun contrôle, elles ne sont nécessaires que pour l'implantation de requêtes fondamentales d'entrée-sortie dans lesquelles les tests utiles sont effectués.

8 Le traitement des interruptions (niveau fondamental)

8.1 Définition d'un module de service

Lors d'une interruption, un branchement à ce module est effectué, on le définit par service-begin (d) ... service-end, où d est un périphérique.

p.e. service-begin(disk) ... service-end.

Il est possible de définir un seul module de service et de séparer les différents traitements spécifiques par des instructions de contrôle.

8.2 Le sauvetage et la restauration de l'état du processeur

Les instructions save-processor et restore-processor effectuent cette tâche. Ces instructions sont générées automatiquement à l'entrée et à la sortie d'un module de service.

8.3 L'enclenchement du système d'interruption

On enclenche le système d'interruption par interrupt(on), on le déclenche par interrupt(off). On peut également permettre ou interdire les interruptions pour un périphérique donné, par l'instruction select-interrupt, ce qui permet de donner des priorités.

p.e. select-interrupt(disk,on)

8.4 La reconnaissance des périphériques qui ont causé l'interruption

La condition interrupt,équipement,on teste si un équipement a interrompu.

Enfin, on peut sélectionner l'équipement qui a interrompu par l'instruction case-interrupt.

```
p.e. if-begin(interrupt,disk,on) ... if-end
     case-interrupt
     of(input) ...
     of(disc) ...
     other-case ...
     case-end
```

Restriction : la condition d'interruption ne peut être utilisée que dans une instruction if; de plus cette instruction if, comme l'instruction case-interrupt, ne peut être employée que dans un module général de service et comme seule instruction de ce module.

Exemple :

Programme qui sonne (ctrl 'G') et qui ne s'arrête que si l'on tape un caractère sur le clavier du terminal.

```
select-interrupt(input,on)
interrupt(on)
clear(w1)
repeat-begin  move(ctrl 'G',output)
repeat-end(w1,eq,1)
```

Le module de service peut être défini de 2 manières :

a) module spécifique :
```
service-begin(input)
  move(1,w1)
service-end
```

b) module général :
```
service-begin
  case-interrupt
    of(input)  move(1,w1)
      .
      .
      .
  case-end

service-end
```

9. Le traitement des fichiers

9.1. Instructions standards

Les instructions disponibles pour les périphériques le sont aussi pour les fichiers-disque, par l'intermédiaire des unités logiques f1 à f10.

Exemples :

a) move(f1,b1)

lecture d'un caractère à partir du fichier f1.

b) move(f2,f3)

copie d'un caractère d'un fichier sur un autre.

c) while-begin(eof,f4,off) S while-end

tant que la fin du fichier f4 n'est pas détectée, la suite d'instructions S est exécutée.

d) find(f2,'$')

recherche du caractère '$' dans le fichier f2.

9.2 Instructions spécifiques

9.2.1 Ouverture et fermeture des fichiers

Un fichier doit être ouvert avant sa première utilisation et fermé après sa dernière utilisation.

Ouverture

L'opération d'ouverture consiste à attacher à une unité logique un fichier donné. Il est possible de tester si le fichier existe et d'obtenir certaines caractéristiques. L'instruction d'ouverture de fichier est l'instruction attach.

Forme : attach(<unité logique>,<nom du fichier>,<type>,
<propriétés>,<status>)

Le nom du fichier est soit une chaîne de texte, soit une queue.

Le type est un registre-byte : il contient, si le fichier est trouvé, une lettre caractéristique.

Les propriétés (protection) sont fournies dans un registre-mot.

Enfin, le status est un registre-mot; 0 au retour signifie que le fichier a été trouvé dans le catalogue.

Exemples :

a) attach(f4,'DATA',b1,w1,w2)

On attache à l'unité f4 le fichier nommé 'DATA'; le type du fichier est retourné dans b1, ses propriétés dans w1 et le status dans w2.

b) attach(f4,q2,b1,w1,w2)

L'ouverture est semblable, mais le nom de fichier est contenu dans la queue q2.

Remarque : l'instruction attach est liée au système d'exploitation du mini-ordinateur. Il est donc fortement souhaitable lorsqu'on développe un nouveau système d'exploitation à l'aide de SPIP, de conserver la structure de cette instruction.

Fermeture

On ferme un fichier par l'instruction close-file.

Exemple : close-file(f2)

9.2.2 Fin de fichier et repositionnement

On peut repositionner un fichier par l'instruction rewind et mettre une marque de fin de fichier par l'instruction end-of-file.

Formes : rewind(<unité logique>)

end-of-file(<unité logique>)

Exemples : rewind(f1) end-of-file(f2)

Remarque : en écriture, un "end-of-file" est écrit automatiquement lors d'un "rewind" ou d'un "close-file".

10. Les modules

10.1 Définition des modules

50 modules peuvent être définis dans un programme par l'instruction "define". Ces modules peuvent être récursifs.

Forme : define-begin(<module>,recursive
<liste de paramètres formels>)
<declarations>
<corps du module>

define-end

Le mot recursive doit être omis dans le cas de modules non récursifs.

10.1.1 Liste des paramètres formels

Forme : <vide>|,<paramètre>|<liste des paramètres><paramètre>

Restriction actuelle d'implantation :

Les paramètres formels ne peuvent être que des registres-mot ou des registres-bytes.

Un paramètre suivi du signe + est non modifiable

Exemples : w3, b4, w2+, b5+

10.1.2 Déclarations

On peut déclarer des registres-mot et des registres-bytes locaux à un module, par la pseudo-instruction loc.

Forme : loc(<registre>,<registre>, ...<registre>)

10.1.3 Corps d'un module

C'est une suite d'instructions pouvant référencer les paramètres, les registres locaux et les objets globaux.

10.2 Utilisation des modules

Un module s'utilise au moyen de l'instruction need.

Forme : need(<module> <liste de paramètres actuels>)

La liste des paramètres actuels est formée de paramètres séparés par des virgules. Ces paramètres doivent correspondre en genre et en nombre aux paramètres formels de la définition du module.

Exemples :

```
define-begin(m1,w2,b4+) ... define-end

need(m1,w3,'A')   }
                  } corrects
need(m1,w3,b2)    }

need(m1,b2,w3)    }
                  }
need(m1,b2)       } incorrects
                  }
need(m1,5,'A')    }
```

10.3 Prédéfinition de modules

Un module ne peut être invoqué qu'après sa définition; si 2 modules récursifs doivent s'appeler mutuellement, il faut prédéfinir l'un des 2 par l'instruction pre-define. Les paramètres doivent apparaître dans la prédéfinition et non dans la définition.

Exemple :

```
pre-define(m1,recursive,w1)

define-begin(m2,b2+)

        .
        .
        .

   need(m1,w2)

        .
        .
        .

define-end

define-begin(m1)

        .
        .
        .

   need(m2,'A')

        .
        .
        .

define-end
```

10.4 Exemples de modules

10.4.1 Module non récursif

Module calculant le nombre de fois qu'un caractère se trouve dans un texte (unité f1).

```
define-begin(m1,b1+,w1)
loc(b2)
rewind(f1)   clear(w1)
while-begin(eof,f1,off)   move(f1,b2)
   if-begin(b1,eq,b2)  increment(w1)
   if-end
while-end
define-end
```

p.e. pour calculer le nombre de 'A'

```
need(m1,'A',w2)
```

Le résultat est donné dans le registre w2.

10.4.2 Module récursif

pgcd de 2 nombres

```
define-begin(m4,recursive,w1+,w2+,w3) « w3=pgcd(w1,w2) »
  if-begin(w2,eq,0)  move(w1,w3)
      otherwise   arithmetic(w1,modulo,w2,w4)
                  need(m4,w2,w4,w3)
  if-end
define-end
```

On calculera le pgcd de 36 et 45 par :

```
need(m4,36,45,w1)
```

10.5 Branchement conditionnel

L'instruction condition permet un branchement conditionnel à des modules.

Forme : condition(<relation>,<module>,<module>)

Si la condition est vérifiée, un branchement au premier module s'opère, sinon il se fera au second.

Les modules ne peuvent pas avoir de paramètres.

condition(w1,lt,w2,m1,m2) est équivalent à :

```
if-begin(w1,lt,w2) need(m1)
        otherwise need(m2)
if-end
```

Exemple :

```
define-begin(m1)
  move('ERREUR W1 TROP GRAND',output)
  clear(w1)
define-end

define-begin(m2)
  increment(w1)  arithmetic(w1,+,6,w1)
define-end
    .
    .
    .
condition(w1,gt,100,m1,m2)
```

10.6 Modules standards

3 modules standards peuvent être utilisés dans les instructions need et condition.

continue : ignore l'appel

halt : arrête le processeur

system : donne le contrôle au moniteur du système

Exemples :

a) need(halt) demande l'arrêt du processeur

b) condition(w1,gt,0,continue,system)

Si w1 est positif, on continue en séquence, sinon on redonne le contrôle au moniteur.

10.7 Les instructions stop, return-monitor et abort

L'instruction stop arrête le processeur, elle est équivalente à need(halt); l'instruction return-monitor est synonyme de need(system).

Enfin, l'instruction abort(<chaîne de texte>) est équivalente à la séquence :

```
move(<chaîne de texte>,output)

return-monitor
```

Exemple : abort('ERREUR FATALE')

Le message 'ERREUR FATALE' est imprimé sur le terminal et le contrôle est rendu au moniteur.

11. Les requêtes

Le principal but du langage SPIP est l'écriture de systèmes d'exploitation. Un moniteur, développé en SPIP, est dans l'implantation actuelle, nécessairement résident. Il se présente comme un ensemble de requêtes.

11.1 Restrictions et extensions au niveau d'un moniteur

Le moniteur est une unité de programmation privilégiée; il se trouve donc au niveau fondamental, ce qui implique les règles suivantes :

i) les adressages indirect, indexé et par page sont autorisés.

ii) on peut autoriser et traiter les interruptions en provenance des périphériques.

iii) les instructions d'accès au disque par secteur sont permises.

iv) l'équipement disk est reconnu.

v) les modules ne sont pas autorisés, à l'exception des modules de traitement d'interruption.

vi) l'utilisation de structures est interdite.

vii) les unités logiques ne peuvent figurer que comme paramètres de requêtes.

viii) une instruction error-return(<numéro>) peut être utilisée dans une requête, elle imprime un message et redonne le contrôle à la partie du moniteur traitant les commandes.

ix) chaque requête a ses propres registres. La transmission d'informations se fait par paramètres.

11.2 Définition d'une requête

Une requête est définie de la manière suivante :

```
request-begin(<chaîne><liste des paramètres formels>)
    <corps de la requête>
request-end
```

Le nom de la requête est une chaîne délimitée par des apostrophes et n'ayant pas plus de 10 caractères.

La liste des paramètres peut être vide, sinon les paramètres sont séparés par des virgules, une virgule sépare la chaîne du premier paramètre.

Un appel à une requête peut figurer dans le corps d'une autre requête. Toutefois, toute récursivité est interdite.

11.2.1 Paramètres

Les paramètres peuvent être :

a) des mots

b) des bytes

c) des unités logiques.

11.2.2 Paramètres, mots et bytes

3 types de passages de paramètres existent :

1. paramètre variable : p.e. w3, b4
2. paramètre non modifiable : p.e. w3+, b4+
3. paramètre par valeur : p.e. w3/, b4/

Dans le cas de paramètres par valeur, c'est réellement la valeur qui est passée comme paramètre, tandis que dans les premiers cas c'est l'adresse qui est transmise.

Exemple :

requête permettant de ranger une valeur dans une table ou d'en extraire une valeur.

```
request-begin('TABLE-INOU',w1/,w2/,w3+,w4)
if-begin(w3,lt,0)  error-return(12)
if-end
if-begin(w3,gt,ind 1 w2)  error-return(12)
if-end
```

```
arithmetic(w3,+,w2,w5)
if-begin(w1,eq,0)  move(w4,ind 2 w5)
     otherwise  move(ind 2 w5,w4)
if-end
request-end
```

w1 est un paramètre dont on transmet la valeur :

0 (ranger) ou 1 (extraire)

w2 est la valeur de l'adresse de la table

w3 est l'indice dans la table, il ne doit pas être modifié

w4 est la composante concernée, elle doit être déclarée variable.

11.2.3 Unités logiques passées comme paramètres

Il est possible de passer des unités logiques comme paramètres, mais leur usage dans le corps de la requête est limité aux 2 possibilités suivantes :

a) l'unité est transmise à nouveau dans un appel à une nouvelle requête;

b) l'adresse de l'unité est traitée par address<unité>

Exemple :

requête retournant dans w1, 1 si la fin du fichier f1 est détectée sinon 0.

```
request-begin('GIVE-EOF',f1,w1)
   move(address f1,w2)  «adresse de l'unité logique»
   move(ind 0 w2,w2)    «adresse de la zone de contrôle
                                              du fichier»
   if-begin(ind 1 w2,lt,0)  move(1,w1)
        otherwise clear(w1)
   if-end
request-end
```

On suppose ici que le mot d'adresse "ind 1 w2" est mis à une valeur négative à la fin du fichier par une requête de lecture.

11.3 Appel à une requête

L'appel à une requête s'effectue par l'instruction request.

Forme : request(<chaîne><liste de paramètres actuels>)

La chaîne constitue le nom de la requête qui doit avoir été définie.

Les paramètres actuels doivent correspondre en genre et en nombre avec les paramètres formels. Les règles suivantes doivent être respectées :

i) pour les paramètres variables mots ou bytes, seuls des registres sont autorisés.

ii) pour les paramètres mots ou bytes non modifiables, des registres, des nombres et des caractères sont autorisés.

iii) pour les paramètres par valeur, seuls des nombres ou des caractères sont acceptés. Néanmoins, il est possible de passer le contenu d'un registre, inconnu à la compilation, lors d'un appel à une requête dans une requête.

Exemples :

a) request('TABLE-INOU',1,100,10,w2)

b) request('GIVE-EOF',f4,w2)

11.4 Requêtes permanentes

Un certain nombre de requêtes sont appelées par le code produit par le compilateur SPIP. Elles doivent donc figurer dans le moniteur de tout système d'exploitation. Le corps de ces requêtes peut être modifié, mais leur séquence doit rester invariante. Pour plus de détails, nous renvoyons l'utilisateur à la référence bibliographique (1).

12. Les pseudo-instructions

Ce sont des instructions qui contrôlent la génération de code-objet, mais qui n'en produisent pas.

12.1 Déclarations de structures

12.1.1 Type des éléments

La pseudo-instruction declare permet de déclarer si une structure est formée de mots ou de bytes. Par défaut, c'est le mode byte qui est choisi.

Forme : declare(<structure>,<type>)

Le type est l'un des mots-clé bytes ou words.

Exemples : declare(t3,words) declare(q2,bytes)

12.1.2 Réservation

La pseudo-instruction dimension permet de réserver la place nécessaire à une structure statique.

Forme : dimension(<structure>,<nombre maximum d'éléments>)

Le nombre d'éléments doit être un entier positif.

Exemples : dimension(t4,122) dimension(q2,20)

Remarques :

I) Il est possible d'omettre la pseudo-instruction dimension; dans ce cas une réservation par défaut de 10 éléments est faite.

II) Il est permis de déclarer une table en accès pile ou queue, en introduisant un troisième champ stack-acces ou queue-acces dans la pseudo-instruction dimension.

Exemple :

```
dimension(t1,20,queue-acces)

dimension(q2,20)

      .
      .
      .

move(q2*,t1*)

oper(out,t1,3,w4)
```

12.2 Registres rapides

Le nombre de registres de la machine est nettement supérieur au nombre de registres d'une machine réelle. Il est donc nécessaire de simuler des registres à l'aide de la mémoire. Lors de l'écriture de requêtes ou de programmes en temps réel, il peut être avantageux de forcer l'utilisation de registres réels pour éviter de trop longs temps d'accès à la mémoire. On appellera registre rapide un registre de la machine λ réalisé par un registre de la machine réelle.

Pour déclarer un registre rapide, on utilise la pseudo-instruction fast-register.

Forme : fast-register(<registre>)

Le registre est un registre mot ou byte.

Si le registre doit être utilisé comme registre d'index, la forme sera :

fast-register(<registre>, indexed)

Lorsqu'un registre n'a plus besoin d'être rapide, on peut utiliser la pseudo-instruction fast-end.

Forme : fast-end(<registre>)

Exemples :

a) fast-register(w4)

```
move(w1,w4)
arithmetic(w4,*,w5,w4)
increment(w4)
move(w4,decimal,f1)
fast-end(w4)
```

b) fast-register(w6,indexed)

```
move(ind 5 w6,w7)
clear(ind 7 w6)
increment(ind 9 w6)
fast-end(w6)
```

12.3 Registres volatils

Un registre de la machine λ peut être déclaré volatil. Lorsqu'un tel registre est chargé, son contenu n'est assuré que jusqu'à la prochaine instruction SPIP.

Exemple :

```
volatil(w5)
arithmetic(w2,+,w4,w5)
move-byte(left,w5,b4)  ← contenu assuré
move(w5,w7)            ← contenu non assuré !!
move(w6,w5)
move(w5,w7)            ← contenu assuré (rechargé)
```

12.4 Objets locaux

Il peut être utile de déclarer des objets locaux à une partie de programme. Il ne s'agit pas des registres locaux à un module, mais d'utiliser des objets définis par SPIP.

Cette facilité est particulièrement intéressante pour les macros (voir chapitre 13).

Exemple :

```
macro(clear-tab)
  local(w1,on)
  loop-begin(w1,1,↑1,1,up)
      oper(into,↑2,↑1,0)
  loop-end
  local(w1,off)
$
```

Au lieu d'utiliser le registre w1, SPIP attribue un registre hors des limites de l'utilisateur.

Forme :

```
local(<objet>,on)
        .
        .
        .
local(<objet>,off)
```

C'est l'utilisateur qui fixe la limite de ses objets par la pseudo-instruction :

limits(w_i,b_j,s_k,q_ℓ,c_m,t_n,m_p,f_q)

par défaut, les limites correspondent à :

limits(w15,b5,s5,q5,c5,t15,m25,f5)

L'utilisateur peut donc élever ses limites mais il doit prendre garde que SPIP utilise parfois quelques registres.

12.5 Priorités d'accumulateurs

Voir référence (1).

12.6 Options

La pseudo-instruction option permet d'enclencher ou de déclencher une option du processeur SPIP. Les options sont initialisées par la commande processeur; on trouvera une description de cette commande et des options dans l'annexe A.

Forme : option(<lettre>,<operation>)

La lettre correspondant à l'option choisie doit être encadrée d'apostrophes.

L'opération on enclenche l'option, tandis que off la déclenche.

Exemple :

```
increment(w2)
                        } le code-source est listé
option('S',off)
move(3,w2)
     .
     .                  } le code source n'est pas listé
     .
clear(w4)
option('S',on)
move(1,w2)              } le code source est listé
```

Il va sans dire qu'il est dénué de sens d'enclencher ou de déclencher en cours de programme des options telles que 'A', 'R' ou 'V'.

12.7 Début et fin d'un programme

Un programme doit commencer nécessairement par la pseudo-instruction title et se terminer par la pseudo-instruction end. Un point doit suivre end.

Formes : title(<nom>)

end.

Le nom est un identificateur de 10 caractères au maximum. Dans le cas du moniteur, ce nom doit être "MONITOR".

Exemple :

```
title(test)

move(input,w1,decimal,echo)

arithmetic(w1,*,w1,w1)

move(' LE CARRE VAUT : ',output)

move(w1,decimal,output)

return-monitor

end.
```

12.8 Adresse de départ

L'exécution d'un programme démarre à la première instruction exécutable. Pourtant, il peut être nécessaire au niveau fondamental de commander l'exécution de code à partir d'une adresse donnée; c'est le rôle de la pseudo-instruction start.

Forme ; start(<mot>)

Le mot doit contenir l'adresse de départ.

Exemple : start(w1) départ à l'adresse contenue dans w1.

13. Les macros et les synonymes

Il est possible de définir et d'utiliser des macro-instructions à l'intérieur d'un programme. Ces macro-instructions (ou macros) peuvent avoir des paramètres. Elles peuvent être redéfinies.

13.1 Macros sans paramètre

13.1.1 Définition d'une macro

Forme : macro(<nom de la macro>)

```
        <texte de la macro>
        $
```

Le nom de la macro est un identificateur. Le caractère $ termine la définition, il ne peut donc pas figurer dans le texte de la macro.

Exemple :

```
macro(entete)
move(new-page,output)
move('RAPPORT CONFIDENTIEL 417',output)
$
```

13.1.2 Utilisation d'une macro

Une macro s'utilise par la simple mention de son nom. Le texte de la macro sera inséré à la place de ce nom.

exemple : entete

sera remplacé par :

```
move(new-page,output)
move('RAPPORT CONFIDENTIEL 417',output)
```

13.2 Macros avec paramètres

13.2.1 Définition d'une macro

Forme : macro(<nom de la macro>)

<texte de la macro>

$

On remarque que la définition d'une macro avec paramètres est semblable à une définition sans paramètre. Aucune liste de paramètres formels ne doit figurer. Les paramètres sont référencés dans le texte de la macro par ↑n, où n est l'ordre du paramètre.

Exemples :

a) macro(skip-line)

```
loop-begin(w1,1,↑1,1,up)
    move(new-line,output)
loop-end
$
```

Cette macro saute un certain nombre de lignes. Ce nombre est défini par le paramètre ↑1.

b) macro(condition)

```
if-begin(↑1,↑2,↑3)  need(↑4)
    otherwise need(↑5)
$
```

Cette macro effectue un branchement au module ↑4 si la condition ↑1,↑2,↑3 est vérifiée, sinon au module ↑5.

13.2.2 Utilisation d'une macro

Forme : nom de la macro(<liste des paramètres actuels>)

Les paramètres actuels doivent être séparés par des virgules; ils doivent être en nombre égal au nombre de paramètres formels.

Un paramètre actuel peut être toute suite de caractères ne comportant pas de virgule.

Exemples :

a) skip-line(5) saut de 5 lignes

L'expansion de la macro est :

```
loop-begin(w1,1,5,1,up)
        move(new-line,output)
loop-end
```

b) condition(w1,lt,0,m1,m2)

L'expansion de la macro est :

```
if-begin(w1,lt,0)  need(m1)
        otherwise need(m2)
if-end
```

Remarque importante : il est possible de définir une macro ayant le nom d'une instruction SPIP; à partir de cette définition, c'est toujours la macro qui est prise en considération.

Exemple :

```
title(exemple)
  compose('A','B',w1)   <— instruction compose
  macro(compose)
    arithmetic(↑1,or,177B,↑2)
  $
  compose(14372,w2)     <— macro compose
end.
```

13.3 Synonymes

13.3.1 Pseudo-instruction name

Pour améliorer la lisibilité des programmes, la pseudo-instruction name permet de donner un nom à un objet (registre, nombre, caractère, etc.). On définit de cette manière des synonymes.

Forme : name(<synonyme>,<objet>)

Exemples :

```
name(trois,3)

name(somme,w1)

name(scanner,m3)

name(dollar,'$')
```

L'utilisation d'un synonyme est possible partout où celle de l'objet correspondant est autorisée.

Exemple : module calculant la somme des carrés de 1 à N.

```
name(somme-carre,m3)

name(n,w4)  name(resultat,w5)

name(carre,w2)  name(entier,w1)

define-begin(somme-carre,n,resultat)

loc(entier,carre)

clear(resultat)

loop-begin(entier,1,n,1,up)

   arithmetic(entier,*,entier,carre)

   arithmetic(resultat,+,carre,resultat)

loop-end

define-end
```

La définition du module est absolument équivalente à :

```
define-begin(m3,w4,w5)

loc(w1,w2)

  clear(w5)

  loop-begin(w1,1,w4,1,up)

     arithmetic(w1,*,w1,w2)

     arithmetic(w5,+,w2,w5)

  loop-end

define-end
```

Remarques :

a) Il est possible de donner plusieurs noms au même objet

p.e. name(texte,q1) name(caractères,q1)

b) Il est possible de redéfinir un symbole du langage par la pseudo-instruction name

p.e. name(output,f3) permet d'écrire le texte destiné au terminal sur l'unité logique f3.

13.3.2 Pseudo-instruction sequence

Par cette pseudo-instruction, il est possible de donner un nom à une suite de nombres entiers commençant par 0.

Forme : sequence(<nom>,<nom>,...,<nom>)

Formellement, sequence(n_0,n_1,n_2,n_3,...,n_K) est équivalent à :

name(n_0,0) name(n_1,1) name(n_2,2) ... name(n_K,K)

Exemples :

a) sequence(rouge,jaune,vert,bleu)

rouge correspond à 0, jaune à 1, vert à 2 et bleu à 3.

b) sequence(lda,sta,add,sub,mul,div)

definit un jeu de 6 instructions dont les codes vont de 0 à 5.

Bibliographie

(1) Thalmann, D.
Ecriture de systèmes d'exploitation portables pour mini-ordinateurs.
Genève, faculté des Sciences, 1977, Thèse No. 1807.

(2) Thalmann, D., Levrat, B.
SPIP : A way of writing portable operating systems.
In : International Computing Symposium 1977. Proceedings of the ACM. Amsterdam, North Holland, 1977, pp. 451-459.

Annexe A : SPIP et O.S. 1100

SPIP fonctionne sur Univac 1108 sous le système O.S.1100.

La commande processeur est la suivante :

```
@P*SPIP.X,options   nom
```

Options disponibles

A : le code produit est absolu (option par défaut).

C : la production de code relogeable est poursuivie même en cas d'erreur interne.

E : l'impression des diagnostics d'erreurs à la compilation est effectuée.

G : le système SPIP est placé en mode privilégié. Les instructions du niveau fondamental sont autorisées.

I : un nouvel élément-source est créé à partir du flot d'entrée.

J : des statistiques sur le code-objet sont imprimées.

K : la liste des requêtes et des overlays du système est fournie à l'utilisateur.

L : le code produit est listé en langage d'assemblage.

M : la table de chargement est imprimée.

N : le code est produit pour un mini-ordinateur Nova.

O : le code produit est listé en octal.

P : le code est produit pour un mini-ordinateur PDP-11.

Q : l'expansion des macros est imprimée.

R : le code produit est relogeable (pas d'édition de liens).

S : le listage du code source est effectué.

V : seule l'analyse du code-source est opérée.

X : l'exécution du code-objet est simulée sur Univac 1108.

Z : le code-objet est envoyé automatiquement de l'Univac à la Nova via le réseau SORGUE.

Le nom spécifié dans la commande processeur constitue un élément du fichier OS-1100 P*SPIP-O-S.

Si l'option I est utilisée, aucun nom ne doit être spécifié, le nom de l'élément créé sera celui figurant dans la pseudo-instruction title.

Si le processeur est appelé sans nom et sans l'option I, le programme est traité, mais la version-source n'est pas conservée.

Annexe B : erreurs

B.1 Erreurs détectées par le compilateur SPIP

(lexicales, syntaxiques et sémantiques)

1. caractère illégal
2. chaîne de texte trop longue
3. nombre trop grand
4. un identificateur était attendu
5. une parenthèse gauche était attendue
6. champ d'instruction illégal
7. une virgule était attendue
8. une parenthèse droite était attendue
9. instruction inconnue
10. champ d'instruction inconnu
11. length ne doit être utilisé qu'avec des structures
12. shift ou ctrl ne sont autorisés que devant des caractères
13. niveau illégal dans exit ou next
14. title était attendu
15. le nom dans title est illégal
16. "local off" sans "local on"
17. incompatibilité dans un case-begin ... case-end
18. le niveau dans exit ou next est trop grand
19. instruction illégale
20. trop de boucles imbriquées
21. trop de repeat-begin ... repeat-end imbriqués
22. trop de while-begin ... while-end imbriqués
23. les modules imbriqués ne sont pas autorisés
24. trop de loop-begin ... loop-end imbriqués
25. trop de if-begin ... if-end imbriqués
26. trop de case-begin ... case-end imbriqués
27. l'indice d'un loop-begin ... loop-end est déjà l'indice d'une boucle imbriquante
28. un déplacement négatif dans un adressage indexé n'est pas autorisé.
29. trop de of dans un case-begin .... case-end
30. module non défini
31. nombre de champs incorrect dans cette instruction
32. après case-begin ou case-interrupt of est nécessaire
33. le déplacement dans un adressage indexé doit être numérique
34. le déplacement dans un adressage indexé est trop grand
35. incompatibilité entre 2 champs
36. digit octal incorrect
37. ceci n'est pas un code de caractère
38. ce caractère en shift n'existe pas
39. ce caractère de contrôle n'existe pas
40. l'usage de 2 caractères de contrôle dans une même instruction est interdit
42. trop de macros
43. paramètre illégal dans une définition de macro
44. paramètre de macro trop long

45. trop de paramètres dans cette macro
46. après page 2 opérandes-mots sont attendus
47. un nombre était attendu
48. adresse illégale
49. un troisième champ n'est autorisé qu'avec des tables
50. dimension illégale
51. requête inconnue
52. fast-end sans fast-register
53. trop de requêtes
54. trop d'overlays
55. paramètre formel illégal
56. paramètre par valeur illégal
57. nombre de paramètres actuels ne correspondant pas au nombre de paramètres formels.
59. objet hors des limites
60. seuls 2 registres volatils sont autorisés
61. les requêtes ne peuvent être imbriquées
62. trop de champs dans cette instruction
63. la définition de module dans une requête n'est pas autorisée
64. trop de locaux dans ce module
65. trop de modules prédéfinis
66. trop d'instructions de contrôle imbriquées

B.2 Erreurs détectées par les procédures de génération de code

(erreurs internes)

Si une telle erreur se produit, une description des registres de la machine réelle est donnée. La ligne du code-source où s'est produite l'erreur est signalée ainsi que le code correspondant à la dernière instruction SPIP reconnue. L'erreur est normalement fatale et provoque une division par 0, permettant ainsi l'indication des appels de procédures de génération impliquées. L'option C du processeur permet de continuer la génération.

Codes d'erreurs

1. page 0 pleine
4. trop de code produit sous ce compteur
5. trop de relais
7. trop de références à des locaux dans cette requête
8. trop de locaux dans cette requête
9. requête trop longue
63. plus d'objets locaux (voir 12.4)
79. trop de constantes.

Pour de plus amples informations voir (1).

B.3 Erreurs détectées par l'éditeur de liens

Ces erreurs sont toujours fatales; elles se présentent sous la forme :

```
$$$ SPIP$LINK: FATAL ERROR

        diagnostic
```

7 diagnostics sont possibles :

1. trop d'instructions
2. trop de chaînes de caractères
3. erreur interne
4. erreur interne
5. moniteur trop long
6. non implanté pour ce mini-ordinateur
7. trop d'adresses à calculer

Les erreurs 3 et 4 sont normalement dues à des erreurs dans la génération de code relogeable avec option C.

B.4 Erreurs à l'exécution

Ces erreurs peuvent se produire à l'exécution sur le mini-ordinateur réel, un même diagnostic peut avoir différents numéros de code suivant la source d'erreurs (requête).

Code	Diagnostic
1,3	pile pleine
2,4	pile vide
5,6	queue pleine
7,8	queue vide
12,13	indice de table hors des limites
20	digit plus grand que la base
21	base numérique non prévue
100,101,110,111	erreur dans un transfert disque
700	fin de chaîne détectée

Annexe C : limitations du compilateur SPIP

Aucune des limites suivantes ne doit être dépassée :

longueur d'une chaîne de texte : 144 caractères

nombre de boucles imbriquées : 40

nombre de repeat-begin ... repeat-end imbriqués : 20

nombre de while-begin ... while-end imbriqués : 20

nombre de loop-begin ... loop-end imbriqués : 20

nombre de case-begin ... case-end imbriqués : 20

nombre de if-begin ... if-end imbriqués : 20

nombre d'instructions de contrôle imbriquées : 40

nombre de requêtes : 70

nombre entier : 32767

nombre de of dans un case-begin ... case-end : 60

déplacement dans un adressage indexé : 127

nombre de champs dans une instruction : 15

nombre de macros : 50

nombre de paramètres dans une macro : 8

nombre de caractères dans un paramètre de macro : 36

Annexe D : objets prédéfinis

new-page	15414b	(2 caractères pour effacer l'écran)
spaces	20040b	(2 espaces)
new-line	6412b	("carriage return + line-feed")
carriage-return	15b	
cr	15b	
line-feed	12b	
space	40b	
quote	47b	
zero-byte	"0	
page-length	256	
max-address	16383	