université de nancy I

u.e.r. de mathématiques

le langage snobol 4 ses applications, son implémentation

THESE

présentée pour l'obtention du doctorat de spécialité de mathématiques appliquées

par

jacques jaray

soutenue le 7 juin 1975

jury : président c. pair

examinateurs j.c. derniame

j.p. haton

a. quéré

université de nancy I
u.e.r. de mathématiques

le langage snobol 4
ses applications,
son implémentation

THESE

présentée pour l'obtention du
doctorat de spécialité
de mathématiques appliquées

par

jacques jaray

soutenue le 7 juin 1975

jury : président c. pair

examinateurs j.c. derniame
j.p. haton
a. quéré

Je remercie R.E. GRISWOLD qui a toujours bien voulu répondre avec beaucoup de rapidité et de précision à mes questions.

Monsieur PAIR a dirigé cette étude ; je le remercie bien sincèrement pour l'expérience qu'il m'a fait acquérir.

Je remercie Jean-Claude DERNIAME et Jean-Paul HATON pour leur aimable participation au Jury.

Alain QUERE m'a aidé à formaliser certaines parties de ce travail ; je le remercie d'avoir accepté de participer à ce Jury.

Je remercie Françoise BELLEGARDE pour la patience dont elle a fait preuve en écoutant mes exposés très informels.

Je remercie également Hubert PISTRE pour sa participation à l'étude sur l'implémentation de Snobol 4.

Pour réaliser ce travail, j'ai dû recourir à des aides variées :
Messieurs HERVOIN et OBER du C.T.A. à Bruz m'ont été d'un secours précieux.
L'équipe du service d'exploitation d'un ordinateur faisant la pluie et le beau temps du programmeur, je tiens à signaler que j'ai toujours trouvé le soleil à l'I.U.C.A., c'est pourquoi, j'ai le plaisir de remercier Albert CAROMEL et son équipe pour m'avoir aidé dans mon travail qui sortait bien souvent du cadre de la routine.

Enfin, je remercie Bernadette KWASNIEWSKI qui a réalisé ce travail et la sympathique et efficace équipe du secrétariat du Département Informatique de l'I.U.T. qui a mis la dernière main à la pâte.

à Brigitte

SOMMAIRE

INTRODUCTION

0. PRESENTATION RAPIDE DU LANGAGE

1. LES MODELES ET LE FILTRAGE

2. METHODES DE PROGRAMMATION

3. ANALYSE SYNTAXIQUE

4. COMPILATION EN SNOBOL

5. IMPLEMENTATION DE SNOBOL

PROGRAMMES

INTRODUCTION

Le langage Snobol 4, dérivé de Snobol[10] , a été conçu dans les laboratoires de la compagnie BELL par R.E. Griswold, J.F. Poage, I.P. Polansky.

Il était initialement destiné à résoudre des problèmes de calcul algébrique, problèmes qui appartiennent à une classe plus vaste, celle du traitement des chaînes de caractères.

Le langage a été élaboré autour d'outils originaux de traitement de chaînes que sont le filtrage (pattern matching) et les modèles.

Il se distingue, en cela, des langages universels dans lesquels les outils de traitement des chaînes sont des fonctions ajoutées au langage, exception faite pour la concaténation. Dans le cadre d'un contrat du CRI, j'ai travaillé avec Hubert Pistré à la réalisation d'un compilateur Snobol 4 sur CII 10070 en collaboration avec la SESA.

Nous avions défini une implémentation originale, conforme à l'esprit du langage et fondée sur une bonne représentation des principaux objets du langage, les chaînes et les modèles. Cette implémentation est décrite au chapitre 5 .

Elle est complétée par une autre étude qui tend à donner aux modèles une représentation mieux adaptée au traitement et à donner aux chaînes la même représentation que les modèles.

L'étude des applications du langage (ch. 1, 2, 3, 4), constitue la plus grande partie de ce travail.

On caractérise la puissance des différents éléments des modèles en fonction des différentes classes de langage qu'ils permettent de reconnaître

Les programmes écrits en Snobol 4 ont une structure caractéristique, ce qui nous a amené à définir des méthodes de programmation en Snobol 4.

Enfin, on a appliqué ces méthodes pour traiter un certain nombre de problèmes : reconnaissance (ch.1), analyse (ch.3), compilation (ch.4), calcul formel (ch.2), génération de formules, édition de contextes (ch. 2)...

SESA : Société d'Etude des Systèmes d'Automation.

0. PRESENTATION RAPIDE DU LANGAGE

0. INTRODUCTION

Désormais nous emploierons aussi bien les termes de Snobol4 que de Snobol pour désigner le même langage. Snobol4 est souvent classé comme langage de traitement de chaînes de caractères, parfois aussi comme langage de traitement de listes. R.E. Griswold [3] s'élève contre cette classification restrictive, car, si elle avait un sens à la naissance du langage, celui-ci a évolué pour devenir un langage d'application générale à dominante traitement de chaînes de caractères.

Nous allons présenter le langage en montrant ce qu'il a de commun avec les autres langages de programmation et pour l'instant nous n'entrerons pas dans le détail des modèles et du filtrage qui seront étudiés par la suite. Pour avoir plus de détails sur le langage on pourra se reporter à [1].

1. LES OBJETS DU LANGAGE

Il n'y a pas de déclaration de type en Snobol.

1.1. Les objets élémentaires sont : les entiers et les réels en virgule fixe, les chaînes de caractères de longueur arbitraire ainsi que des chaînes généralisées, les modèles, utilisées dans l'instruction de filtrage.

Notations de constantes arithmétiques :

4619 43.186 0.672 14.25

Notations de chaînes de caractères :

'128' 'LES 100 JØURS DE PEKIN'

"14 + 912" "L'APØSTRØPHE"

Pour noter la chaîne vide on peut utiliser deux apostrophes ou deux guillemets successifs ('' ou "") ou bien l'identificateur NULL ou encore dans certains cas l'absence de symbole.

1.2. Chaînes de caractères, variables.

La chaîne de caractères est un objet de base du système SNØBØL4.

On peut comparer son rôle en Snobol4 à celui des entiers dans les ordinateurs, ceux-ci pouvant représenter aussi bien des adresses que leurs contenus.

En effet, il n'est pas fait de distinction a priori entre la chaîne de caractères qu'est le nom d'une variable Snobol4 et les autres chaînes, si ce n'est qu'elle possède l'attribut NØM de VARIABLE. Cet attribut n'est pas attribué de façon statique, on peut créer une variable de façon dynamique à partir d'une chaîne au moyen de l'opérateur $.

Exemple : ARBRE = 'FØRET'
VEGETAL = 'ARBRE'
ØUTPUT = VEGETAL
ØUTPUT = $VEGETAL

Ce morceau de programme éditera ARBRE puis FØRET. Il s'agit d'un adressage indirect appliqué aux chaînes de caractères. En appliquant plusieurs fois $, on peut faire de l'adressage indirect de profondeur arbitraire.

Dans les retours de fonction un retour par nom est réalisé en donnant comme valeur une chaîne de caractères et en précisant qu'elle a l'attribut NØM de VARIABLE au moyen du retour par NØM : NRETURN (voir fonction p.0.6 et exemple d'utilisation ch. 3).

1.3. Les autres objets sont construits dynamiquement grâce à des fonctions standard, c'est le cas :

1.3.1. des tableaux avec la fonction ARRAY. L'affectation: TABLEAU = TAB(C[,V]), construit un tableau de nom TABLEAU dont le nombre de dimensions et leurs bornes sont contenus dans la chaîne C, les éléments du tableau sont initialisés à la valeur V et à défaut à la valeur vide. Un élément de TABLEAU sera référencé T <expression> .

Exemples : 1) T = ARRAY('10') construit un tableau à une dimension de 10 éléments référencés T <1> , T <2> .... T <10> et initialisés implicitement, à la valeur vide.

T = ARRAY(10) a le même effet, car l'argument peut être converti en une chaîne.

2) TAB = ARRAY('-5:5,10,-4:-1','A')
crée un tableau à trois dimensions dont les bornes du premier, par exemple sont -5 et +5 ; tous ses éléments sont initialisés à la chaîne 'A'.

Remarque : La construction d'un tableau étant dynamique on pourra écrire :

TAB2 = ARRAY(CH) où CH est une chaîne de caractères construite à l'exécution par exemple, par l'affectation suivante :
CH = N1 ':' N2 ',' N3 ':' N4 ',' N4 ':' N5

1.3.2. des tables avec la fonction TABLE.

Exemple : X = TABLE(20,10) définit une table à 20 entrées étendue à 30 entrées si nécessaire.

Utilisation : X <'BIAS'> = '4E00'
ØUTPUT = X<ENTREE>

1.3.3. des objets composés avec la fonction DATA qui définit une fonction de génération d'objets composés.

Exemple 1. DATA('CØMPLEX(R,I)') crée la fonction CØMPLEX qui permet de générer des objets à deux champs.

C = CØMPLEX(3.05,6) crée un objet de nom C à deux champs de valeurs respectives 3.05 et 6, référencés par R(C) et I(C).

```
X = R(C) + 4.09
I(C) = I(C) + R(C)
```

Exemple 2. On peut construire une liste chaînée en définissant ELT : fonction de génération d'un élément de liste.
DATA('ELT(CONTENU,PØINTEUR)').
On crée et on initialise un élément en écrivant : X = ELT(INPUT,NULL).
Le champ CØNTENU a pour valeur 80 caractères lus sur carte et PØINTEUR a pour valeur la chaîne vide (NIL).
CØURANT = X (COURANT repère le dernier élément créé)
PØINTEUR(CØURANT) = ELT(INPUT) crée et initialise un nouvel élément et met à jour le champ PØINTEUR de l'élément précédant.
CØURANT = PTR(CØURANT) repère le dernier élément créé.

2. LES INSTRUCTIONS.

Il existe en tout cinq instructions qui sont : l'affectation, le filtrage, le filtrage et remplacement, l'instruction dégénérée et l'instruction END.
L'instruction END est formée de l'étiquette END de blancs et éventuellement d'une étiquette qui indique le début d'exécution.
L'instruction dégénérée permet l'appel de fonctions utilisées, par exemple, comme sous-programmes.
Les trois premières instructions sont les plus importantes, examinons si on retrouve les instructions habituelles des autres langages :

2.1. L'affectation. On la retrouve en Snobol4.

Exemples :

```
X = B ** 2 - 4 * A * C
CH = 'CHAINE LUE :' INPUT
T <I>= S
PLACE = 'NO DE PLACE =' RANG / NB + 1
```

C'est l'instruction d'affectation des autres langages. En Snobol, il est nécessaire de séparer les opérateurs des opérandes. On peut trouver dans le deuxième membre, des expressions ayant pour valeur des chaînes de caractères (l'opérateur de concaténation est le blanc), des expressions ayant pour valeur des modèles.

2.2. Les instructions de branchement.

Il n'y a pas à proprement parler d'instructions de branchements. Mais une instruction Snobol4 peut être formée de trois parties dont la dernière est utilisée pour indiquer un branchement vers la prochaine instruction à exécuter. Si cette partie est vide, il y a exécution en séquence.

2.2.1. Structure d'une instruction Snobol :

[étiquette] blanc(s)[corps de l'instruction] blanc(s)[: branchements]

Exemples :

2.2.2. Branchement inconditionnel.

```
 LABELLE

       X = X + 1 : (LABELLE)
```

2.2.3. Branchement conditionnel.

```
 LABELLE

       X = EQ(X,0)  X + 1 : S(LABELLE) F(LALAIDE)

 LALAIDE
```

La condition porte sur l'évaluation du corps de l'instruction. Si une condition d'échec est détectée alors il y a branchement à l'étiquette qui suit F si elle existe et exécution en séquence sinon. En cas de succès il y a branchement à l'étiquette qui suit S si elle existe et exécution en séquence sinon.

L'instruction PL/I suivante :
IF condition THEN GØTØ E1 ELSE GØTØ E2 ; pourra être traduite par :

	Fonction prédicat : S(E1)F(E2)
Exemple :	GT(DELTA,0) : S(E1) EQ(DELTA,0) : S(E2)

Le corps de ces instructions est réduit à une fonction, c'est un exemple d'instruction dégénérée.
Pour exprimer des conditions sur les chaînes on utilisera l'instruction de filtrage de préférence aux fonctions prédicats.

Exemple :

```
1)       B    CH = TRIM(INPUT)          : F(END)
2)            CH  PØS(0) 'RØSA'  ('E' |'M'| ") RPØS(0) :  S(ØUI) F(NØN)
3)     ØUI    ØUTPUT = 'ØUI'    CH  :(B)
4)     NØN    ØUTPUT = 'NØN'    CH  :(B)
5)     END
```

explications sur le programme :

1) INPUT est une variable associée à la lecture d'une carte, elle a pour valeur les 80 caractères de la carte. La fonction TRIM supprime les derniers blancs. La fin du fichier carte provoque une condition d'échec ce qui entraîne un branchement à END.

2) Cette instruction est une instruction de filtrage, l'opérateur de filtrage est le blanc qui suit CH.CH est le sujet, l'expression à droite de l'opérateur, le modèle.

L'exécution du filtrage se fera avec succès si dans CH il existe une sous chaîne qui satisfait au modèle c'est-à-dire dont le début coïncide avec celui de CH(fonction PØS(0)) composée ensuite de la chaîne 'RØSA' suivie soit de 'E' soit de 'M' soit de la chaîne vide et dont le dernier caractère coïncide avec celui de CH(fonction RPØS(0)).

3) et 4) ØUTPUT est une variable de sortie standard. L'affectation d'une valeur à ØUTPUT entraîne l'impression de cette valeur.

5) END signale à la fois la fin du programme Snobol et la fin d'éxécution.

2.3. Les instructions de bouclage.

On peut regretter leur absence en Snobol. Pour les remplacer le test et l'incrément d'indice se font de façon compacte, soit à traduire :

```
DØ I = 1 TØ  N ;          ; END ;
```

On pourra remplacer cette instruction par :

```
            I = 0
DØI         I = LT(I,N) I + 1 : F(DØEXIT)
            .
            .
            .
            .
ENDI                          : (DØI)
DØEXIT
```

Donnons un autre exemple avec des DØ imbriqués

Exemple : Soit à développer la formule

$$\sum_{i=o}^{n} X_i \cdot \sum_{k=o}^{n-i} X_k \; X_{n-i+k}$$ (cf. annexe 1).

```
            I = -1
DØI         I = LT(I,N) I = 1  : F(DØIEXIT)
            k = -1
DØK         k = LT(k,N - I) k + 1 : F(DØI)
            FØRMULE = '+' FØRMULE 'X' I 'X' k 'X' N - 1 + k
                                  : (DØK)
DØIEXIT
```

2.4. Les entrées-sorties.

Ce sont des entrées-sorties de chaînes de caractères. Nous avons déjà rencontré des entrées/sorties standard INPUT et ØUTPUT. On peut faire des traitements de fichiers séquentiels comme en Fortran en définissant d'autres variables d'entrée-sortie au moyen de fonctions standard.

```
INPUT('DØNNEES', 100,60)
ØUTPUT('RESULTATS',106,'(20A4)')
```

Une référence à DØNNEES entraîne la lecture des 60 premiers caractères d'un enregistrement du fichier dont la référence externe est 100.

L'affectation d'une chaîne de caractères à RESULTATS entraîne l'écriture en format 20A4 d'un enregistrement sur le fichier dont la référence est 106.

2.5. Les fonctions. La fonction standard DEFINE permet de définir dynamiquement des fonctions. Il peut s'agir de fonctions récursives sans que le programmeur ait à le signaler. Syntaxe de DEFINE :
DEFINE ('F(X ,Y,...) L1,L2,... ','ENTREE')
F est le nom de la fonction ; X, Y... les paramètres ; L1,L2,... des variables locales et ENTREE désigne le début du texte de la procédure qui ne suit pas nécessairement la fonction DEFINE.
Si le deuxième argument, le point d'entrée, ne figure pas dans DEFINE, c'est le nom de la fonction qui sert de point d'entrée.
Le retour à l'instruction qui contient l'appel est réalisé par l'étiquette RETURN

Exemple :

```
      DEFINE ('SØM(T)')    : (FIN.SØM)

SØM     I = 0
BØUC    I = LT (I,N)  I + 1 :  F(RETURN)
        SØM = SØM + T <I>      : (BØUC)
FIN.SØM
        ØUTPUT = SØM(A)
END
```

Remarques :

La définition de la fonction est dynamique si bien que l'on peut calculer son nom, ses paramètres, son point d'entrée.

Le texte d'une fonction est directement exécutable sans appeler la fonction un branchement par RETURN provoque toutefois une erreur dans ce cas.

2.6. Instructions propres à Snobol.

Nous venons de passer en revue toutes les caractéristiques que Snobol partage avec les autres langages. Deux instructions n'ont pas de correspondant dans les langages habituels, ce sont le filtrage et le filtrage et remplacement. Elles utilisent toutes deux des modèles qui leur donnent toute leur puissance.

Nous avons rencontré l'instruction de filtrage, l'instruction de filtrage et remplacement en est un prolongement. Syntaxiquement elle est formée en ajoutant à droite du filtrage le signe = et une expression à valeur chaîne. Cette expression sera évaluée à la suite de l'exécution du filtrage. La sous-chaîne reconnue dans le sujet au cours du filtrage sera remplacée par la valeur de l'expression.

Exemple :

```
CH  =  'CECI EST UNE VESSIE'
CH  'VESSIE' = 'CHANDELLE'
ØUTPUT = CH
```

A la suite du filtrage et remplacement CH aura pour valeur la chaîne : CECI EST UNE CHANDELLE qui sera imprimée à l'exécution de l'instruction suivante. Nous utiliserons très souvent le filtrage et remplacement qui facilite la programmation de certains problèmes. Toutefois nous n'en ferons pas une étude systématique, car il est toujours possible de remplacer cette instruction par un filtrage et une affectation.

La puissance de ces instructions dépendant des modèles, nous allons les étudier systématiquement en relation avec les langages formels qu'ils reconnaissent.

1. LES MODELES ET LE FILTRAGE

0. INTRODUCTION

Le modèle est un objet du langage utilisé dans les instructions de filtrage. La puissance de Snobol tient dans les possibilités de description syntaxique offertes par les modèles.

Pour savoir quels langages peuvent être traités en Snobol, il faut savoir quels langages sont reconnus par les modèles. Cette étude est l'objet de ce chapitre.

Les modèles se construisent à l'aide d'expressions. Ils apportent au traitement des problèmes de chaînes de caractères une souplesse comparable à celle des expressions arithmétiques dans les problèmes de calcul numérique.

Comme les autres objets du langage on peut les affecter à des variables. Si un modèle est utilisé dans plusieurs instructions de filtrage, nous préférerons donc l'affecter à une variable plutôt que d'en écrire plusieurs fois l'expression.

1. RECONNAISSANCE DE LANGAGES

L'instruction de filtrage résout le problème suivant : existe-t-il dans une chaîne donnée (le sujet) une sous-chaîne (la chaîne filtrée) qui satisfait à certains critères décrits par le modèle.

Les débuts et les fins des chaînes sujets et des chaînes filtrées ne coïncident pas nécessairement.

Dans les problèmes de reconnaissance on recherche l'identité entre le sujet et la chaîne filtrée, pour cela on imposera à la chaîne recherchée des critères supplémentaires : qu'elle ait même origine que le sujet, ce qui s'exprime par la fonction modèle PØS(0) et qu'elle ait même extrémité, RPØS(0). Si, pour tous les filtrages du programme, on impose que la chaîne recherchée ait même origine que le sujet on pourra supprimer le modèle PØS(0) en positionnant la clé &ANCHØR = 1

D'autre part, nous sous-entendrons souvent l'existence de RPØS(0), sauf dans les programmes qu'on exécutera. Soit V un alphabet et $\alpha = a_1a_2....a_n$ un mot sur V. La chaîne notée '$a_1a_2...a_n$' ou bien "$a_1a_2...a_n$" est aussi un modèle, affectons le à la variable M1.

M1 = '$a_1a_2.... a_n$'

Dans un filtrage le modèle M1 reconnait α .

Soit $\beta = b_1b_2 ... b_p$ un mot sur V. Le modèle M2 = '$b_1b_2.... b_p$' reconnaît β .

On peut composer des modèles au moyen de deux opérateurs :

a) l'alternative |

ainsi M = M1 | M2 reconnaît les mots de L = {α,β}

b) la concaténation dont le symbole est le caractère blanc.

ainsi MU = M1 M2 reconnaît le mot αβ et MUPRIME = MU M les mots αβα , αββ . De façon générale, étant donné un langage fini $L = \{\alpha_i\}_{i=1,n}$ et des M_i reconnaissant respectivement les α_i le modèle M = M1 | M2 | ..... | Mn reconnaît L.

1.2. Reconnaissance des langages réguliers.

Soient L1 et L2 deux langages reconnus respectivement par les modèles M1, et M2, alors L1 ∪ L2 est reconnu par le modèle M1 | M2 et L1L2 par le modèle M1 M2.

La fonction ARBNØ(M1) a pour résultat un modèle qui reconnaît l'itéré $L1^*$ de L1.

Exemple :
```
LETTRE = 'A' | 'B' | 'C' ..... 'Y' | 'Z'
CHIFFRE= '1' | '2' | '3' | '4' | '5' ..... '9' | '0'
CARAC  = LETTRE CHIFFRE
IDENT  = LETTRE ARBNØ(CARAC)
```

IDENT est un modèle qui reconnaît des chaînes qui commencent par une lettre et qui ne contient que des chiffres et des lettres. Il est donc possible de construire des modèles qui reconnaissent des langages composés de langages finis par un nombre fini de réunions, de produits et d'itérations. Tout langage régulier peut donc être reconnu par un modèle.

1.3. Langages algébriques

Soit G = (T,N :: =, X) une C-grammaire où T est le vocabulaire terminal, N le vocabulaire non-terminal, X l'axiome,:: = la relation de production.

A partir de cette grammaire nous allons fabriquer des modèles en associant à chaque non terminal un identificateur (de même nom pour simplifier) et à chaque règle une affectation dont le deuxième membre est un modèle.

Exemple :
```
T = {a,b,c}   X l'axiome
Y::= a | b
X::= aY |c    les règles.
```

Pour faire la correspondance avec les modèles, il est nécessaire de regrouper toutes les règles de même premier membre et de les écrire au moyen d'alternatives.

La grammaire de l'exemple engendre L1 = { aa, ab, c }.
Soient les instructions Snobol suivantes :

```
Y = 'a'  | 'b'
X = 'a' Y | 'c'
```

Ces affectations exécutées, le modèle X aura pour valeur

```
'a'  ('a'  | 'b' )  | 'c'    autrement dit
'aa' | 'ab' |  'c'.
```

Il reconnaît donc L1.

Dans une grammaire l'ordre d'écriture des règles n'a pas d'importance ; ce n'est pas le cas des affectations de modèles qui sont exécutées séquentiellement.

Exemple : Si dans l'exemple précédent on inverse les affectations :

```
X = 'a'  Y | 'c'
Y = 'a'  | 'b'
```

et si l'on suppose que Y n'avait auparavant pas de valeur autre que la valeur implicite (la chaîne vide), l'exécution de la première instruction donne à X la valeur 'a' | 'c' . Un filtrage avec le modèle X reconnaîtra {a,c} mais pas L1.

Dans les mêmes conditions le modèle X = 'a' X | 'a' permet de reconnaître {a} et seulement {a}.

Or dans une grammaire il arrive le plus souvent qu'une référence à un non-terminal soit écrite avant sa définition : il n'est pas possible de faire autrement si la grammaire est récursive, donc si le langage engendré est infini.

Exemple : Sur le même vocabulaire que plus haut X étant l'axiome, soient les règles :

```
Y :: = a X | b
X :: = a Y |  c
```

Le langage engendré est L2 = $\{ a^{2n}c \mid n \geqslant o \} \cup \{a^{2n+1}b \mid n \geqslant o \}$

Soient les affectations associées.

```
Y = 'a'  X | 'b'
                    (1)
X = 'a'  Y | 'c'
```

X a pour valeur le modèle 'a' ('a' 'b') | 'c'
X reconnaît L1 mais pas L2.

Le remplacement de X par sa valeur à l'exécution de la première affectation empêche la récursivité car X à ce moment là a la valeur vide.

Pour permettre la récursivité dans les modèles, il faudra utiliser l'opérateur unaire * dit d'évaluation retardée.

1.3.1. L'opérateur unaire * .

Cet opérateur, placé devant une expression, empêche son évaluation lors de l'exécution d'une instruction d'affectation. Lorsqu'il est exécuté, cet opérateur donne un résultat de type EXPRESSIØN. L'évaluation sera alors réalisée, soit par la fonction EVAL, soit au cours de l'exécution de l'instruction de filtrage.

Exemple d'utilisation de EVAL.

```
      N = 5
      X = *('SIEGE NO ' N)
ØUTPUT = X
ØUTPUT = EVAL(X)
```

L'exécution du premier ØUTPUT provoque la sortie du message EXPRESSIØN et celle du second ØUTPUT la sortie de :

```
SIEGE NO  5
```

Exemple d'application dans les modèles.

Les instructions suivantes :

```
Y = 'a' *X | 'b'
X = 'a' *Y | 'c'
```

élaborent un modèle qui, sous certaines conditions que nous allons développer, reconnaît L2.

1.3.2. Fonctionnement du filtrage, Quicksan et Fullscan

Pour comprendre comment un filtrage avec le modèle X peut reconnaître L2 il faut connaître le fonctionnement du processeur de filtrage.

L'algorithme de base du filtrage est un algorithme d'analyse syntaxique descendant avec retour arrière. (backtracking).

Il parcourt simultanément le sujet et le modèle, si possible, jusqu'à la fin du modèle. Si la fin du modèle est atteinte le filtrage se termine sur un succès.

Dans le modèle, le processeur de filtrage peut se trouver devant plusieurs alternatives. Il choisit, alors la première ; si elle conduit à un échec, il y a retour en arrière dans le sujet et le modèle et le processeur essaye la seconde alternative, etc...

Au cours de l'analyse, il peut rencontrer un modèle non évalué c'est à dire précédé de * ; l'analyse se poursuit alors en parcourant ce modèle et ceci peut être récursif.

En mode FULLSCAN, c'est à dire, lorsque la clé FULLSCAN est positionnée à 1 par l'affectation &FULLSCAN = 1 (par défaut elle est positionnée à 0), le filtrage fonctionne comme nous venons de l'indiquer.

Les inefficacités de cet algorithme sont réduites en mode QUICKSCAN (clé &FULLSCAN positionnée à 0). On améliore dans ce cas l'algorithme en programmant un certain nombre d'heuristiques qui permettent d'éviter l'essai d'alternatives infructueuses. En analyse syntaxique ces heuristiques sont parfois appelées "conditions à priori". Le nombre et la définition en sont laissés aux soins de l'implémentation. La plus évidente est la suivante : on ne choisira pas une alternative si le nombre de caractères qu'elle peut reconnaître est supérieur au nombre de caractères restant dans la chaîne sujet. Elle est réalisée sur toutes les implémentations. Dans le comptage des caractères on considère que le nombre de caractères pouvant être acceptés par un modèle non évalué est au moins égal à 1. Cette estimation risque de provoquer des erreurs si la grammaire dont on veut reconnaître les mots possède des règles vides.

Exemple : Soient les modèles :

```
Y = 'a' *X
X = 'a'| ''
```

La grammaire correspondante, d'axiome Y, engendre $\{a, a^2\}$. Dans un filtrage de modèle Y, en mode QUICKSCAN, le nombre de caractères pouvant être acceptés par Y est estimé à 2, le processeur refusera donc de commencer le filtrage avec a comme sujet.

En mode FULLSCAN une instruction de filtrage dont le modèle correspond à une grammaire récursive à gauche risque de boucler indéfiniment.

Exemple : Le modèle X = *X 'a' |'a' fait boucler le processeur de filtrage quelle que soit la chaîne à analyser. Le programme s'arrêtera sur un débordement de piles utilisées par le processeur.

En mode QUICKSCAN le processeur rencontre une condition d'échec lorsque le nombre de modèles empilés est tel que la chaîne qu'ils devraient reconnaître est de longueur supérieure au nombre de caractères restant à analyser. Cet échec entraîne des retours en arrière, de nouveaux choix et un déblocage. Dans l'exemple précédent le filtrage en Quinckscan reconnaît $\{a^n \mid n > o\}$.

Les modèles permettent de reconnaître, en mode Quickscan les mots engendrés par les C-grammaires sans règles vides. Nous avons vu (p. 1.2) qu'il était immédiat de construire ces modèles à partir des grammaires.

Revenons-en au mode FULLSCAN.

Nous avons dit plus haut que le filtrage en mode FULLSCAN risquait de boucler sur un modèle correspondant à une grammaire récursive à gauche. Ceci nécessite des explications :

Exemple : Le modèle X = 'a' | *X 'a' | 'b' correspondant à une grammaire ayant une règle récursive gauche, X:: = Xa, permet de reconnaître les $\{a^n \mid n > o\}$ mais fait boucler indéfiniment le processeur de filtrage pour tout autre sujet et en particulier un mot de $\{ba^n \mid n > o\}$.

Pour reconnaître ainsi les mots de langages $\{ba^n \mid n > o\}$ il suffit de réordonner les alternatives dans le modèle de la façon suivante :

X = 'a' | 'b' | *X 'a'

En général on peut dire que si une C-grammaire a des règles récursives à gauche, mais pas plus d'une pour un même non terminal, membre gauche de règles, alors on peut construire un modèle qui, dans un filtrage en mode FULLSCAN, reconnaîtra les mots du langage engendré par la grammaire.

Le filtrage bouclera indéfiniment pour tout autre sujet, ce qui provoquera l'arrêt du programme sur un débordement de piles. Si on se contente de faire de la reconnaissance, on emploiera le mode QUICKSCAN si la C-grammaire est sans vide.

Exemple : soit : X = *X 'a' | " . Le modèle X reconnaît en mode QUICKSCAN les $\{a^n \mid n > o\}$. Si la grammaire n'est pas récursive à gauche on pourra utiliser le mode FULLSCAN. On sait d'autre part qu'il est toujours possible de transformer une grammaire récursive gauche en une grammaire équivalente qui ne l'est pas [13]. Comme nous le verrons par la suite on emploie ce mode de filtrage lorsque l'on tient à ce que toutes les alternatives soient essayées pour obtenir des effets annexes à la reconnaissance.

Ces restrictions faites, les modèles permettent de reconnaître les langages algébriques. Pour une C-grammaire donnée il est trivial de construire le modèle correspondant.

1.4. Modèles standard et fonctions-modèle standard.

Les modèles que nous allons citer facilitent certaines descriptions syntaxiques, mais ne permettent pas de reconnaître des langages plus généraux que les langages algébriques. Nous avons déjà vu ARBNØ (§ 1.1). Soient A le vocabulaire formé de tous les symboles permis en Snobol, T un sous ensemble de A, CH une chaîne quelconque formée de tous les caractères de T.

Les fonctions modèles SPAN, BREAK, ANY, NØTANY ont comme argument une chaîne de caractères. Examinons ce qu'elles acceptent avec CH comme argument.

SPAN(CH) accepte au cours du filtrage, la plus grande chaîne ne contenant que des caractères de T.
BREAK(CH) accepte la plus grande chaîne composée de caractères n'appartenant pas à T.
ANY(CH) accepte un caractère s'il appartient à T et NØTANY(CH) accepte un caractère s'il n'appartient pas à T.

Le modèle ARB accepte la chaîne vide au premier passage (cf. fonctionnement du processeur de filtrage), réexaminé à la suite d'un retour arrière il accepte un caractère, puis ensuite deux caractères...

Le modèle BAL accepte toute expression bien parenthésée, en particulier si elle ne commence pas par une parenthèse ouvrante ni fermante il accepte un caractère de la chaîne qui reste à accepter.

Exemples : Considérons les affectations suivantes :

```
            ALPHA      = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
            NUM        = '0123456789'
            ALPHANUM   = ALPHA NUM
1.          IDENTIF    = ANY(ALPHA) (SPAN(ALPHANU)  | NULL)
2.          IDENTIER   = ANY('IJKLMN') (SPAN(ALPHANU) |  NULL)
3.          INSTRUCT   = ';' BREAK (';') ';'
4.          AFFECT     = ARB '=' SUITE
5.          APPELFØN   = IDENTIF BAL
```

Le modèle 1. accepte un caractère alphabétique puis, ensuite, ou bien une chaîne alphanumérique arbitraire, ou bien la chaîne vide.

Le modèle 2. diffère de 1. par le premier caractère accepté.

Le modèle 3. accepte un";"puis n'importe quoi jusqu'à la première occurence d'un";"et enfin ce dernier";".

Le modèle 4. accepte n'importe quoi suivi de caractère = puis, enfin, ce qui est accepté par SUITE.

Le modèle 5. accepte la chaîne filtrée par le modèle 1. puis une expression bien parenthésée.

Les fonctions TAB, RTAB, LEN ont un entier comme argument.

TAB et RTAB sont particulièrement intéressantes pour reconnaître les différents champs d'une structure (voir l'exemple dans le § 1.4.).

TAB(n) accepte tous les caractères jusqu'à et y compris le $n^{ième}$ caractère du sujet si celui-ci n'a pas encore été accepté dans le filtrage, sinon provoque un ECHEC. RTAB(n) joue le même rôle, mais les caractères sont comptés à partir de la fin.

LEN(n) accepte les n premiers caractères de la chaîne restant à filtrer, s'ils existent sinon provoque un ECHEC.

REM accepte le reste de la chaîne restant à filtrer.

Enfin, les modèles FAIL, FENCE, ABØRT, SUCCEED permettent de diriger d'une certaine façon le processeur de filtrage.

Leur utilisation intempestive est à proscrire comme l'utilisation intempestive du GØTØ dans les langages usuels.

FAIL provoque une condition d'échec et peut être utiliser pour forcer le processeur de filtrage à parcourir toutes les alternatives d'un modèle.

Cela fournit un moyen de sortir toutes les analyses syntaxiques pour un langage engendré par une grammaire ambiguë.

FENCE empêche tout retour en arrière, ABØRT provoque l'arrêt du filtrage; ils peuvent être utilisés pour optimiser un filtrage.

SUCCEED est équivalent à NULL| NULL| .... ; en association avec FAIL il permet de faire boucler indéfiniment le processeur de filtrage. Un exemple d'utilisation est donné dans[1]page 61 , dans un programme qui sort un dessin infini (limité par le nombre maximum de pages) représentant des dents de scie... Expression d'un peu de fantaisie ou manière d'obtenir à bas prix des motifs pour tapisser une pièce?

1.5. Affectations dans les modèles.

Les modèles permettent de reconnaître des chaînes en donnant une description, par des éléments de modèles, des sous-chaînes qui les composent. Dans beaucoup d'applications, après le sujet reconnu, il est nécessaire de pouvoir utiliser les sous-chaînes qui le composent. On peut le faire en associant des variables aux éléments de modèle. Cette association est matérialisée par l'un des opérateurs "." et $.

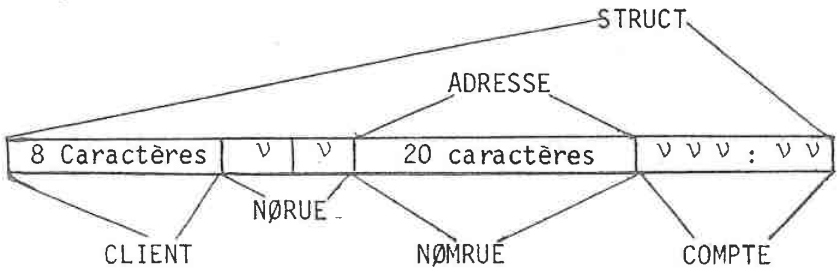
A la fin du filtrage la variable associée prend la valeur de la sous-chaîne filtrée par l'élément de modèle.

Exemples: 1) Le modèle 4 page 1.7permet de reconnaître une affectation. Pour traiter ce qui est à gauche du signe = il faudra modifier le modèle en insérant une variable :

```
AFFECTATIØN  = ARB .VAR '=' SUITE.
```

Après un filtrage de modèle AFFECTATIØN et de sujet la chaîne : 'T(I)=B*B -4*D**2', VAR aura pour valeur la chaîne 'T(I)'.

2) soit la structure suivante



Sa description PL/I donnerait :

```
DCL  1  STRUCT,
     2 CLIENT CHAR(8),
     2 ADRESSE,
     3 NØRUE  P'99',
     3 NØMRUE  CHAR(20),
     2 CØMPTE    P'999V99' ;
```

En PL/I une lecture 'INTØ STRUCT' permet la décomposition de la chaîne lue (l'enregistrement). En Snobol 4 on exprimera la structure par un modèle M. Le découpage se fera dans une instruction de filtrage ayant comme modèle M et comme sujet l'enregistrement.

Il est possible de construire un modèle qui aura même effet que la structure ci-dessus, sans rendre compte toutefois de la position du point virtuel pour CØMPTE.

Nous nous proposons d'en faire un peu plus en supprimant pour CLIENT et NØMRUE les blancs inutiles à droite.

Pour des raisons de clarté, nous allons décomposer le modèle.

```
          NU       =   '0123456789'
1.        NCØMPTE  =   SPAN(NU) . CØMPTE
2.        MNØMRUE  =   (BREAK (' ') . NØMRUE TAB(8)) | TAB(8) . NØMRUE
3.        MCLIENT  =   (BREAK (' ') . CLIENT TAB(8)) | TAB(8) . CLIENT
4.        MNØRUE   =   (ANY(NU) ANY(NU)) . NØRUE
5.        M        =   (MCLIENT (MNØRUE  MNØMRUE)  . ADRESSE NCØMPTE)
  +                      . STRUCT
```

Le modèle accepte une chaîne numérique qui est ensuite affectée à NØMRUE.

Le modèle 2. est formé de deux alternatives. La première réussit si la longueur du nom du client est strictement inférieure à 8 caractères, donc si le champ CLIENT est complété par des blancs ; les huit premiers caractères se sont alors acceptés et CLIENT aura pour valeur le nom du client sans les blancs.

La seconde alternative est essayée si les huits premiers caractères ne contiennent pas de blancs. Cette chaîne est alors affectée à CLIENT ; elle échoue si le sujet contient moins de huit caractères.

Dans le modèle 4. les parenthèses délimitent un sous-modèle qui accepte deux caractères numériques, ils seront affectés à NØRUE.

Enfin dans le modèle 5. ADRESSE aura pour valeur la chaîne filtrée par le sous-modèle précédant l'opérateur "." et STRUCT est associé au modèle tout entier.

On modifiera CØMPTE en y insérant un point réél en vue de calculs ultérieurs par une instruction de filtrage et remplacement, de la façon suivante :

```
CØMPTE   LEN(3)   .  V1 = V1  '.'
```

Le filtrage accepte trois caractères qui sont affectés à V1, puis ils sont remplacés par V1 concaténé à '.'.

1.6. L'opérateur binaire $.

Dans les exemples précédents on aurait obtenu les mêmes résultats en remplaçant l'opérateur "." par l'opérateur"$". Ces deux opérateurs sont pourtant différents.

Avec "." l'affectation n'est réalisée qu'à la fin du filtrage s'il se termine avec succès, c'est l'affectation conditionnelle. Tandis qu'avec"$" l'affectation s'effectue dynamiquement au cours du filtrage.

Une première application de"$"consiste à associer la variable de sortie ØUTPUT au sous-modèle SM.

A chaque essai du sous-modèle SM il y a alors impression de la chaîne acceptée par SM.

Cette technique est intéressante pour mettre au point des programmes Snobol.

Mais cet opérateur a un pouvoir beaucoup plus important dans le filtrage, celui de faire "varier" le modèle. Nous allons le voir sur des exemples :

Exemple 1. On se propose de reconnaître des mots dont les n premiers caractères sont identiques aux n derniers. Le modèle suivant permettra de reconnaître de tels mots : M = LEN(N) $ MEME ARB *MEME
En effet, au cours du filtrage les n premiers caractères sont acceptés et affectés à la variable MEME, ARB acceptera une chaîne telle qu'elle soit suivie de la sous-chaîne MEME.

Ce nouvel outil va nous permettre de simplifier certaines descriptions syntaxiques.

Exemple 2. Singulier et pluriel. Soient les instructions suivantes :

```
  ART = 'LE'  ('' | 'S') $  MP
  ADJ = '  PETIT'  |  'GRAND'
SUBST = 'CHEMIN' |  'BØULEVARD'
SUJET = ART ADJ *MP  SUBST *MP
```

Le modèle SUJET permet de reconnaître les phrases suivantes :

$$\text{LE}\begin{Bmatrix}\text{PETIT}\\ \text{GRAND}\end{Bmatrix}\qquad\begin{Bmatrix}\text{CHEMIN}\\ \text{BØULEVARD}\end{Bmatrix}$$

$$LES\begin{Bmatrix}PETITS\\GRANDS\end{Bmatrix}\quad\begin{Bmatrix}CHEMINS\\BOULEVARDS\end{Bmatrix}$$

Sans l'emploi des modèles "variables" il aurait fallu écrire deux fois les modèles.

Les modèles variables permettent de reconnaître des langages plus généraux que les langages algébriques.

Exemple 3. Le langage $P = \{a^n b^p a^n b^p \mid n \geq o \quad p \geq o\}$ n'est pas algébrique. Pour construire un modèle qui le reconnaisse on construit un modèle M1 = '' | 'a' *M1 | *M1 'b' qui reconnaît les $a^n b^p$. Le modèle M = M1 $ VM1 *VM1 reconnaît alors P.

L'étude formelle des langages reconnus par les modèles variables n'est pas facile à faire à cause, essentiellement, de l'ordre dans lequel le modèle est examiné par le processeur de filtrage. Quand il s'agit de reconnaître un langage dans les mots duquel il y a des répétitions de sous-chaînes il est parfois plus agréable d'utiliser des modèles variables même si le langage est engendré par une C-grammaire qui est parfois difficile à trouver.

Exemple 4. On connaît bien la grammaire qui engendre les $\{a^n b^p \mid n \geq o \; p \geq o\}$ soit M = XY le modèle qui accepte les mots de ce langage, X acceptant les a^n et Y les b^p. Il est alors plus rapide d'en déduire le modèle qui accepte les mots de $P = \{a^n b p a^n \mid n \geq o \quad p \geq o\}$ que de chercher sa grammaire. En effet MP = X $ VX Y *VX reconnaît les mots de P.

1.7. L'opérateur unaire @

Au cours du filtrage la position de l'analyse dans le sujet est repérée par une variable appelée le curseur. L'opérateur permet d'affecter la valeur du curseur à une variable.

Exemple : Un modèle simple pour reconnaître les $a^n b^n c^n$ est le suivant :

```
M = SPAN ( 'A') @I SPAN ( 'B') PØS (*(2  *  I)) SPAN ( 'C')  PØS(*(3 *  I))
    *RPØS(O)
```

SPAN (CH) est une fonction standard de type modèle dont l'argument est une chaîne et qui accepte la plus grande chaîne formée de caractères contenus dans CH.

On se rappellera que l'opérateur * permet au processeur de filtrage de calculer la valeur des expressions (2 * I) et (3 * I).

Dans la suite nous verrons d'autres modèles pour reconnaître les $a^n b^n c^n$. En fait la reconnaissance des $a^n b^n c^n$ ne sera qu'un prétexte à l'étude des modèles construits par des fonctions.

1.8. Modèles définis par des fonctions.

Dans les programmes que nous avons écrit jusque maintenant les modèles sont statiques, évalués une seule fois avant leur utilisation dans les filtrages. C'est souvent le cas dans un grand nombre d'applications de Snobol.

Néanmoins nous allons donner un exemple où le modèle est modifié au cours de l'exécution et en plus, nous utiliserons une fonction pour obtenir un modèle.

Il s'agit d'un programme qui simule l'élève dans un dialogue enseignant enseigné. Le thème de l'apprentissage est la classification des êtres en trois règnes : animal, végétal, minéral.

Le maître apprend à l'élève qu'un être appartient à un certain règne, il contrôle également les connaissances de l'élève. Les paroles du maître sont données sur cartes. Les réactions de l'élève sont simulées par le programme. Les règnes sont représentés par des modèles : ANIMAL, VEGETAL, MINERAL.

Lorsque le maître "apprend" que l'être x est un végétal ceci est réalisé en modifiant le modèle VEGETAL.VEGETAL étant la valeur de CLASSE :

```
$CLASSE = $CLASSE | x
```

Aux questions du maître "l'élève répond" en utilisant le modèle approprié. Par exemple, la question : EST-CE QUE y EST UN ANIMAL entraînera l'exécution du filtrage.

```
y ANIMAL
```

Le programme "élève" avant de répondre VOUS NE ME L'AVEZ PAS APPRIS, vérifiera s'il appartient à un autre règne. Le modèle "autre règne que R" est une sorte de complémentaire de R dans le modèle ANIMAL | VEGETAL | MINERAL. On le calcule à l'aide d'une fonction CØMPL(R).

Cette fonction ôte 'R | ' de la chaîne 'ANIMAL | VEGETAL | MINERAL' et convertit cette chaîne en modèle.

Exemple : voir programme page A.2.

Les fonctions à valeurs de modèles, précédées de l'opérateur * permettent de construire des modèles qui pourront se modifier au cours du filtrage. Cela permet de rendre compte de caractères contextuels dans des descriptions syntaxiques. Nous y reviendrons dans un prochain paragraphe.

L'étude de tels modèles nous amenera à étudier des doubles grammaires et l'utilisation de modèles dans la reconnaissance syntaxique des langages engendrés.

1.9. Modèles définis de façon récursive, par des fonctions.

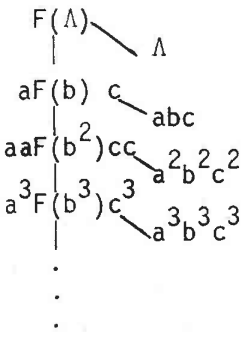
Ces fonctions modèles peuvent être définies de façon récursive, de façon un peu particulière pour que le contrôle de leur évaluation n'échappe pas au filtrage qui les utilise, c'est ce que nous verrons dans un exemple.

Les définitions de telles fonctions modèles qui s'utilisent entre elles forment des systèmes comparables aux systèmes à point fixe que l'on utilise dans la définition des langages algébriques. Les non-terminaux y sont remplacés par des fonctions.

Nous nous intéressons à ces modèles car ils sont faciles à programmer et permettent de reconnaître, comme nous allons le voir des langages plus généraux que les langages algébriques.

Exemple : F(X) = a F(bX) c | X

L'évaluation de F avec pour argument la chaîne vide engendre les évaluations suivantes de manière récursive.



Le modèle F(Λ) permet de reconnaître les $\{a^n b^n c^n \mid n \geq 0\}$ à condition de s'arranger pour que la récursivité ait lieu sous le contrôle du filtrage comme dans le programme suivant tiré de [2].

```
          DEFINE ('F(X)EXP')    : (FIN.F)
F         EXP = "'" X " ' | 'A'  *F('B" X" ') 'C'"
          F = CØNVERT (EXP, 'EXPRESSIØN') : (RETURN)
FIN.F     .
          .
          .
          CH PØS(0) F()  RPØS(0) :.....
          .
          .
          .
END
```

Explications : F() est évalué pendant le filtrage le résultat sera le modèle NULL | 'A' *F('B') 'C' et le second appel de F pour l'argument 'B' sera effectué par le filtrage si CH n'est pas vide et si son premier caractère est 'A'.

Par la suite nous n'entrerons pas dans les détails de la programmation et nous admettrons que lorsqu'on écrit F = X | 'A' *F('B' X) 'C', le résultat de la fonction sera le même que ci-dessus et nous

simplifierons l'écriture des caractères en utilisant des petites lettres. Par exemple, a au lieu de 'A'.

Revenons sur le principe de génération des $a^n b^n c^n$ avec une fonction récursive.

Posons $F(\Lambda) = F_o$ et pour tout $n > o$ $F(b_n) = F_n$

$F(\Lambda)$ est équivalent au modèle Fo défini par le système infini suivant :

$$\begin{aligned} Fo &= a\ F_1\ c \mid \Lambda \\ F_1 &= a\ F_2\ c \mid b \\ F_2 &= a\ F_3\ c \mid b^2 \\ &\vdots \\ F_n &= a\ F_{n+1}\ c \mid b^n \end{aligned}$$

A partir de $F(\Lambda)$, $F(X) = aF(bX)c \mid X$ engendre une infinité de règles, ce qui n'est pas sans rappeler la façon dont sont engendrés les mots par une syntaxe de Van Wijngaarden [17].

Ces fonctions engendrent des modèles mais on peut également considérer qu'elles engendrent des langages. Nous allons étudier ces langages en les définissant à l'aide d'une grammaire.

1.9.1 Les F-grammaires.

1.9.1.1. Définitions.

Définition 1. F-grammaire.

. T est un ensemble fini de terminaux.

. N un ensemble fini de symboles fonctionnels. Les symboles ( ) et X n'appartiennent ni à T ni à N on pose $T' = T \cup \{X\}$ et $V = V' \cup N\ (T'^*)$

. Pour un certain $H \in N$ et un certain $\mu \in T'^*$ on définit une F-grammaire que l'on notera $G_{H(\mu)}$ par le quadruplet $(N, T, :, H(\mu))$ où : est une relation de $N(X)$ dans V^* dont tous les couples, appelés schémas de règle, sont en nombre fini. $H(\mu)$ est l'axiome.

Définition 2. Règle.

Une règle est un couple de $N\ (T^*) \times V^*$ notée $\alpha :: = \beta$ telle qu'il existe un schéma de règle

$H(X) : \gamma$ et $\nu \in T^*$ tels que $\alpha = H(\nu)$ et β est obtenu en remplaçant dans γ toutes les occurrences de X par ν. Ce remplacement est un homomorphisme de V^* dans V^* que l'on notera σ_ν.

Nota : Ces définitions correspondent exactement à ce que l'on peut programmer en Snobol4. Pour cette raison on n'accepte pas de schéma

$G(\gamma) : \delta$ où $\gamma \neq X$.

Exemple : T = {a,b} est l'alphabet terminal d'une F-grammaire, G et H les symboles fonctionnels non terminaux et G(X) : aH(aX) et G(X) : X ses schémas de règles.

G(Λ) :: = aH(a), G(aba):: =aba, G(aba):: = aH(aaba) sont des règles.

Le langage engendré va être défini comme pour les langages de Chomsky, seul diffère le fait que l'on a une infinité de règles.

Définition 3. Réécriture

On appelle réécriture la relation binaire sur V^* notée $\succ\!\!\!-$ définie par :

$\alpha \succ\!\!\!- \beta \Leftrightarrow (\exists\, \lambda, \mu, \gamma \in V^*, \nu \in T^*, H \in N \mid \alpha = \lambda H(\nu) \text{ et } \beta = \lambda\gamma\mu \text{ et } H(\nu) :: = \gamma)$

La relation $\overset{*}{\succ\!\!\!-}$ est la fermeture transitive de $\succ\!\!\!-$.

Définition 4. Langage engendré.

Le langage engendré par une F-grammaire $G_{H(\mu)}$ est, avec les notations définies ci-dessus, le sous-ensemble $L_{H(\mu)}$ de T^* défini par :

$L_{H(\mu)} = \{\alpha \in T^* \mid H(\mu) \overset{*}{\succ\!\!\!-} \alpha\}$

Terminologie. On appelle argument tout élément de T'* placé entre parenthèses.

Exemple 1. Soit $G_{H(\Lambda)}$ une F-grammaire où N = { H}
T = {a,b,c} et H(Λ) l'axiome et H(X) : X, H(X) : aH(bX)c les schémas de règles.
Alors $L_{H(\Lambda)} = \{a^n b^n c^n \mid n \geq o\}$ et ce n'est pas un langage algébrique.
En effet, démontrons que $L_{H(\Lambda)} \subset \{a^n b^n c^n \mid n \geq o\}$ par récurrence sur la longueur des dérivations.

Tout mot de V^* qui dérive de H(Λ) par une dérivation de longueur n est de la forme $a^{n-1} b^{n-1} c^{n-1}$ ou $a^n H(b^n) c^n$. C'est vrai pour n = 1, supposons le vérifié jusqu'à l'ordre n. Si δ dérive de H(Λ) par une dérivation de longueur n + 1 on a :

$H(\Lambda) \overset{n}{\succ\!\!\!-} a^n H(b^n) c^n \succ\!\!\!- \delta$ et δ est soit $a^n b^n c^n$, soit $a^{n+1} H(b^{n+1}) c^{n+1}$.

Démontrons que $\{a^n b^n c^n \mid n \geq o\} \subset L_{H(\Lambda)}$. On démontre que quelquesoit n $a^n H(b^n) c^n$ dérive de H(Λ).

C'est vrai pour n = 0 supposons le vrai jusqu'à l'ordre n.

Comme $a^{n+1} H(b^{n+1}) c^{n+1} = a^n aH(b^{n+1}) c\ c^n$ et qu'il existe une règle $H(b^n) :: = aH(b^{n+1}) c$ alors $H(\Lambda) \overset{*}{\succ\!\!\!-} a^n H(b^n) c^n \succ\!\!\!- a^{n+1} H(b^{n+1}) c^{n+1}$. D'où le résultat pour les $a^n b^n c^n$.

Exemple 2. N = {F, G} , T = {a,b,c,d,e,f}

Lemme 3. Si $G_{H(\mu)}$ est une F-grammaire sans Λ alors il existe une F-grammaire $\tilde{G}_{H(\mu)}$ équivalente à $G_{H(\mu)}$ et n'admettant pas de schéma du type $H(X) : A(\nu)$.

On définit $\tilde{:}$ dans $\tilde{G}_{H(\mu)}$ comme pour les grammaires de Chowsky

$A(X) \tilde{:} \varphi \Leftrightarrow ((|\varphi| \geq 2$ ou $\varphi \in T')$ et $(\exists B \in N, \nu \in T'^{*})$

$A(X) \overset{*}{\underset{G_{H(\mu)}}{\succ}} B(\nu)$ et $B(\nu) :: = \varphi)$

Lemme 4. Si L est engendré par une F-grammaire alors il existe une F-grammaire sans Λ et sans schéma du type $H(X) : A(\nu)$ engendrant L.

Enfin il résulte de ces lemmes que si α est un mot du langage engendré par une F-grammaire $G_{H(\mu)}$ il existe une dérivation de $H(\mu)$ à α de longueur au plus égale à $2\,|\alpha|$.

Par un nombre fini de dérivations, il est possible de savoir si un mot sur T* appartient au langage engendré.

1.10. W-grammaires et F-grammaires.

1.10.1. Rappel sur les W-grammaires [17].

Les W-grammaires ont été introduites par Van-Wijngaarden pour décrire la syntaxe d'Algol 68. Ce sont des doubles grammaires qui généralisent les C-grammaires.

Pour définir une W-grammaire on se donne deux vocabulaires finis N et M et une partie finie T de MM^* .

Les éléments de N sont appelés des métanotions. On note V la réunion de N et de M.

On donne également une relation de $(VV^* - T)$ dans $(V^*)^*$ notée ":" dont les couples, en nombre fini, sont appelés schémas de règles, et une C-grammaire sans axiome G = (N,M,::) appelée métagrammaire.

Les éléments de V^* dans le second membre des schémas sont appelés des hypernotions ; dans l'écriture des schémas on les sépare par des virgules.

Pour chaque Xi appartenant à N on note L_{Xi} le langage qui dérive de X_i dans la métagrammaire. Des schémas de règles on déduit des règles en remplaçant les éléments Xi de N par les mots de L_{Xi} uniformément dans les deux membres. A partir d'un élément donné de VV^* , l'axiome de la W-grammaire, on engendre, grâce aux règles ainsi obtenue une partie de T^*. C'est le langage défini par la W-grammaire.

1.10.2 Proposition 4.

Pour toute F-grammaire engendrant un langage L, il existe une W-grammaire engendrant le même langage.

Cette W-grammaire est ainsi définie :

X est le seul élément de N.

Le vocabulaire M est composé des symboles ( ) et des symboles fonctionnels de la F-grammaire. Les schémas sont les mêmes. Ici il n'est pas nécessaire de séparer les hypernotions par des virgules, la notation $F(\alpha)$ est sans ambiguïté.

Le vocabulaire T est identique à celui de la F-grammaire.

Les règles de la métagrammaire sont telles que de X dérive T .

La W-grammaire a même axiome que la F-grammaire.

Exemple : De la F-grammaire p. 1.15 on déduit la W-grammaire suivante :

$$N = \{ X\} \qquad T = \{a,b,c\} \qquad M = \{F,),(a,b,c\}$$

dont les schémas de règles sont :

$$F(X) \ : \ aF(bX)c \ | \ X$$

l'axiome est $F(\Lambda)$.

Les règles de la métagrammaire sont :

$$X :: aX \ | \ bX \ | \ cX \ | \ \Lambda$$

On déduit de cette W-grammaire les règles :

$$F(\alpha) :: = aF(b\ \alpha)c \ | \ \alpha \text{ pour tout } \alpha \in T$$

Cette W-grammaire engendre les $a^n b^n c^n$; la démonstration est analogue à celle qui a été faite pour la F-grammaire correspondante. Comme seules sont utilisées dans les réécritures les règles $F(b^n) :: = aF(b^{n+1})\ c | \ b^n$, on ne change pas le langage engendré en réduisant la métagrammaire aux règles suivantes : $X :: bX \mid \Lambda$

1.10.3 Proposition 5.

Les W-grammaires et les F-grammaires ne sont pas équivalentes.

En effet, nous avons démontré que les langages engendrés par les F-grammaires étaient décidables et on sait qu'il n'en est pas de même pour les W-grammaires [16]. Néanmoins, nous allons chercher à utiliser des fonctions ayant pour valeurs des modèles pour construire des accepteurs pour certaines W-grammaires.

Nous montrerons sur des exemples comment utiliser ces fonctions.

1.10.4. Construction de modèles pour reconnaître les mots engendrés par certaines W-grammaires.

1.10.4.1. Cas des $a^n b^n c^n$. On engendre les $a^n b^n c^n$ de façon plus naturelle que dans l'exemple du § 1.10.2 par la W-grammaire suivante :

N = {X}; M = {F,G,H,U,(,),R,a,b,c}

T = {a,b,c}

Les règles de la métagrammaire sont : X:: Λ | UX

Les schémas de règles :

1. R : MX
2. MX : FX,GX,HX
3. FUX : a,FX
4. F : Λ
5. GUX : b,GX
6. G : Λ
7. HUX : c,HX
8. H : Λ

L'axiome est R.

Soit P = NULL | 'U' *P le modèle associé à la métagrammaire.

Le schéma 2 sera traduit par la définition d'une fonction ayant pour valeur un modèle : M(X) = F(X) G(X) H(X).

Les schémas 3-4, 5-6, 7-8 également mais en effectuant un filtrage sur l'argument, comme nous allons le voir.

Notons $\in$ l'opération de filtrage : $x \in y$ signifie : x est accepté par le modèle y.

On définit 3-4 par une fonction de la manière suivante :

F(y) = si y = ΛalorsΛ
sinon si y = 'U'z où $z \in P$ alors 'a' *F(z)
sinon FAIL.

R est l'axiome, le modèle qui devra traduire R doit permettre de générer :

M() | M(U) | $M(U^2)$ | ....

On peut engendrer toutes ces chaînes par récurrence grâce à la fonction R(X) = M(X) | R(UX), appelée avec pour argument le mot vide.

R() accepte alors les $a^n b^n c^n$ et fait boucler le processeur de filtrage pour tout sujet n'appartenant pas aux $a^n b^n c^n$. Nous donnerons une explication de ce résultat un peu plus loin.

1.10.4.2. Programme voir page 1.20.

1.10.4.3. Cas général.

Les fonctions ayant pour valeur des modèles, définies en commençant par effectuer un filtrage sur l'argument, constituent un outil fondamental pour construire l'analyseur de certaines W-grammaires.

Accepteur du langage engendré par la W-grammaire définie page 1.19 :

```
          MX = SPAN('U') | ''                                          1
        DEFINE('R(X)')    :(FIN.R)                                     2
R       A =  *M('  X  ') | *R('U' '  X  ')                             3
        R = CONVERT(A,'EXPRESSION')          :(RETURN)                 4
FIN.R   DEFINE('M(X)')   :(FIN.M)                                      5
M       A =  *F('  X  ') *G('  X  ') *H('  X  ')                       6
        M = CONVERT(A,'EXPRESSION')          :(RETURN)                 7
FIN.M                                                                  8
        DEFINE('H(X)')    :(FIN.H)                                     9
H          H = IDENT(X)             :S(RETURN)                        10
        X POS(0) 'U' MX . B RPOS(0)    :F(FRETURN)                     11
        A =  'C' *H('  B  ')                                           12
        H = CONVERT(A,'EXPRESSION')   :(RETURN)                        13
FIN.H                                                                  14
        DEFINE('G(X)')    :(FIN.G)                                     15
G          G = IDENT(X)             :S(RETURN)                        16
        X POS(0) 'U' MX . B RPOS(0)    :F(FRETURN)                     17
        A =  'B' *G('  B  ')                                           18
        G = CONVERT(A,'EXPRESSION')   :(RETURN)                        19
FIN.G                                                                  20
        DEFINE('F(X)')    :(FIN.F)                                     21
F          F = IDENT(X)             :S(RETURN)                        22
        X POS(0) 'U' MX . B RPOS(0)    :F(FRETURN)                     23
        A =  'A' *F('  B  ')                                           24
        F = CONVERT(A,'EXPRESSION')   :(RETURN)                        25
FIN.F                                                                  26
          &FULLSCAN = 1                                                27
B       CH = TRIM(INPUT)        :F(END)                                28
          OUTPUT = ' DONNEE : ' CH                                     29
        CH POS(0)    R() RPOS(0)       :S(OUI)F(NON)                   30
OUI       OUTPUT = ' LA DONNEE EST ACCEPTEE'        :(B)               31
NON     OUTPUT =      CH 'NON'    :(B)                                 32
END                                                                    33
```

```
NO ERRORS DETECTED DURING COMPILATION
```

Résultats :

```
 DØNNEE : AABBCC
 LA DØNNEE EST ACCEPTEE
 DØNNEE : ABC
 LA DØNNEE EST ACCEPTEE
 DØNNEE : AAAABBBBCCCC
 LA DØNNEE EST ACCEPTEE
 DØNNEE : ABBAC
1ERRØR TERMINATIØN IN STATEMENT  23 AT LEVEL  1
 STACK ØVERFLØW.
```

```
 SNØBØL4 STATISTICS SUMMARY
```

```
  1224 MS. CØMPILATIØN TIME
 24264 MS. EXECUTIØN TIME
   814 STATEMENTS EXECUTED,      131 FAILED
     0 ARITHMETIC ØPERATIØNS PERFØRMED
   134 PATTERN MATCHES PERFØRMED
    12 REGENERATIØNS ØF DYNAMIC STØRAGE
 29.81 MS. AVERAGE PER STATEMENT EXECUTED
```

Nous allons voir comment les appliquer pour traduire des schémas de plus en plus complexes. Nous rencontrerons un certain nombre de problèmes que nous résoudrons si c'est possible et qui, sinon, nous permettront de mettre en évidence les limites de ces outils. Dans la traduction des schémas, aux premiers membres des règles correspondent des définitions de fonctions et à certaines hypernotions des appels de fonctions.

Nous allons d'abord nous arranger pour que toutes les hypernotions commencent par une lettre de M.

Si dans un schéma S une hypernotion commence par une métanotion Y , on remplace ce schéma par autant de schémas qu'il y a de métarègles de premier membre Y . Ces schémas sont obtenus à partir de S en remplaçant uniformément dans les deux membres les occurrences de Y par les seconds membres des métarègles dont Y est le premier membre.

Exemple : Soit une W-grammaire dont les métarègles sont :

Y :: a Y| d a,b,c,d appartenant à M.

Le schéma Y ac : bc Y d, Y , b sera remplacé par

les schémas aYac : bca Y d, a Y , b
dac : bcdd,d,b

le schéma a Y : Y ,b Y a par

les schémas aa Y : a Y,ba Y a
ad : d,bda

le schéma a : c, Y b par les schémas a : c,aY b | c,adb

On obtient bien, ainsi, une W-grammaire équivalente dont les hypernotions commencent par une lettre de M car on peut toujours transformer la métagrammaire de sorte que les seconds membres des règles commencent par un élément terminal [13] .

A) Traduction des premiers membres des schémas de règles.

Dans cet alinéa nous ne détaillerons pas les transformations que l'on effectue sur le deuxième membre m_2 d'un schéma de règle. On notera $t(m_2)$ cette transformation.

1. Tous les premiers membres commencent par une lettre différente.

- Soit le schéma aX : m_2 où a $\in$ M et X $\in$ N

On définit la fonction : a(X) = si X $\in$ L_X alors $t(m_2)$ sinon FAIL

- Soit le schéma abXc : m_2

On définit alors une fonction a(z) = si z se décompose en bαc où $\alpha \in L_X$ alors $t(m_2)$ sinon FAIL. Une des opérations de t consistera à remplacer dans m_2 toute occurrence de X par α.

- Soit le schéma abXcYd : m_2

La solution consiste à définir la fonction :

a(z) = si z se décompose en b αc β d où $\alpha \in L_X$ et $\beta \in L_Y$ alors $t(m_2)$ sinon FAIL.

Cette solution n'est valable que si la décomposition est unique. La recherche de toutes les solutions est un problème facilement soluble en Snobol mais il serait très lourd d'insérer sa solution dans l'analyseur.

Nous imposerons donc qu'il y ait unicité de la décomposition quand on aura plus d'une métanotion dans le premier membre d'un schéma de règle

- Soit le schéma abc : m_2

On traite ce schéma de la même façon que les précédents en définissant une fonction a :

a(z) = si z = bc alors $t(m_2)$
sinon FAIL

En résumé on traite le premier membre d'un schéma de règle en définissant une fonction qui a pour nom la première lettre du schéma, le reste étant l'argument.

2. Plusieurs schémas commencent par la même lettre.

- Soient les schémas abXd : m_2^1
ac : m_2^2
aYd : m_2^3

On les regroupe en une seule définition de fonction a. Si l'argument est égal à c il n'y a qu'un résultat possible.
Si l'argument appartient à bL_Xd il peut également appartenir à L_Yd.

D'où la définition suivante :

a(z) = si z = c alors $t(m_2^2)$
sinon si z se décompose en b αd où $\alpha \in L_X$
alors $t(m_2^1) \mid a'(z)$
sinon a'(z)

a'(z) = si z se décompose en αd où $\alpha \in L_Y$ alors $t(m_2^3)$ sinon FAIL

Notons que le résultat de a(z), si z = b αd, est le modèle $(t(m_2^1) \mid a'(z))$ c'est à dire que si $t(m_2^1)$ échoue le filtrage essaiera a'(z). Si on est certain que l'intersection de bL_Xd et de L_Yd est vide (dans le cas général le problème est indécidable) alors la définition peut être simplifiée, ce qui donne :

a(z) = si z = c alors $t(m_2^2)$
sinon si z se décompose en bαd où $\alpha \in L_X$
sinon $t(m_2^1)$
sinon si z se décompose en αd où $\alpha \in L_Y$
alors $t(m_2^3)$

B) traduction des seconds membres des schémas de règles.

On traite les différentes hypernotions du deuxième membre les unes après les autres.

Si l'hypernotion est un terminal, elle sera traduite par une chaîne de caractères.

On traduit les autres hypernotions par des appels de fonctions dont le nom est constitué par la première lettre et l'argument par le reste de l'hypernotion.

Exemple : a,bc,donnera a(bc). La solution n'est pas aussi simple lorsque les hypernotions contiennent des métanotions.

Plusieurs cas se présentent :

- la métanotion a une occurrence dans le premier membre. On donne à toutes ses occurrences dans le second membre la valeur qu'elle a dans le premier membre.

Exemple : abX : cdXe

sera traduit : a(z) = si z = bα alors c(dα e) où $\alpha \in L_X$

- la métanotion n'a pas d'occurrence dans le premier membre, nous dirons qu'elle est libre. C'est le cas de la règle 1 page 1.19 .

Soit Y une telle métanotion, le schéma qui la contient permet d'engendrer plusieurs règles en remplaçant la métanotion par les mots du langage L_Y.

Si le langage L_X est infini, on transforme la W-grammaire de façon à supprimer l'occurrence libre.

Exemple :

Soient N = {X,Y} , M = {u,v,d,e,a,b,c,d,...}

les métarègles X :: u Y v | d
Y :: v X u

et le schéma m_1 : α, ab X c, μ

où m_1 ne contient pas d'occurrences de X et α,μ pas d'occurrences libres de métanotions. On ajoute deux éléments à M : f_X et f_Y et on remplace le schéma m_1 : α, abXc, μ par les schémas suivants :

m_1 : f_Xd
f_XX : α , abXc, μ | f_Y u X v
f_YX : f_X v X u

C) Résultats du modèle obtenus pendant le filtrage.
A l'axiome correspond un appel de fonction ayant pour valeur un modèle. Si le problème de la reconnaissance est indécidable le processeur de filtrage bouclera et s'arrêtera à la suite d'un débordement de pile.

Si la reconnaissance est possible le modèle acceptera le langage engendré sous certaines conditions dues au fonctionnement du processeur de filtrage.

Des conditions analogues ont déjà été données dans l'étude de la reconnaissance des langages algébriques.

En mode Fullscan si les modèles sont récursifs à gauche, le processeur de filtrage boucle, mais comme nous l'avons exposé chap. 1.3.2 , on peut accepter que parmi plusieurs règles de même premier membre l'une d'elles soit récursive à gauche. Il faudra alors écrire le modèle de sorte qu'elle soit la dernière alternative. Si la chaîne analysée appartient au langage elle sera acceptée par le modèle sinon le filtrage bouclera et s'arrêtera sur un débordement de pile.

En mode Quickscan la W-grammaire ne doit pas avoir de règles vides et les fonctions à valeur des modèles doivent accepter des chaînes de longueur au moins égales à un. Toutefois il faut remarquer que l'emploi de fonctions à valeur des modèles limite l'efficacité du Quickscan à empêcher que le processeur de filtrage ne boucle indéfiniment. En effet, si une fonction a pour valeur un modèle qui accepte une chaîne dont la longueur dépend de l'argument, le processeur considérera que le modèle accepte un seul caractère. C'est pour cette raison que le programme p. 1.20 boucle en mode Quickscan lorsque le sujet n'appartient pas au langage.

Pour construire un analyseur pour une W-grammaire on définit des modèles uniquement au moyen de fonctions, mais on peut avoir des syntaxes qui utilisent à la fois des règles et des schémas. L'analyseur contiendra des modèles définis par des affectations et des fonctions à valeur de modèles. Ce mélange est aisé, essayons-le sur un exemple.

1.10.4.3. Utilisation de fonctions à valeur des modèles dans la reconnaissance de langages de programmation.

On a défini un langage de programmation en utilisant à la fois des règles et des schémas de règles. On se donne également la métagrammaire associée aux schémas.

Les règles et les schémas de règles sont notées avec le symbole ::= et les métarègles avec ::

métarègle: M :: entier | booléen

règles :

1. Instruction ::= affectation de mode M | autre
2. Affectation de mode M ::= repère de mode M := expression de mode M
3. Expression de mode M ::= constante de mode M | repère de mode M

4. Constante de mode entier :: = suite de chiffres
5. Constante de mode booléen :: = Notation booléenne
6. Notation booléenne :: ='0' B | '1' B
7. Repère de mode entier :: = (I| J| K| L| M| N) suite éventuellement vide de caractères alphabétiques.
8. Repère de mode booléen :: = B suite de caractères alphabétiques éventuellement vide.

La métagrammaire engendrant un langage fini on peut éclater le schéma de règle 1 en :

Instruction :: = affectation de mode entier | affectation de mode booléen | autre.

L'accepteur s'en déduit immédiatement.

```
ALPHANU     = '0123456789ABCDEFGHIJKLMNØPQRSTUVWXYZ'

      M     = 'ENTIER' | 'BØØLEEN'

SUITCHIFFRE =  SPAN('0123456789')

NØTABØØL    = "'0'B"  | "'1'B"

INST        =  AFFMØDE('ENTIER') |  AFFMØDE('BOOLEEN')  | AUTRE

AFFMØDE(X)  = si X ∈ M alors repère de modèle (X) ':='
                 EXPMØDE(X) sinon FAIL.

EXPMODE(X)  = si X  ∈ M alors REPMØDE(X) |  CØNSTMØDE(X) sinon FAIL

CØNSTMØDE(X)= si X = 'ENTIER' alors SUITCHIFFRE
              si X = 'BOOLEEN' alors NØTABØØL

REPMØDE(X)  = si X = 'ENTIER' alors  ANY('IJKLMN')  '' |
               SPAN(ALPHANU)
              si X = 'BØØLEEN' alors 'B'  SPAN(ALPHANU)
```

Le modèle INST reconnaît ce langage. Le mélange des règles et des schémas ne complique pas la construction de l'accepteur.

Mais la solution de l'éclatement du schéma 1 n'est plus possible lorsque la métagrammaire engendre un langage infini.

On utilise alors la technique de traduction des schémas dans les seconds membres desquels se trouve une occurrence libre d'une métanotion.

Si les fonctions ayant pour valeur des modèles sont précieuses pour construire des accepteurs pour des W-grammaires, il n'est pas toujours très simple de les utiliser et ceci est lié à la complexité des W-grammaires. Il semble que ces outils soient mieux adaptés pour traiter les grammaires affixes. C'est ce que nous allons étudier.

1.11. Reconnaissance des langages engendrés par les grammaires affixes.

Les grammaires affixes ont été introduites par Koster [19]. Koster a démontré qu'elles sont aussi puissantes que les W-grammaires et il a indiqué une méthode pour décrire la syntaxe d'Algol68 avec des grammaires affixes. De plus ces grammaires ont été conçues de manière à rendre facile l'analyse syntaxique.

Les grammaires affixes sont des doubles grammaires qui généralisent les C-grammaires. Le principe de généralisation est différent des W-grammaires.

Dans les W-grammaires on a un mécanisme de génération de non-terminaux et de construction de règles. A partir de ces règles on engendre le langage par dérivation à partir de l'axiome comme dans les C-grammaires. Dans les grammaires affixes les règles sont données, les non-terminaux sont en nombre fini.

Les règles contiennent, en plus des symboles non-terminaux, et terminaux, des symboles prédicats, auxquels sont associées des fonctions à valeur dans $\{\Lambda,\omega\}$ où Λ désigne la chaîne vide et ω le symbole "interdit", et des paramètres attachés aux non-terminaux et aux prédicats, appelés des affixes.

Soit X un symbole non-terminal et $y_1, y_2, \ldots. y_n$ les affixes associés. X et ses affixes forment une expression affixe notée $X + y_1 + y_2 + \ldots. + y_n$. X est appelé la tête de l'expression affixe et y_i la ième position affixe. Les affixes sont définis par une C-grammaire. Pour éviter les confusions on appellera règles affixes, les règles de la C-grammaire des affixes et règles de production les règles de la grammaire affixe.

Dans les dérivations on réécrira les non-terminaux comme on le faisait dans les C-grammaires et on effectuera en plus des substitutions sur des affixes ; les symboles prédicats produiront la valeur de la fonction associée, évaluée avec les affixes du symbole prédicat comme arguments. Si la fonction associée produit ω les dérivations produiront un mot qui n'appartient pas au langage.

Dans la définition d'une grammaire affixe on distingue les affixes qui sont des données des fonctions prédicats et des affixes qui sont des résultats. Les premiers sont appelés affixes hérités et les seconds affixes dérivés ou synthétisés.

1.11.1 Exemple : Comment engendrer les $a^n b^n c^n$ avec une grammaire affixe.

Une solution consiste à se donner d'abord une grammaire qui engendre les $\{a^n b^n c^p \mid n \geqslant o \quad p \geqslant o\}$, soit, par exemple, la grammaire suivante :

Z,X,Y sont les symboles non-terminaux et Z l'axiome. a,b,c les symboles terminaux et les règles :

1. Z:: = XY
2. X:: = aXb
3. X:: = Λ
4. Y:: = cY
5. Y:: = Λ

Puis on insère des affixes et des symboles prédicats pour imposer qu'au cours des réécritures on applique la règle 2 autant de fois que la règle 4.

On obtient alors une grammaire affixe AG1 suivante :

Grammaire des affixes :

- 1 est l'unique symbole terminal.
- D,R les symboles affixes non-terminaux.
- Les règles sont telles que de chaque non-terminal dérive 1* .

D:: = 1 D | Λ
R:: = D

Grammaire affixe :

- a,b,c les symboles terminaux.
- X,Y,Z les symboles non-terminaux et Z l'axiome.
- DECØMPTE, IDENT les symboles prédicats.
- règles de production :

1'. Z : X + R, Y + R
2'.X+D : a, DECØMPTE + D + R, X + R, b
3'.X+D : IDENT + D + Λ
4'.Y+D : c,DECØMPTE + D + R, Y + R
5'.Y+D : IDENT + D + Λ

Les virgules servent à séparer les expressions affixes. Elles ne font pas partie du langage engendré.

- Les fonctions associées aux prédicats sont :

$F_{DECOMPTE}(\alpha,\beta)$ = si $\alpha \neq \beta$ et $\alpha = \beta$ 1 alors Λ sinon ω
$F_{IDENT}(\alpha,\beta)$ = si $\alpha = \beta$ alors Λ sinon ω

Le langage engendré.

On obtient le langage engendré en effectuant des dérivations à partir de l'axiome comme dans les C-grammaires. Dans les dérivations on travaille sur des suites d'expressions affixes.

On appelle expression affixe concrête une expression dont tous les affixes appartiennent aux langages correspondants engendrés par la grammaire des affixes.

Dans les dérivations on ne réécrit que des expressions affixes concrêtes. L'axiome, ici Z, est une expression affixe concrête.

On réécrit une expression affixe dont la tête est un non-terminal en utilisant une règle de production dont le premier membre commence par ce non-terminal ; on remplace dans le second membre de cette règle les affixes ayant leur correspondant dans le premier membre par les valeurs correspondantes des affixes de l'expression affixe.

Les affixes non-terminaux se trouvant au second membre d'une règle et qui n'ont pas leur correspondant dans le premier membre sont appelés des affixes libres.

Dans les réécritures on remplacera un affixe non-terminal libre par un mot quelconque du langage qui en dérive dans la grammaire des affixes.

Ainsi, dans l'exemple qui nous intéresse, on réécrit l'axiome Z en remplaçant chaque occurrence de R dans le second membre de la règle 1'. par un certain 1^n. Z se réécrit, alors = X + 1^n, Y + 1^n.

Soit à réécrire X + 1^n. Il existe deux règles 2' et 3' dont le premier membre commence par X.

- Si n est différent de 0 alors 1^n est différent du mot vide et IDENT + 1^n produit ω (cf. F_{IDENT}) ; la règle 3' produisant ω on applique la règle 2'.

Dans cette règle R est un affixe libre, mais en fait le choix de la valeur 1^p à donner à R est limité, car pour $p \neq n - 1$
DECØMPTE + 1^n + 1^p produit ω (cf $F_{DECOMPTE}$), X + 1^n se réécrit β = a, DECØMPTE + 1^n + 1^{n-1}, X + 1^{n-1}, b etβ se réécrit a,Λ, X + 1^{n-1}, b

- Si n est égal à 0 alors 1^n est vide et quel que soit p, DECØMPTE +Λ + 1^p produit ω et IDENT + Λ + Λ produit Λ . Par suite X + 1^n engendre $a^n b^n$ (après suppression des virgules qui séparaient les expressions affixes). De la même façon Y + 1^n engendre c^n. Le langage engendré par la grammaire affixe est donc$\{a^n b^n c^n \mid n > 0\}$.

Type des affixes.

Dans la règle 1', R est une donnée, c'est un affixe hérité, tandis que R dans DECØMPTE + D + R (règle 2') est considéré comme résultat, c'est un affixe dérivé.

Une définition plus formelle de la syntaxe rend compte du type des affixes ι, pour les affixes hérités, δ pour les affixes dérivés.

On définit pour cela le contrôle S, qui est un ensemble de quintuplets, un par non-terminal et par symbole prédicat.

$$S = \left\{ \begin{array}{l} <Z,\ 0,\ \emptyset,\ \emptyset\ \rightarrow> \\ <X,\ 1,(\iota),\ (1^*),\ \rightarrow> \\ <Y,\ 1,(\iota),\ (1^*),\ \rightarrow> \\ <\text{DECØMPTE},\ 2,\ (\iota,\delta),\ (1^*,\ 1^*),\ F_{\text{DECØMPTE}}> \\ <\text{IDENT},\ 2,\ (\iota,\iota),\ (1^*,\ 1^*),\ F_{\text{IDENT}}> \end{array} \right\}$$

Le premier élément d'un quituplet est le symbole,
le 2ème élément : le nombre d'affixes associés au symbole,
le 3ème élément : le type des affixes associés,
le 4ème élément : le domaine des affixes associés,
le 5ème élément : la fonction associée (dans le cas d'un symbole prédicat).

1.11.2 Reconnaissance syntaxique :

1.11.2.1. Difficultés rencontrées pour construire l'analyseur.

On se propose de construire un modèle qui reconnaisse les mots du langage engendré par une grammaire affixe. On utilisera des fonctions ayant pour valeur des modèles et les affixes en seront les paramètres. Toutefois certaines règles ne peuvent pas être traduites de façon immédiate. Examinons-les les unes après les autres.

Règle 1'. : l'affixe non-terminal libre et hérité R est une cause d'indéterminisme.

Règle 2'. : Dans l'expression affixe DECOMPTE + D + R, R est un affixe non-terminal libre, mais il s'agit d'un affixe dérivé : on pourra modifier la définition de $F_{DECOMPTE}$ qui impose une relation entre D et R en une fonction qui calcule R en fonction de D. Soit,

$F_{DECOMPTE}(x,y)$ = si $x \neq \Lambda$ alors si $x = zl$ alors $y = z$ sinon ω .

Dans l'expression X + R, R est un affixe non-terminal, libre et hérité, c'est apparemment le même cas que dans la règle 1' ; toutefois, dans cette règle, il existe une occurrence dérivée de R située à gauche.Le filtrage parcourant la règle de gauche à droite R aura une valeur lorsque X + R sera pris en compte.

1.11.2.2. Une grammaire équivalente mieux adaptée à l'analyse.

Les difficultés à déduire l'accepteur proviennent de ce que la grammaire affixe a été conçue pour la génération. On peut la transformer en une grammaire affixe AG2 équivalente, mieux adaptée à l'analyse. AG2 est définie de la façon suivante.

Grammaire des affixes :

Symbole terminal affixe : l
Symboles non-terminaux affixes : R, R1.
Les règles sont telles que de chaque non-terminal dérive l^* .

Grammaire affixe :

Symboles terminaux : a,b,c
Symboles non-terminaux : X,Y,Z l'axiome est Z
Symboles prédicats : CPT, IDENT, INIT

les fonctions associées aux prédicats

$F_{CPT}(\alpha, \beta)$ = si $\alpha, \beta \in l^*$ et $\alpha = \beta l$ alors Λ sinon ω .

$F_{IDENT}(\alpha, \beta)$ = si $\alpha = \beta$ alors Λ sinon ω.

$F_{INIT}(\alpha, \beta)$ = si $\alpha = \beta$ alors Λ sinon ω.

Règles :

1) Z = X + R, Y + R1, IDENT + R + R1
2) X + R = a, X + R1, CPT + R + R1, b | INIT + Λ + R
3) Y + R = c, Y + R1, CPT + R + R1 | INIT + Λ + R

Contrôle :

$$S = \left\{ \begin{array}{l} <Z,\ 0,\ \emptyset,\ \emptyset,\ ->\\ <X,\ 1,(\delta),\ (1^*),\ ->\\ <Y,\ 1,(\delta),\ (1^*),\ ->\\ <CPT,\ 2,\ (\delta,\iota),\ (1^*,1^*)\ F_{CPT}>\\ <IDENT,2,(\iota,\delta),\ (1^*,1^*)\ F_{IDENT}> \end{array} \right\}$$

1.11.2.3. Comparaison entre AG1 et AG2.

Dans la règle 1' de AG1, l'affixe R permet d'imposer la contrainte suivante : le nombre de réécritures de X doit être égal au nombre de réécritures de Y.

On obtient la même contrainte en associant à X (resp. Y) deux affixes non-terminaux dérivés R (resp R1) et en ajoutant à la règle un prédicat qui produit ω si les affixes terminaux correspondant à R et R1 sont différents.

On a également remplacé le prédicat DECØMPTE dont la fonction associée effectue un décompte par le prédicat CPT dont la fonction associée effectue un comptage.

Enfin toutes les règles sont telles que, ou bien les seconds membres ne contiennent pas d'affixes libres hérités, ou bien toute occurrence d'un affixe libre hérité dans le second membre d'une règle est précédée d'une occurrence dérivée.

Il reste à prouver que, si on ne s'intéresse qu'au langage engendré, on puisse toujours effectuer la transformation qui fait passer de AG1 à AG2.

1.11.2.4. Programmation de l'accepteur.

Une première idée consiste à traduire les règles de production par des fonctions à valeur de modèles. Mais en Snobol 4 il n'est pas permis de faire des effets annexes sur les arguments des fonctions et on ne peut donc pas traduire les affixes dérivés par des paramètres de fonction. Si cela était possible l'accepteur s'en déduirait immédiatement.

Pour le réaliser on utilisera des variables globales et cette modification n'est pas toujours aussi simple à réaliser que dans notre exemple où deux variables globales suffisent.

1.11.2.5. Programme

```
      MT = ARBNO('L')                                                    1
      DEFINE('CPTX(U)') ; DEFINE('CPTY(U)')  :(FIN.DEF)                  2
CPTX  T1 POS(0) MT RPOS(0)  :F(FRETURN)                                  4
      T1 = T1 'L'   :(RETURN)                                            5
CPTY  T2 POS(0) MT RPOS(0)  :F(FRETURN)                                  6
      T2 = T2 'L'   :(RETURN)                                            7
FIN.DEF                                                                  8
*
      X = 'A' *CPTX() *X 'B' | NULL                                      9
      Y = 'C' *CPTY() *Y  | NULL                                         10
      Z = X Y *IDENT(T1,T2)                                              11
*
      &FULLSCAN = 1                                                      12
BOUCLE   CH = TRIM(INPUT)   :F(END)                                      13
      CH POS(0) Z RPOS(0)   :S(E1)F(E2)                                  14
E1    OUTPUT = CH ' OUI '   :(BOUCLE)                                    15
E2    OUTPUT = CH ' NON '   :(BOUCLE)                                    16
END                                                                      17
```

RESULTATS :

```
NO ERRORS DETECTED DURING COMPILATION

AABBCC OUI
ABC OUI
AABBACBB NON
BABCC NON
AAAABBBCCC NON
AAABBBBCCC NON
AAABBBCCCC NON

        756 MS. COMPILATION TIME
       1008 MS. EXECUTION TIME
         73 STATEMENTS EXECUTED,        6 FAILED
          0 ARITHMETIC OPERATIONS PERFORMED
         29 PATTERN MATCHES PERFORMED
          0 REGENERATIONS OF DYNAMIC STORAGE
      13.81 MS. AVERAGE PER STATEMENT EXECUTED
```

1.11.2.6. Conclusion

A partir des grammaires affixes, Koster définit un langage de description de compilateurs : CDL. D'autre part, META 5 [23] est fondé sur le même principe. On peut obtenir une implémentation rapide d'un compilateur de CDL en modifiant le compilateur Snobol4, pour qu'il soit permis de faire des effets annexes sur les arguments.

Mais sans faire cette modification on dispose avec Snobol 4 d'un langage qu'on peut utiliser pour décrire des compilateurs.

C'est ce que nous avons fait, en partie, dans l'écriture du compilateur d'AGILES.

2. METHODES DE PROGRAMMATION

0. INTRODUCTION

Jusqu'ici l'étude a porté sur des applications très particulières : reconnaître les mots de certains langages. Les programmes étaient essentiellement constitués de modèles, le reste contenant une instruction de filtrage et éventuellement une boucle pour reconnaître une suite de mots.

Mais Snobol peut être utilisé dans des applications plus générales. On peut y traiter toutes sortes de problèmes et même des problèmes pour lesquels il n'est pas fait a priori, comme, par exemple, le calcul du produit de matrices.

Nous nous intéressons plutôt aux applications qui demandent des calculs sur les chaines : après la reconnaissance de leur appartenance à certains langages et leur analyse, on veut effectuer sur elles des traitements.

Un programme sera, en général, formé de deux parties : une partie dans laquelle sont élaborés les modèles et l'autre dans laquelle on les utilise (instruction de filtrage) et dans laquelle on effectue les traitements autres que la reconnaissance.

On remarquera que dans la plupart des cas les affections de modèles sont effectuées une seule fois de sorte que les modèles ainsi élaborés sont constants et peuvent être considérés comme des déclarations.

1. CONCEPTION D'UN PROGRAMME SNOBOL 4, DEFINITION DES MODELES QU'IL UTILISE.

Nous allons traiter diverses applications de Snobol en mettant en évidence une méthode de programmation.

Une première méthode suivie consistera à partir des résultats et à définir les chaines résultats au moyen de fonctions. Cependant, cette méthode conduit, dans certains cas à des lourdeurs et il est parfois plus agréable de partir des données, d'en déduire des modèles et d'organiser le reste du programme à partir de ces modèles, ce qui nous amènera à exposer une autre méthode.

2. ELABORATION D'UN PROGRAMME SNOBOL 4 A PARTIR DES RESULTATS.

Dans certaines applications de Snobol 4 on est amené à utiliser toute la puissance des modèles et du filtrage ; dans d'autres, en revanche, ces outils permettent seulement d'apporter une solution élégante pour tester le contenu des chaines et pour en extraire des parties.

La structure des programmes obtenus dans ce dernier cas n'est pas fondamentalement différente de la structure de programmes écrits dans des langages usuels. Il est alors tentant de choisir une méthode de programmation employée avec les langages usuels.

Une méthode sûre consiste à partir des résultats, dont on donne une définition en fonction de résultats intermédiaires et/ou des données.

On définit ensuite les résultats intermédiaires jusqu'à ce que les définitions ne contiennent plus que des données. On peut aussi utiliser des définitions récursives. Les définitions sont aisées quand il s'agit de valeurs arithmétiques car on utilise les opérations arithmétiques. Sur les chaines, on dispose de même de la concaténation. Mais le plus souvent, la concaténation ne suffit pas et on doit extraire des sous-chaines des chaines arguments, suivant des critères variés. Ce sera grâce aux modèles et aux affectations dans les modèles que nous pourrons accèder à ces sous-chaines.

2.1. Exemple 1 : Polynôme dérivé d'un polynôme en X à coefficients entiers.

Il s'agit d'effectuer la dérivation formelle d'un polynôme donné sous forme d'une chaine de caractères. Le coefficient d'un monôme est une constante entière signée, si sa valeur absolue est 1, il peut être réduit au signe, il n'est pas séparé du symbole X par un caractère. L'exposant, séparé de X par deux astérisques, est une constante non signée. Si l'exposant ne figure pas les astérisques n'apparaissent pas. On note Λ la chaine vide.

2.1.1. Analyse :

On peut considérer un polynôme non réduit à un monôme comme un monôme signé suivi d'un polynôme. On évite de faire du premier monôme un cas particulier en ajoutant le signe "+" si son coefficient est positif et si le signe "+" ne figure pas explicitement.

On définit l'opérateur D de dérivation d'un polynôme dont le premier monôme est signe par récurrence .

D(P) = si P est un monôme alors D_1 (P)

sinon si P = M P' où M est un monôme signé et P' un polynôme alors D_1 (M)

D (P').

D1 (M) = si M est un coefficient alors Λ

sinon si M = C X où C est un coefficient alors UN (C)

Sinon si M = C X** E où E est l'exposant alors
si E = 1 alors UN(C)
sinon UN(C) * E X** (E-1).

En fait la dernière expression est un peu plus compliquée : si UN(C) E est positif le résultat ne sera pas signé il faudra alors rétablir le signe +.

UN(X) = si X ='+' ou '-' alors X '1'

Remarque 1. Si dans la décomposition de P, M est défini comme un monôme on le reconnaîtra grâce à un modèle de monôme. La solution est inefficace car on effectue deux fois la reconnaissance.

Mais pour effectuer le découpage on n'a pas besoin d'une syntaxe aussi précise : il suffit de remarquer que M peut être défini comme le plus petit facteur gauche non vide de P suivi d'un des caractères + ou - et de définir P' comme le reste de la chaîne. Une telle optimisation n'est pas possible dans tous les exemples.

2.1.2. Construction des modèles :

De l'analyse on peut déduire immédiatement les modèles. Le découpage monôme polynôme sera réalisé dans un filtrage par le modèle MØNØPØLY construit d'après la remarque précédente :

MØNØPØLY = (('+' | '-') BREAK ('+-')) . M REM . PPRIME MØNØPØLY
accepte une chaîne composée de deux sous-chaînes la première commence par un signe et est limitée à droite par un des signes + ou -.

La seconde est formée des caractères restants.

Ces deux chaînes sont respectivement affectées à M et àPPRIME, en cas de succès de filtrage.

Dans la définition de D1 on a besoin des modèles CØEFFICIENT et EXPØSANT.

CØEFFICIENT = ('+' | '-') SPAN ('1234567890').

EXPØSANT = SPAN ('1234567890').

2.1.3. Programme : voir page 2.5 et 2.6.

Sinon si M = C X** E où E est l'exposant alors

si E = 1 alors UN(C)

sinon UN(C) * E X** (E-1).

En fait la dernière expression est un peu plus compliquée : si UN(C) E est positif le résultat ne sera pas signé il faudra alors rétablir le signe +.

UN(X) = si X ='+' ou '-' alors X '1'

Remarque 1. Si dans la décomposition de P, M est défini comme un monôme on le reconnaîtra grâce à un modèle de monôme. La solution est inefficace car on effectue deux fois la reconnaissance.

Mais pour effectuer le découpage on n'a pas besoin d'une syntaxe aussi précise : il suffit de remarquer que M peut être défini comme le plus petit facteur gauche non vide de P suivi d'un des caractères + ou - et de définir P' comme le reste de la chaîne. Une telle optimisation n'est pas possible dans tous les exemples.

2.1.2. Construction des modèles :

De l'analyse on peut déduire immédiatement les modèles. Le découpage monôme polynôme sera réalisé dans un filtrage par le modèle MØNØPØLY construit d'après la remarque précédente :

MØNØPØLY = (('+' | '-') BREAK ('+-')) . M REM . PPRIME MØNØPØLY
accepte une chaîne composée de deux sous-chaînes la première commence par un signe et est limitée à droite par un des signes + ou -.

La seconde est formée des caractères restants.

Ces deux chaînes sont respectivement affectées à M et à PPRIME, en cas de succès de filtrage.

Dans la définition de D1 on a besoin des modèles CØEFFICIENT et EXPØSANT.

CØEFFICIENT = ('+' | '-') SPAN ('1234567890').

EXPØSANT = SPAN ('1234567890').

2.1.3. Programme : voir page 2.5 et 2.6.

2.1.4. Remarques sur le programme.

a) Dans la définition de D1, le modèle CØEFFICIENT intervient dans trois filtrages successifs, ce qui crée une inefficacité. En admettant que les monômes complets sont plus fréquents que les autres on diminue l'inefficacité en ordonnant correctement les instructions de filtrage.

On pourrait également effectuer une sorte de mise en facteur :

X CØEFFICIENT . C REM . RESTE

RESTE ayant pour valeur la sous-chaine située à gauche du coefficient.

Si RESTE est vide alors D1 = Λ

Si RESTE = X alors D1 = UN(C)

Si RESTE = X * E alors UN(C) * E X **(E - 1)

b) Il existe d'autres manières de décomposer un polynôme : celle qui consiste à le découper en un monôme non signé suivi d'un signe suivi d'un polynôme complique la solution lorsqu'un des monômes est constant.

Si on considère que le polynôme dérivé est formé de la concaténation des dérivéesdes monômes qui le composent, on évite l'emploi de définitions récursives.

2.1.5. Analyse

DERPØL : I ; tant que $\gamma \neq \Lambda$ faire P

I : RES = Λ

P : si γ se décompose en δ s γ' où δ est un monôme, s le signe +, - ou la chaine vide, alors RES = D1 (δ); γ= s γ'sinon erreur.

D1 étant défini comme dans l'autre programme.

2.2. Exemple 2 : Réduction de sous-expressions trigonométriques dans des expressions arithmétiques. L'expression arithmétique est donnée sous forme d'une chaine de caractéres.

PROGRAMME :

```
   PROGRAMME DE CALCUL DERIVEE POLYNOME
    RANCHOR = 1                                                          1
          NU = '0123456789'                                              2
          S = '+' | '-' | NULL                                           3
          MONOPOLY = ( S LEN(1)     BREAK('+-')) . DELTA REM . GAMP       4
          MOST = S SPAN(NU)                                              5
          MOEFF = S ( SPAN(NU) | NULL)                                   6
    MSPX = MOEFF . C 'X'                                                 7
    MGENE = MOEFF . C 'X**' MOST . E                                     8
*
*   RESTE DU PROGRAMME
*
    DEFINE('D(X)') ; DEFINE('D1(X)') ; DEFINE('UN(X)')  :(FIN.DEF)       9
*
D       D = IDENT(X)  :S(RETURN)                                         12
        X  MONOPOLY   :F(DECP)                                           13
        D  =  D1(DELTA) D(GAMP)  :S(RETURN)F(FRETURN)                    14
DECP    D  =  D1(X)              :S(RETURN)F(FRETURN)                    15
*
D1      X  MOST  RPOS(0)  :S(RETURN)                                     16
        X  MSPX  RPOS(0)  :S(MON1)                                       17
        X  MGENE RPOS(0)   :F(FRETURN)                                   18
* ON RETABLIT  LE  COEFFICIENT  SI C'EST  UN
        C = UN(C)                                                        19
        EQ(E,1)             :F(PLN)                                      20
           D1 = C    :(RETURN)                                           21
PLN     CD = C * E  ; CD = GT(CD,0) '+' CD                               22
        D1 =  CD  'X**' (E - 1)  :(RETURN)                               24
*
*
MON1    C = UN(C)  ;  C = GT(C,0) '+' C  ;  D1 = C  :(RETURN)            25
*
*
UN      X  ('+' | NULL)   RPOS(0)    :S(PLUS1)                           28
        X   '-'  RPOS(0)             :S(MOINS1)                          29
        UN  =  X                     :(RETURN)                           30
PLUS1   X = 1    :(RETURN)                                               31
MOINS1  X = -1     :(RETURN)                                             32
*
FIN.DEF                                                                  33
*
BOU     CH = TRIM(INPUT) :F(END)                                         34
          OUTPUT = ' POLYNOME DONNE : '  CH                             35
        DCH = D(CH)      :F(ERREUR)                                      36
          OUTPUT = ' POLYNOME RESULTAT : ' DCH  ; DCH =    ; :(BOU)      37
ERREUR    OUTPUT  =  'ERREUR'  :(BOU)                                    40
END                                                                      41


 NO ERRORS DETECTED DURING COMPILATION
```

RESULTATS :

```
P0LYN0ME D0NNE : 81X+181X**2-14X**8-18X**3
P0LYN0ME RESULTAT : +81+362X**1-112X**7-54X**2
P0LYN0ME D0NNE : +13X**2-6X**1+3X**6+186X**1
P0LYN0ME RESULTAT : +26X**1-6+18X**5+186
P0LYN0ME D0NNE : +13X**2-6X**1+3X**6+186X**51
P0LYN0ME RESULTAT : +26X**1-6+18X**5+9486X**50


N0RMAL TERMINATI0N AT LEVEL  0


LAST STATEMENT EXECUTED WAS   34


SN0B0L4 STATISTICS SUMMARY-
          1584 MS. C0MPILATI0N TIME
          1152 MS. EXECUTI0N TIME
           189 STATEMENTS EXECUTED,      73 FAILE
            16 ARITHMETIC 0PERATI0NS PERF0RMED
            71 PATTERN MATCHES PERF0RMED
             0 REGENERATI0NS 0F DYNAMIC ST0RAGE
          6.10 MS. AVERAGE PER STATEMENT EXECUTED
```

On souhaite faire cette réduction en appliquant la formule $\sin(\alpha + \beta) = \cos\alpha \sin\beta + \cos\beta \sin\alpha$. Ainsi, dans la chaîne

sin(y) - a + x ** 2 + sin(b-c) cos(a) - z + cos(b-c) sin(a)

la réduction est

sin(y) - a + x ** 2 - z + sin(b-c+a)

Pour simplifier un peu le problème on admettra que l'argument ne commence jamais par un signe.

2.2.1. Analyse. Définition de la fonction Réduire (γ) γ étant une expression arithmétique.

Réduire (γ) = si γ s'écrit δ_1 $f_1(\alpha)$ f_2 (β) δ_2 $\begin{matrix} f_1 (\beta) f_2 (\alpha) \\ \text{ou} \\ f_2 (\alpha) f_1 (\beta) \end{matrix}$ δ_3

où δ_1, δ_2, δ_3 sont des sous-expressions,

f_1 est soit sin soit cos et

f_2 = si f_1 = sin alors cos.

si f_2 = cos alors sin.

α ,β sont des chaînes quelconques comprises entre des parenthèses.

alors δ_1 sin (α+ β) δ_2 δ_3

sinon γ

2.2.2. Programme voir page 2.8

2.3. Exemple 3. On recherche dans un texte les occurrences des mots d'un certain langage acceptés par le modèle M, on veut éditer chaque occurrence avec ses contextes droits et gauches, l'occurrence sera entourée de trois astérisques.

2.3.1. Analyse. On construit la fonction EDITER qui traite tout le texte et sort les éditions de toutes les occurrences.

EDITER (τ) = si τ se décompose en γ_1 μ γ_2, où γ_1 est une suite de mots

$\notin$M et $\mu \in$ M alors début δ = $\delta\gamma_1$; sortir δ *** μ *** γ_2 ;

$\delta = \delta\mu$; EDITER (γ_2) ; fin.

On remarque que l'appel récursif de EDITER (γ_2) se trouve à la fin de l'instruction composée et on peut donc remplacer la fonction récursive par une boucle.

PROGRAMME DE REDUCTION DE SOUS-EXPRESSIONS TRIGONOMETRIQUES

```
-LIST
 REDUCTION
* FONCTION  C(X) SI X=SIN ALORS COS SINON SIN
      DEFINE('C(X)')    :(FIN.C)                                              1
C     C = IDENT(X,'SIN') 'COS'   :S(RETURN)                                   2
      C = IDENT(X,'COS') 'SIN'   :S(RETURN)F(FRETURN)                         3
FIN.C                                                                         4
*
      DEFINE('REDUIRE(X)')     :(FIN.RED)                                     5
REDUIRE  X POS(0) ARB . D1 '+' ('SIN' | 'COS') $ F1                           6
+         '(' BREAK(')') $ A1 ')'                                             6
+         *C(F1) $ F2 '(' BREAK(')') $ A2 ')' ARB . D2                        6
+    '+' (*F1 '(' *A2 ')' *F2 '(' *A1 ')' | *F2 '(' *A1 ')' *F1 '(' *A2      6
+   ')' ) REM . D3   :F(E1)                                                    6
 REDUIRE = D1 '+SIN(' A1 '+' A2 ')' D2  REDUIRE(D3) :S(RETURN)F(FRETURN)      7
E1        REDUIRE = X      :(RETURN)                                          8
FIN.RED                                                                       9
          &ANCHOR = 1                                                         10
L         CH = TRIM(INPUT)          :F(END)                                   11
          OUTPUT = 'EXPRESSION INITIALE : ' CH                                12
          OUTPUT = 'EXPRESSION REDUITE : ' REDUIRE(CH)    :S(B)F(ER)          13
ER        OUTPUT = 'EXPRESSION ERRONNEE '                 :S(B)               14
END                                                                           15
```

NO ERRORS DETECTED DURING COMPILATION

RESULTATS :

```
EXPRESSION INITIALE : A+COS(X)SIN(Y)+B+COS(Y)SIN(X)-B
EXPRESSION REDUITE : A+SIN(X+Y)+B-B
EXPRESSION INITIALE : +SIN(X)COS(Y)+COS(Y)SIN(X)-COS(X)SIN(Y)
EXPRESSION REDUITE : +SIN(X)COS(Y)+COS(Y)SIN(X)-COS(X)SIN(Y)
EXPRESSION INITIALE : +SIN(X)COS(Y)+COS(X)SIN(Y)-COS(X)SIN(Y)
EXPRESSION REDUITE : +SIN(X+Y)-COS(X)SIN(Y)


NORMAL TERMINATION AT LEVEL  0
LAST STATEMENT EXECUTED WAS   11
SNOBOL4 STATISTICS SUMMARY-
            864 MS. COMPILATION TIME
           1368 MS. EXECUTION TIME
             29 STATEMENTS EXECUTED,        6 FAILED
              0 ARITHMETIC OPERATIONS PERFORMED
              5 PATTERN MATCHES PERFORMED
              1 REGENERATIONS OF DYNAMIC STORAGE
          47.17 MS. AVERAGE PER STATEMENT EXECUTED
```

EDITER : si τ se décompose en $\gamma_1 \mu \gamma_2$ début $\delta = \delta\gamma_1$; sortir δ *** μ *** γ_2 ; $\delta = \delta\mu$; $\tau = \gamma_2$; allera EDITER fin.

Ce genre de programme, que l'on peut classer dans la catégorie "programme de service pour les applications littéraires" est trés pénible à obtenir dans d'autres langages que Snobol4.

En réalité le problème est tout de même un peu plus compliqué car on désire souvent, en plus, arrêter les contextes des mots recherchés à quelques mots ou à la limite de signes de ponctuation (.,;). Nous allons également reprendre l'analyse pour répondre à des contraintes technologiques. Il est clair que si le texte à traiter est long on ne pourra pas le ranger totalement en mémoire ; à chaque instant on travaillera sur une sous-chaîne γ_2 de τ formée de huits mots sauf vers la fin du texte τ . On gardera également le contexte gauche CGAUCHE de τ réduit à 8 mots ou vide si les premiers caractères de γ_2 sont des séparateurs (.,;).

On construira une fonction ELEM() qui fournira les mots du texte un à un suivis de leurs séparateurs (.,;), cette fonction ne lira qu'une carte à la fois en mémoire ; si elle est appelée à la fin du texte source elle a pour valeur la chaîne vide.

- initialisation de γ_2 : lire 8 mots $\rightarrow \gamma_2$

  CGAUCHE =

- Bouc.

  si γ_2 se décompose en MOT δ où MOT est un mot suivi de séparateurs et δ le reste de γ_2

  alors début si mot = μsep où $\mu \in M$

  alors début sortir CGAUCHE *** μ *** f_2 (γ_2) fin.

  γ_2 = δ elem () ;

  CGAUCHE = CGAUCHE MØT ;

  Réduire (CGAUCHE) ; allera Bouc fin

  sinon si $\gamma_2 = \Lambda$ alors arrêt sinon erreur.

- f_2 (α) : si $\alpha = \beta$ sep γ où sep $\in$ {.,;} β ne contient pas de caractères de sep

  alors $f_2 = \beta$

  sinon $f_2 = \alpha$

- elem(x) : si CH = Λalors si CLE = EOF alors début
ELEM = Λ ; retour fin sinon lire ch

si CH = MOT RESTE où MOT est un mot suivi de séparateur

alors début ELEM = MOT ; CH = RESTE fin

sinon erreur.

2.3.2. Programme voir pages 2.11 et 2.12

2.4. Définition de la méthode.

Comme nous l'avons vu dans les exemples, on est amené à définir des fonctions éventuellement récursives sur des chaines en faisant un découpage des arguments, de la manière suivante :

A_1, A_2, ... An étant des langages, on définit

F (γ) = si $\gamma = \alpha_1\ \alpha_2 \ldots \alpha_n$ où pour i = 1 à n $\alpha_i \in A_i$ alors

T (U_1 (α_1), U_2 (α_2), ..., Un (α_n)).

T désigne un traitement dans lequel on effectue des calculs sur des chaines obtenues à partir de α_1, α_2, ... α_n au moyen de fonctions U_1, U_2,... U_n.

Soient M_{A_1}, M_{A_2}, ... M_{A_n} les modèles qui acceptent respectivement les mots des langages A_1, A_2, ... A_n. Le découpage est alors réalisé par l'instruction de filtrage suivante :

$$\gamma \quad M_{A_1} \cdot \alpha_1 \quad M_{A_2} \cdot \alpha_2 \quad \ldots\ M_{A_n} \cdot \alpha_n$$

Les fonctions U_1, U_2,..... U_n peuvent à nouveau être définies de la même façon.

Cette méthode est applicable dans tous les cas, mais peut conduire,comme nous allons le voir sur un exemple,à des inefficacités.

Exemple : Les articles d'un fichier sont formés d'une clé composée de trois chiffres suivis de deux lettres et d'une partie contenant vingt caractéres.

On veut imprimer les 20 derniers caractéres des articles dont la clé n'a pas cette structure ; de plus, on veut recopier dans un fichier

PROGRAMME D'EDITION DE CONTEXTE

```
*
*         PROGRAMME D'EDITION DE CONTEXTE
*
        CLE = 0 ; BPV = ' ,.;' ;  PVP =  '.,;'                          1
* LA FONCTION ELEM
        DEFINE('ELEM(X)')      :(FIN.ELEM)                              4
ELEM    ELEM =  ;  IDENT(CH) :F(SUITE1)                                 5
        EQ(CLE,1)              :S(RETURN)                               7
        CH = TRIM(INPUT) ' ' :F(POSCLE)S(SUITE1)                        8
POSCLE  CLE = 1                :(RETURN)                                9
SUITE1  CH ( BREAK(BPV) SPAN(BPV) ) . ELEM  =  :(RETURN)                10
FIN.ELEM                                                                11
*
*  MODELE  DES MOTS RECHERCHES
*
        RAC = 'AUCTORIT'                                                12
        DESI = 'AS' | 'ATIS' | 'ATEM' | 'ATE' | 'ATES'                  13
        M  =  RAC DESI                                                  14
INITIAL  I = I + 1  ; GAM2 = LE(I,8) GAM2 ELEM()   :S(INITIAL)          15
* DECOMPOSITION DE GAM2
        CGAUCHE =  ;  SANCHOR = 1                                       17
BOUCLE   IDENT(GAM2)          :S(END)                                   19
        GAM2 BREAK(BPV) . MOT  SPAN(BPV) . SEP  REM . DELTA  :F(ERREUR) 20
*
* LE PREMIER MOT DE GAM2 EST IL LE MOT RECHERCHE
        MOT M                  :F(RENVOI)                               21
*
*  LIMITATION  DE GAM2    CONTEXTE DROIT
*  CALCUL DU CONTEXTE DROIT
          SEP ARB ANY(PVP)     :F(INCAL)                                22
          CDROIT = SEP         :(PL)                                    23
INCAL     DELTA (BREAK(PVP) SPAN(PVP)) . CDROIT :S(PL)                  24
          CDROIT = DELTA                                                25
PL      OUTPUT = CGAUCHE ' **** ' MOT ' **** ' CDROIT                   26
*
RENVOI  SEP ARB ANY(PVP)    :S(CGNULL)F(CGHUIT)                         27
CGNULL  J = 1 ; CGAUCHE = SEP     :(MAJGAM2)                            28
CGHUIT  CGAUCHE =  CGAUCHE MOT SEP                                      30
          J = LT(J,8) J + 1    :S(MAJGAM2)                              31
          CGAUCHE  BREAK(BPV) SPAN(BPV)  =                              32
MAJGAM2 GAM2 = DELTA ELEM() :(BOUCLE)                                   33
ERREUR  OUTPUT = ' ERREUR'  :(END)                                      34
END                                                                     35


NO ERRORS DETECTED DURING COMPILATION
```

TEXTE ORIGINAL :

ULLA VEL REQUIRENDI VEL HABENDI PESSIMA PRESUMPTIO . OBSECRAMUS ET PER
AUCTORITATEM XII APOSTOLORUM, PRESERTIM SANETI JOHANNIS,DILECTISSIMI DOMINI, CUJ
US IN PREFATA MONASTERIO DENS HABETUR ET VENERATUR UT CONFIRMATUM EST INIBI AUCT
ORITATE AUCTORITATE BEATI NONI LEVIUS PAPE ET PER AMOREM OMNIUM SANCTORUM NECNO
N ET ORTESTAMUR PER NOSTRAM QUAM A DEO ACCEPIMUS POTESTATEM UT NEQUE A SE NEQUE
A QUOLIBET IRRITUM FIERI PERMITTANT ET AUCTORITATES NOSTRAE MAMUS CONFIRMAMUS

RESULTAT :

. OBSECRAMUS ET pER **** AUCTORITATEM **** XII ApOSTOLORUM,
ET VENERATUR UT CONFIRMATUM EST INIBI AUCT ORITATE **** AUCTORITATE **** BEATI NONI LEVIUS PAPE ET PER AMOREM
SE NEQUE A QUOLIBET IRRITUM FIERI PERMITTANT ET **** AUCTORITATES **** NOSTRAE MAMUS CONFIRMAMUS

NORMAL TERMINATION AT LEVEL 0
LAST STATEMENT EXECUTED WAS 19
SNOBOL4 STATISTICS SUMMARY-
1404 MS. COMPILATION TIME
5472 MS. EXECUTION TIME
821 STATEMENTS EXECUTED, 325 FAILED
31 ARITHMETIC OPERATIONS PERFORMED
330 PATTERN MATCHES PERFORMED
2 REGENERATIONS OF DYNAMIC STORAGE
6.67 MS. AVERAGE PER STATEMENT EXECUTED

les articles dont la partie numérique de la clé est supérieure à 500.

Selon la méthode décrite précédemment une première analyse de ce problème donnera :

BØUCLER : lire CARTE si fin fichier arrêt

si CARTE = CLE SUITE où CLE $\in L_{CLE}$ et SUITE $\in L_{SUITE}$

alors si CLE = NU LETTRES où NU $\in L_{NU}$ et LETTRES $\in L_{LETTRES}$

alors si NU >500 copier.

BØUCLER

On sait d'autre part que L_{CLE} est défini comme le produit de L_{NU} par $L_{LETTRES}$.

Les instructions de filtrage qui effectuent le découpage de CARTE et de CLE utilisent des modèles en commun et ces modèles s'appliquent aux mêmes chaînes.

Le processeur de filtrage effectuera donc deux fois les mêmes reconnaissances. Nous allons compléter la méthode pour éviter ces inefficacités.

2.5. Optimisation. Le principe consiste à faire du découpage à l'avance, ce qui donne dans l'exemple précédent :

BOUCLER : lire CARTE si fin fichier arrêt.

si CARTE = NU LETTRES suite où

NU $\in L_{NU}$, LETTRES $\in L_{Lettres}$, SUITE $\in L_{Suite}$

alors si NU > 500 alors copier

sinon imprimer "erreur" CARTE.

BØUCLER.

Dans cet exemple on économise même une instruction de filtrage, ce qui n'est pas, a priori, le but recherché. On veut essentiellement éviter les redondances dans les modèles qui sont utilisés pour filtrer la même chaîne.
Examinons plus en détail ce type d'optimisation. Il faut distinguer deux cas :

2.5.1. Les modèles redondants sont composés par concaténation.

$f(\gamma)$: si $\gamma = \alpha\beta$ où $\alpha \in L_1 L_2$, $\beta \in L_3$ alors $T(U(\alpha), V(\beta))$.

$U(\alpha)$: $W(\alpha)$; si $\alpha = \alpha' \alpha''$ où $\alpha' \in L_1$, $\alpha'' \in L_2$ alors

$T'(U'(\alpha'), U''(\alpha''))$.

On fait passer le découpage de α en α' et α'' dans la définition de f, ce qui donne :

$f(\gamma)$: si $\gamma = \gamma\beta$ où $\alpha = \alpha' \alpha''$, $\alpha' \in L_1$, $\alpha'' \in L_2$ et $\beta \in L_3$

alors $T'''(W(\alpha), V(\beta), T'(U'(\alpha'), U''(\alpha'')))$.

2.5.2. Les modèles redondants sont composés par alternative.

$f(\gamma)$: si $\gamma = \alpha\beta$ où $\alpha \in L_1$, $\beta \in L_2 \cup L_3$ alors

$T_1(U(\alpha), V(\beta))$

$V(\beta)$: si $\beta = \beta'$ où $\beta' \in L_2$ alors $T_2(W(\beta'))$.

si $\beta = \beta''$ où $\beta'' \in L_3$ alors $T_3(X(\beta''))$.

On ne peut plus opérer comme dans l'exemple précédent car on ne sait plus ce que le filtrage a reconnu des membres de l'alternative.

On a alors deux solutions possibles :

i) développer les modèles en utilisant la distributivité de la concaténation par rapport à l'alternative, ce qui donne :

$f(\gamma)$ = si $\gamma = \alpha\beta$ où $\alpha \in L_1$, $\beta \in L_2$ alors

$T'(U(\alpha), V(\beta), W(\beta))$.

si $\gamma = \alpha\beta$ où $\alpha \in L_1$, $\beta \in L_2$ alors

$T''(U(\alpha), V(\beta), X(\beta))$.

ii) faire noter par le processeur de filtrage l'alternative reconnue, par effet annexe, à l'aide d'une fonction non évaluée (voir page1.4)à valeur vide et utiliser la marque par la suite.

Pour optimiser on pourra alors opérer comme dans le cas 1 en effectuant le découpage à l'avance.

Ce qui donne :

f (γ) = si $\gamma = \alpha\beta$ où $\beta \in L_2$, resp $\beta \in L_3$ alors

faire T ' (U (α), V (β), U (β)) resp T " (U (α), V (β), X (β))

L'adverbe "respectivement" est traduit de la façon suivante :

F est une fonction affectant à la variable RESP son argument.

```
                DEFINE ( ' F (X) ')   :    (FIN. DEF)
F               RESP =  X             :    (RETURN)
FIN.DEF
                &ANCHØR = 1
                M = M L1  . α  (ML2 . β'  *F('E1') |
                ML3 . β"   *F ('E2')):  S (§ RESP)
E1
                .
                .
                .
                τ'
                .
                .
                .       : (SUITE)
E2              .
                .
                .
                τ"
                .
                .
                .       : (SUITE)
```

On a réalisé un branchement indirect § RESP, RESP valant, suivant le cas, E1 ou E2.

2.5.3. Remarques sur les optimisations.

Les optimisations des exemples précédents visaient à éviter l'utilisation de modèles redondants devant servir à des filtrages sur les mêmes chaines.

Si ceci a entraîné la suppression de filtrage, il ne faudrait pas croire que rechercher à diminuer le nombre des filtrages augmente l'efficacité du programme.

En effet, on peut être amené au contraire à augmenter le nombre des filtrages devant opérer sur une chaîne, mais avec des modèles différents, le premier modèle effectuant un découpage grossier avec une grammaire qui décrit la chaine de manière superficielle et des modèles plus précis permettant d'analyser de manière plus fine les sous-chaînes ainsi définies.

Cette façon d'opérer permet de diminuer les inconvénients de la méthode de fonctionnement du processeur de filtrage (retour en arrière).

Nous avons utilisé cette technique dans l'exemple 1, ainsi que dans la réalisation du compilateur du langage AGILES où un filtrage reconnaît et décompose les instructions, tout en notant les composants qui seront traités ultérieurement.

2.6. Problèmes faisant intervenir des fonctions de façon récursive.

On désire sortir toutes les sous expressions bien parenthésées dans un texte dans l'ordre des longueurs croissantes. On va stocker toutes les expressions bien parenthésées et on les trera.

On définit une fonction qui extrait les sous-expressions bien parenthésées d'une chaîne τ et qui les empile.

$f(\tau)$ = si $\tau = \alpha\ (\ \delta)\ \beta$ où α ne contient ni), ni (
et où $\delta \in L_{expr}$, $\beta \in L_{expr}$
alors empiler β ; $f(\delta)$; $f(\gamma)$
sinon erreur.

L_{expr} est défini par des règles pour lesquelles nous donnons les modèles correspondants :

```
SANS_PAR = BREAK ('()') (')' FAIL | NULL)

ENTRPAR  = '(' *LEXPR ')'

LEXPR    = SANS_PAR ( *ENTRPAR     *EXPR  | NULL)
```

La fonction ainsi définie de manière récursive va entraîner un certain nombre de filtrages sur des chaînes communes avec des modèles identiques (LEXPR, ENTPAR).

On effectuera la même optimisation que dans les problèmes ne faisant pas intervenir des fonctions récursives en prévoyant le découpage à l'avance, au moyen d'affectations dans les modèles. Comme les modèles sont récursifs il faudra que les affectations le soient également. On utilisera une variable de type pile, une affectation à cette variable consistera à mettre la valeur sur une pile associée.

Le type pile n'existant pas en Snobol il sera réalisé par une fonction dont le résultat est un nom (au sens de Snobol, c'est-à-dire un élément qui peut recevoir une valeur dans une affectation).

Ce nom sera un élément de tableau T< I> où I représente le sommet de la pile.

Exemple Définition de P

```
     T = ARRAY (50)
     DEFINE ('P(X)')    : (FIN. P)
P    I = L T(50)  I + 1   :  S(PLP)
     ØUTPUT = 'DEBØRDEMENT PILE' : (FRETURN)
PLP  P =  .  T< I>  : (NRETURN)
FIN. P
```

L'affectation dans le modèle s'écrira *ENTRPAR . *P() ; avec cette définition de P le programme s'écrit :

```
SANSPAR  =  BREAK (')(') (')' FAIL | NULL)
ENTRPAR  =  '(' *LEXPR ')'
LEXPR    =  SANSPAR ( *ENTRPAR  .  *P() *EXPR | '')
&ANCHØR  =  1
CH       = INPUT
CH  LEXPR  RPØS(0)   :  F(END)

TRI    DE    T
       .
       .
       .
```

Remarque : Dans cet exemple, le traitement ne dépend pas du choix des alternatives. S'il en dépendait, il faudrait noter, en plus, l'alternative choisie, ce qui pourrait être réalisé par effet annexe d'une fonction non évaluée dans un modèle (comme la fonction de l'exemple). Nous appliquerons ces techniques à l'analyse syntaxique, dans le chapitre suivant.

3. ELABORATION DE PROGRAMME SNOBOL 4 A PARTIR DE MODELES.

Dans la définition de la méthode précédente nous avions remarqué que, si on ne donnait pas un rôle privilégié aux modèles, si on les considérait seulement comme un outil d'accés à des chaines de caractéres, alors la structure des programmes ressemblait beaucoup à la structure des mêmes programmes écrits dans un langage usuel.

Cependant, avec les possibilités d'affectation, d'appel de fonction, d'insertion de prédicats... les modèles constituent un véritable langage de programmation dont l'environnement est formé des objets du langage Snobol 4.

La structure de contrôle de ce langage le rend trés différent des langages usuels.

Nous allons voir qu'il est agréable à utiliser dans les problèmes dont la solution peut être déduite de la description des données.

Un programme dans ce langage sera un modèle dans lequel sont incorporés des appels de fonction.

3.1. Comment programmer dans ce langage ?

Nous avons à traiter un problème dont la solution se déduit d'une description adéquate des données. Dans sa généralité le problème n'est pas simple, en effet, on a à choisir la bonne description.

Nous nous restreindrons au cas où l'énoncé donne la description adéquate, ce qui est souvent le cas, par exemple pour les problèmes de reconnaissance, d'analyse syntaxique,...

On construit alors le modèle correspondant à la description syntaxique, puis on insére des affectations et des points d'action. Enfin on définit les fonctions réalisant les points d'action.

Nous allons appliquer la méthode à un exemple.

3.1.1. Problème.

Une chaine CH est formée d'un nombre arbitraire de sous-chaines SCH. Chaque sous-chaine est formée également d'un nombre arbitraire de composants CP_{ij}.

Tous les composants sont formés d'une lettre, la même pour tous les composants d'une même chaine SCH_i, et d'un nombre qui suit cette lettre. On veut éditer, pour chaque SCH_i, la lettre qui la caractérise ainsi que la somme des nombres qui la suivent dans les CP_{ij}.

3.1.2. Exemple de donnée :

b5b12b4a326a437d8e13e1226e1e14

SCH_1 CP_{22}

Résultat à obtenir :

b 21

a 763

d 8

e 1264

3.1.3 Solution : Dans un premier temps on construit un modèle de toute la chaine qui caractérise les différents CPij et les différents SCHi.

On initialise L à ''.

NBRE = SPAN ('0123456789') 1

DEB = NØTANY (L) $ L NBRE

CØNT = L NBRE CØNT | NULL 2

MSCH = DEB CØNT

MCH = ARBNØ (MSCH 3) R PØS(0)

Pour simplifier l'exemple, nous ne nous préoccuperons pas de savoir si L est bien une lettre, la donnée est supposée exacte.

Il aurait fallu disposer d'un opérateur de filtrage

(ET) : ANY ('AB.....Z') & NØTANY (L).

Ce modèle rend compte du changement de lettre, c'est-à-dire du changement des SCHi.

Ensuite on détermine les actions et leur emplacement :

en 1. On ajoute $VAL *IA(VAL) ; on affecte à VAL la valeur filtrée par NBRE et on initialise CUMUL à VAL.

en 2. On ajoute $VAL *AD(VAL) ; la fonction AD augmentant CUMUL de la valeur VAL.

en 3. On ajoute *RESUL( ) ; qui imprimera le résultat.

Ce qui donne le programme suivant :

```
          DEFINE ('IA(X)')
          DEFINE ('AD(X)')
          DEFINE ('RESUL(X)')          :    (FIN.DEF)

IA        CUMUL = X                    :    (RETURN)
AD        CUMUL = CUMUL + VAL          :    (RETURN)
RESUL     ØUTPUT = L '***' CUMUL       :    (RETURN)
FIN.DEF
          A = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
          L = ' *'
          NBRE = SPAN('0123456789')
          DEB  =   ANY(A)   $ L NBRE    $   VAL   *IA (VAL)
          CØNT = *L NBRE $ VAL *AD(VAL) *CØNT  | NULL
          MSCH = DEB CØNT
          MSCH = ARBNØ (MSCH  *RESUL( )   ) RPØS(0)

DEBUT DE PROGRAMME

          INPUT    MCH             :  S(END)
          ØUTPUT = ' ERREUR '

END
```

3.2. Conclusion

Cet exemple est moins artificiel qu'il en a l'air. On le rencontre trés souvent en gestion. Par exemple, on dispose d'un fichier de bons de commande dans lesquels figurent : le numéro du représentant, le montant des produits vendus. Le fichier est tiré suivant les numéros de représentant et on veut éditer un état du chiffre d'affaire par représentant. Les langages de programmation comme CØBØL et PL/I permettent de faire des descriptions (variables structurées) pour accéder aux différents champs d'un enregistrement. Ensuite il faut encore décrire les ruptures, les acquisitions etc. Snobol permet de décrire tout le fichier et non pas seulement ses enregistrements, donc les ruptures qui peuvent être traitées de la même manière que les champs à l'intérieur d'un enregistrement.

En analyse de gestion on décrit généralement la structure globale d'un fichier avec ses hiérarchies ; elle pourrait être exprimée par des modèles, qu'il faudrait éventuellement enrichir de nouvelles fonctions comme le modèle de fin d'enregistrement, etc...

Les problèmes d'accéder aux enregistrements, de placer les lectures, seraient alors résolus.

L'outil de description que constituent les modèles est un intérêt de la méthode. Celle-ci a certainement des limites : un énoncé peut ne pas contenir la description des données qui convient à la solution du problème.

Si la description est insuffisante, le problème est insoluble ; mais elle risque de contenir plus de renseignements qu'il n'en faut, elle conduit alors à des inefficacités. Par exemple, on peut donner dans un énoncé la syntaxe complète des expressions arithmétiques et n'avoir à compter que le nombre de parenthèses ; il serait alors trés lourd d'utiliser la grammaire donnée dans l'énoncé.

3. ANALYSE SYNTAXIQUE

0. INTRODUCTION

Les modèles permettent de faire très facilement de la reconnaissance syntaxique. Etant donnés une grammaire G et un certain mot M on sait construire un modèle qui reconnaît si M appartient au langage engendré par G.

On désire aussi savoir comment, par quelles régles. Autrement dit en terme de filtrage, nous voulons connaître, dans les modèles, les alternatives choisies par l'opérateur de filtrage.

Une solution consiste à numéroter les alternatives choisies ; le résultat de l'analyse est alors la suite des numéros des règles qui ont engendré M, plus précisément le résultat de l'analyse est une ramification [11] et la suite obtenue est sa représentation postfixée. Nous voulons construire en Snobol un programme qui calcule cette suite.

1. UNE PREMIERE SOLUTION pour réaliser cette numérotation consiste à insérer des actions d'écriture dans les règles sous forme de fonctions à évaluation retardée. Lorsque la fonction est évaluée par le processeur de filtrage, son argument, c'est-à-dire le n° de la règle, est édité.

Malheureusement la solution n'est pas si simple. En effet, il faut tenir compte du fonctionnement du processus de filtrage ; en cas d'échec il revient en arrière pour essayer une autre alternative, mais les résultats sortis subsistent.

On évite ces résultats parasites en sortant les numéros de règles sur une pile où on place des marques et on effectue des remises à jour de la pile lors des retours en arrière. Pour gérer toutes ces opérations on introduit de nouvelles fonctions à évaluation retardée, à valeur vide. Le modèle initial, qui correspond à la grammaire, s'alourdit et devient moins lisible.

Cette solution présente toutefois un intérêt : le programmeur contrôle par des fonctions et une pile le fonctionnement de l'analyseur. En demandant une TRACE de la pile on peut suivre le fonctionnement de l'analyseur par les états de la pile, ce qui est utile pour faire des mises au point et ce qui constitue un outil expérimental d'étude d'analyse syntaxique. On pourra ainsi remarquer, par exemple, l'influence de la grammaire sur l'analyseur....

1.1. Exemple : Voir programme page 3.4

2. LA SECONDE SOLUTION est un peu moins naturelle, mais plus élégante. Elle utilise une technique employée chapitre 2 p 17 pour réaliser une variable de type pile.

Elle consiste à insérer, dans le modèle correspondant à la grammaire, à l'endroit qu'il faut numéroter, une affection conditionnelle à une variable générée par une fonction à évaluation retardée.

L'affectation étant conditionnelle la fonction n'est évaluée que lorsque le filtrage est réussi et on n'aura donc pas de résultats parasites.

2.1. Exemple 1 : Soient les règles suivantes :

Z = X | Y et le modèle correspondant de l'acceptation :

MZ = MX | MY.

La fonction MARQUER ayant comme résultat principal un nom et comme résultat secondaire l'impression de l'argument, dans la transformation de l'accepteur en analyseur le modèle s'écrira :

MZ = MX . *MARQUER(1) | MY . *MARQUER(2)

2.2. Exemple 2.

Analyse des expressions arithmétiques. Quand on analyse des expressions arithmétiques on s'intéresse, en ce qui concerne les identificateurs et les constantes, à obtenir les chaînes de caractères qui les composent plutôt que la suite des numéros des règles qui les ont engendrées.

De plus, le langage qui les engendre est régulier et leur grammaire se décrit très bien avec des fonctions modèles standard . On distinguera ces opérandes dans la chaîne résultat en les séparant des numéros de règles par des parenthèses.

Comme on prévoit d'utiliser le résultat de l'analyse dans un compilateur on ne numérotera que les règles pour lesquelles il faudra générer un texte objet. Le programme se déduit immédiatement de la grammaire des expressions arithmétiques.

Un tel analyseur est programmé dans le compilateur d'AGILES, p. 4.11. Le programme suivant est un exemple simplifié.

2.3. Programme : voir page 3.3

ANALYSEUR SYNTAXIQUE
D'EXPRESSIONS ARITHMETIQUES ENGENDREES PAR LA
GRAMMAIRE DONT LES REGLES SONT :

E ::= E + T | T

T ::= T * F | F

F ::= (E) | a | b | c

```
      DEFINE('N(X)')        :(FIN.NOTE)                                    1
N     PILE = X PILE                                                        2
      N  = 'BIDON'          :(NRETURN)                                     3
FIN.NOTE                                                                   4
*
*  LA  GRAMMAIRE
*
      E  = *E  '+' *T . *N(1)  | *T . *N(2)                                5
      T = *T  '*' *F . *N(3)  | *F . *N(4)                                 6
    F = '(' *E ')' . *N(5) | 'A' . *N(7) | 'B' . *N(8) | 'C' . *N(9)       7
*
      &ANCHOR = 1                                                          8
BOUCLE   CH  =  TRIM(INPUT)       :F(END)                                  9
         PILE =                                                            10
         CH  E  RPOS(0)   :S(RESULT)                                       11
          OUTPUT = ' DONNEE ERRONNEE : ' CH       :(BOUCLE)                12
RESULT   OUTPUT = 'DONNEE : ' CH ' RESULTAT : ' PILE          :(BOUCLE)    13
END                                                                        14
```

RESULTATS :

```
DONNEE : A RESULTAT : 247
DONNEE : A+(B*C)+B RESULTAT : 14814523948247
DONNEE : B*(B+C)*A RESULTAT : 2373514924848
DONNEE : C*A RESULTAT : 23749
DONNEE : B+C RESULTAT : 149248
DONNEE : (B+C)*(A+B) RESULTAT : 235148247451492 48
 DONNEE ERRONNEE : ((B*(B+C))+A))


SNOBOL4 STATISTICS SUMMARY-
            756 MS. COMPILATION TIME
           9252 MS. EXECUTION TIME
            150 STATEMENTS EXECUTED,        2 FAILED
              0 ARITHMETIC OPERATIONS PERFORMED
              7 PATTERN MATCHES PERFORMED
              0 REGENERATIONS OF DYNAMIC STORAGE
          61.68 MS. AVERAGE PER STATEMENT EXECUTED
```

ANALYSEUR D'EXPRESSIONS ARITHMETIQUES UTILISANT

UNE PILE GEREE PAR LE PROGRAMMEUR.

LA GRAMMAIRE EST DONNEE SOUS FORME GENERALISEE

SANS RECURSIVITE A GAUCHE.

```
          &FULLSCAN = 1    ;  &ANCHOR = 1                                    1
          &TRACE = 500                                                       3
          TRACE('PILE','VALUE')                                              4
          S1 = ( '+' | '-' ) $ VS1                                           5
          S2 = ( '*' | '/' ) $ VS2                                           6
T = *P('C') *F *P(4) *AROBN0(S2 *F *Q(5))                                    7
F = *P('A') 'A' *P(7) | *V('A')  '(' *E ')' *P(8) | *0('A') FAIL              8
E = *P('E')  *T *P(1) *AROBN0(S1 *T *Q(2))                                   9
          DEFINE('P(X)')          :(FIN.P)                                   10
P         PILE = X PILE ; P =           :(RETURN)                            11
FIN.P                                                                        13
          DEFINE('V(X)')          :(FIN.V)                                   14
V         PILE BREAK(X) =  ; V =        :(RETURN)                            15
FIN.V                                                                        17
          DEFINE('0(X)')          :(FIN.0)                                   18
0         PILE BREAK(X) X =                                                  19
          0 =           :(RETURN)                                            20
FIN.0                                                                        21
          DEFINE('Q(X)')          :(FIN.Q)                                   22
Q         EQ(X,2)                 :F(Q5)                                     23
          X = IDENT(VS1,'-') X * 1        :(Q1)                              24
Q5        X = IDENT(VS2,'/') X + 1                                           25
Q1        PILE = X PILE ; Q =     :(RETURN)                                  26
FIN.Q                                                                        28
B0UC                                                                         29
          CH = TRIM(INPUT)        :F(END)                                    30
          OUTPUT = ' DONNEE : ' CH                                           31
          CH E RPOS(0)            :S(SS)F(FF)                                32
SS        RES = PILE  ; PILE =                                               35
          &ANCHOR = 0                                                        36
BS        RES ANY('ABCDE') =      :S(BS)                                     37
          &ANCHOR = 1
* SNES N0B EL SNAD TATLUSER UD ESIMER
* REMISE DU RESULTAT DANS LE B0N SENS                                        38
          RES1 =                                                             39
BREV      RES ANY('0123456789') . LC =   :F(IMPR)                            40
          RES1 = LC RES1          :(BREV)                                    41
IMPR      OUTPUT = ' RESULTAT : ' RES1      :(B0UC)                          42
FF        OUTPUT = ' ERREUR DE SYNTAXE'               :(B0UC)                43
END
```

RESULTATS :

```
DONNEE : (A-A)/A
   STATEMENT 11: PILE = 'E',TIME = 108
   STATEMENT 11: PILE = 'CE',TIME = 108
   STATEMENT 11: PILE = 'ACE',TIME = 108
   STATEMENT 15: PILE = 'ACE',TIME = 108
   STATEMENT 11: PILE = 'EACE',TIME = 108
   STATEMENT 11: PILE = 'CEACE',TIME = 108
   STATEMENT 11: PILE = 'ACEACE',TIME = 108
   STATEMENT 11: PILE = '7ACEACE',TIME = 108
   STATEMENT 11: PILE = '47ACEACE',TIME = 108
   STATEMENT 11: PILE = '147ACEACE',TIME = 108
   STATEMENT 11: PILE = 'C147ACEACE',TIME = 108
   STATEMENT 11: PILE = 'AC147ACEACE',TIME = 108
   STATEMENT 11: PILE = '7AC147ACEACE',TIME = 108
   STATEMENT 11: PILE = '47AC147ACEACE',TIME = 108
   STATEMENT 26: PILE = '347AC147ACEACE',TIME = 108
   STATEMENT 11: PILE = '8347AC147ACEACE',TIME = 108
   STATEMENT 11: PILE = '48347AC147ACEACE',TIME = 108
   STATEMENT 11: PILE = '148347AC147ACEACE',TIME = 180
   STATEMENT 11: PILE = 'A148347AC147ACEACE',TIME = 216
   STATEMENT 11: PILE = '7A148347AC147ACEACE',TIME = 252
   STATEMENT 26: PILE = '67A148347AC147ACEACE',TIME = 288
   STATEMENT 11: PILE = '167A148347AC147ACEACE',TIME = 288
   STATEMENT 34: PILE = '',TIME = 324
RESULTAT : 741743841761
DONNEE : A/A*A
   STATEMENT 11: PILE = 'E',TIME = 684
   STATEMENT 11: PILE = 'CE',TIME = 684
   STATEMENT 11: PILE = 'ACE',TIME = 720
   STATEMENT 11: PILE = '7ACE',TIME = 756
   STATEMENT 11: PILE = '47ACE',TIME = 792
   STATEMENT 11: PILE = '147ACE',TIME = 828
   STATEMENT 11: PILE = 'A147ACE',TIME = 864
   STATEMENT 11: PILE = '7A147ACE',TIME = 900
   STATEMENT 26: PILE = '67A147ACE',TIME = 936
   STATEMENT 11: PILE = '167A147ACE',TIME = 972
   STATEMENT 11: PILE = 'A167A147ACE',TIME = 1008
   STATEMENT 11: PILE = '7A167A147ACE',TIME = 1044
   STATEMENT 26: PILE = '57A167A147ACE',TIME = 1044
   STATEMENT 11: PILE = '157A167A147ACE',TIME = 1080
   STATEMENT 34: PILE = '',TIME = 1116
RESULTAT : 741761751
```

```
DONNEE : A+A/A
   STATEMENT 11: PILE = 'E',TIME = 1368
   STATEMENT 11: PILE = 'CE',TIME = 1368
   STATEMENT 11: PILE = 'ACE',TIME = 1404
   STATEMENT 11: PILE = '7ACE',TIME = 1440
   STATEMENT 11: PILE = '47ACE',TIME = 1476
   STATEMENT 11: PILE = '147ACE',TIME = 1512
   STATEMENT 11: PILE = 'C147ACE',TIME = 1548
   STATEMENT 11: PILE = 'AC147ACE',TIME = 1584
   STATEMENT 11: PILE = '7AC147ACE',TIME = 1584
   STATEMENT 11: PILE = '47AC147ACE',TIME = 1620
   STATEMENT 26: PILE = '247AC147ACE',TIME = 1656
   STATEMENT 11: PILE = 'A247AC147ACE',TIME = 1692
   STATEMENT 11: PILE = '7A247AC147ACE',TIME = 1728
   STATEMENT 26: PILE = '67A247AC147ACE',TIME = 1764
   STATEMENT 26: PILE = '267A247AC147ACE',TIME = 1800
   STATEMENT 34: PILE = '',TIME = 1800
RESULTAT : 741742762
DONNEE : A*(A+A)*A
   STATEMENT 11: PILE = 'E',TIME = 2052
   STATEMENT 11: PILE = 'CE',TIME = 2088
   STATEMENT 11: PILE = 'ACE',TIME = 2124
   STATEMENT 11: PILE = '7ACE',TIME = 2160
   STATEMENT 11: PILE = '47ACE',TIME = 2160
   STATEMENT 11: PILE = '147ACE',TIME = 2196
   STATEMENT 11: PILE = 'A147ACE',TIME = 2268
   STATEMENT 15: PILE = 'A147ACE',TIME = 2268
   STATEMENT 11: PILE = 'EA147ACE',TIME = 2304
   STATEMENT 11: PILE = 'CEA147ACE',TIME = 2340
   STATEMENT 11: PILE = 'ACEA147ACE',TIME = 2376
   STATEMENT 11: PILE = '7ACEA147ACE',TIME = 2376
   STATEMENT 11: PILE = '47ACEA147ACE',TIME = 2412
   STATEMENT 11: PILE = '147ACEA147ACE',TIME = 2448
   STATEMENT 11: PILE = 'C147ACEA147ACE',TIME = 2520
   STATEMENT 11: PILE = 'AC147ACEA147ACE',TIME = 2520
   STATEMENT 11: PILE = '7AC147ACEA147ACE',TIME = 2556
   STATEMENT 11: PILE = '47AC147ACEA147ACE',TIME = 2592
   STATEMENT 26: PILE = '247AC147ACEA147ACE',TIME = 2628
   STATEMENT 11: PILE = '8247AC147ACEA147ACE',TIME = 2664
   STATEMENT 26: PILE = '58247AC147ACEA147ACE',TIME = 2700
   STATEMENT 11: PILE = '158247AC147ACEA147ACE',TIME = 2736
   STATEMENT 11: PILE = 'A158247AC147ACEA147ACE',TIME = 2772
   STATEMENT 11: PILE = '7A158247AC147ACEA147ACE',TIME = 2808
   STATEMENT 26: PILE = '57A158247AC147ACEA147ACE',TIME = 2844
   STATEMENT 11: PILE = '157A158247AC147ACEA147ACE',TIME = 2880
   STATEMENT 34: PILE = '',TIME = 2916
RESULTAT : 7417417428851751


     SNOBOL4 STATISTICS SUMMARY-
                1476 MS. COMPILATION TIME
                3276 MS. EXECUTION TIME
                 352 STATEMENTS EXECUTED,       21 FAILED
                   4 ARITHMETIC OPERATIONS PERFORMED
                  89 PATTERN MATCHES PERFORMED
                   0 REGENERATIONS OF DYNAMIC STORAGE
                9.31 MS. AVERAGE PER STATEMENT EXECUTED
```

ANALYSEUR D'EXPRESSIONS ARITHMETIQUES UTILISANT

UNE PILE GEREE PAR LE PROGRAMMEUR.

LA GRAMMAIRE EST NON RECURSIVE A GAUCHE.

```
          &ANCHOR = 1                                                          1
          &FULLSCAN = 1                                                        2
          &TRACE = 500                                                         3
          TRACE('PILE','VALUE')                                                4
F = *P('F') 'A' *P(7) | *V('F') '(' *E ')' *P(8) | *O('F') FAIL                 5
B = *P('B') | *V('B') '*' *F *P(5) *B | *V('B') '/' *F *P(6)                   6
+ *B | *O('B') FAIL                                                            6
T = *P('T') *F *P(4) *B | *O('T') FAIL                                         7
A = *P('A') | *V('A') '+' *T *P(2) *A | *V('A') '-' *T *P(3)                   8
+ *A | *O('A') FAIL                                                            8
E = *P('E') *T *P(1) *A | *O('E') FAIL                                         9
          DEFINE('P(X)')          :(FIN.P)                                     10
P         PILE = X PILE           :(RETURN)                                    11
FIN.P                                                                          12
          DEFINE('V(X)')          :(FIN.V)                                     13
V         PILE BREAK(X) =         :(RETURN)                                    14
FIN.V                                                                          15
          DEFINE('O(X)')          :(FIN.O)                                     16
O         PILE BREAK(X) X =       :(RETURN)                                    17
FIN.O                                                                          18
          &TRACE = 100                                                         19
                                                                               20
BOUC      TRACE('PILE','VALUE')                                                21
          CH = TRIM(INPUT)        :F(END)                                      22
          OUTPUT = ' DONNEE : ' CH                                             23
          CH E RPOS(0)            :S(SS)F(FF)                                  24
SS        RES = PILE ; PILE =                                                  25
          &ANCHOR = 0                                                          27
BS        RES ANY('ABFTE') =      :S(BS)                                       28
          &ANCHOR = 1                                                          29
* SNES NOB EL SNAD TATLUSER UD ESIMER
* REMISE DU RESULTAT DANS LE BON SENS
          RES1 =                                                               30
BREV      RES ANY('0123456789') . LC =  :F(IMPR)                               31
          RES1 = LC RES1          :(BREV)                                      32
IMPR      OUTPUT = ' RESULTAT : ' RES1     :(BOUC)                             33
FF        OUTPUT = ' ERREUR DE SYNTAXE'            :(BOUC)                     34
END                                                                            35
```

RESULTATS : à l'aide de la TRACE on peut voir l'évolution de la pile d'analyse avec ses marques (lettres) pour le retour arrière et les numéros de règles. La pile est dans le sens opposé au sens habituel pour en faciliter la gestion. Le résultat est remis dans le bon sens (post fixé)

```
DONNEE : (A-A)/A
    STATEMENT 11: PILE = 'E',TIME = 216
    STATEMENT 11: PILE = 'TE',TIME = 252
    STATEMENT 11: PILE = 'FTE',TIME = 288
    STATEMENT 14: PILE = 'FTE',TIME = 288
    STATEMENT 11: PILE = 'EFTE',TIME = 324
    STATEMENT 11: PILE = 'TEFTE',TIME = 360
    STATEMENT 11: PILE = 'FTEFTE',TIME = 396
    STATEMENT 11: PILE = '7FTEFTE',TIME = 396
    STATEMENT 11: PILE = '47FTEFTE',TIME = 432
    STATEMENT 11: PILE = 'B47FTEFTE',TIME = 468
    STATEMENT 11: PILE = '1B47FTEFTE',TIME = 468
    STATEMENT 11: PILE = 'A1B47FTEFTE',TIME = 504
    STATEMENT 14: PILE = 'A1B47FTEFTE',TIME = 540
    STATEMENT 14: PILE = 'A1B47FTEFTE',TIME = 576
    STATEMENT 11: PILE = 'TA1B47FTEFTE',TIME = 576
    STATEMENT 11: PILE = 'FTA1B47FTEFTE',TIME = 612
    STATEMENT 11: PILE = '7FTA1B47FTEFTE',TIME = 648
    STATEMENT 11: PILE = '47FTA1B47FTEFTE',TIME = 684
    STATEMENT 11: PILE = 'B47FTA1B47FTEFTE',TIME = 684
    STATEMENT 11: PILE = '3B47FTA1B47FTEFTE',TIME = 720
    STATEMENT 11: PILE = 'A3B47FTA1B47FTEFTE',TIME = 756
    STATEMENT 11: PILE = '8A3B47FTA1B47FTEFTE',TIME = 792
    STATEMENT 11: PILE = '48A3B47FTA1B47FTEFTE',TIME = 792
    STATEMENT 11: PILE = 'B48A3B47FTA1B47FTEFTE',TIME = 828
    STATEMENT 11: PILE = '1B48A3B47FTA1B47FTEFTE',TIME = 864
    STATEMENT 11: PILE = 'A1B48A3B47FTA1B47FTEFTE',TIME = 900
    STATEMENT 14: PILE = 'A1B48A3B47FTA1B47FTEFTE',TIME = 936
    STATEMENT 14: PILE = 'A1B48A3B47FTA1B47FTEFTE',TIME = 936
    STATEMENT 17: PILE = '1B48A3B47FTA1B47FTEFTE',TIME = 972
    STATEMENT 14: PILE = 'B48A3B47FTA1B47FTEFTE',TIME = 1008
    STATEMENT 14: PILE = 'B48A3B47FTA1B47FTEFTE',TIME = 1044
    STATEMENT 11: PILE = 'FB48A3B47FTA1B47FTEFTE',TIME = 1044
    STATEMENT 11: PILE = '7FB48A3B47FTA1B47FTEFTE',TIME = 1080
    STATEMENT 11: PILE = '67FB48A3B47FTA1B47FTEFTE',TIME = 1116
    STATEMENT 11: PILE = 'B67FB48A3B47FTA1B47FTEFTE',TIME = 1152
    STATEMENT 11: PILE = '1B67FB48A3B47FTA1B47FTEFTE',TIME = 1152
    STATEMENT 11: PILE = 'A1B67FB48A3B47FTA1B47FTEFTE',TIME = 1188
    STATEMENT 26: PILE = '',TIME = 1224
RESULTAT : 74174384761
```

```
DONNEE : A/A*A
   STATEMENT 11: PILE = 'E',TIME = 1620
   STATEMENT 11: PILE = 'TE',TIME = 1620
   STATEMENT 11: PILE = 'FTE',TIME = 1656
   STATEMENT 11: PILE = '7FTE',TIME = 1692
   STATEMENT 11: PILE = '47FTE',TIME = 1692
   STATEMENT 11: PILE = 'B47FTE',TIME = 1728
   STATEMENT 11: PILE = '1B47FTE',TIME = 1764
   STATEMENT 11: PILE = 'A1B47FTE',TIME = 1800
   STATEMENT 14: PILE = 'A1B47FTE',TIME = 1800
   STATEMENT 14: PILE = 'A1B47FTE',TIME = 1836
   STATEMENT 17: PILE = '1B47FTE',TIME = 1872
   STATEMENT 14: PILE = 'B47FTE',TIME = 1908
   STATEMENT 14: PILE = 'B47FTE',TIME = 1908
   STATEMENT 11: PILE = 'FB47FTE',TIME = 1944
   STATEMENT 11: PILE = '7FB47FTE',TIME = 1980
   STATEMENT 11: PILE = '67FB47FTE',TIME = 2016
   STATEMENT 11: PILE = 'B67FB47FTE',TIME = 2016
   STATEMENT 11: PILE = '1B67FB47FTE',TIME = 2052
   STATEMENT 11: PILE = 'A1B67FB47FTE',TIME = 2088
   STATEMENT 14: PILE = 'A1B67FB47FTE',TIME = 2088
   STATEMENT 14: PILE = 'A1B67FB47FTE',TIME = 2124
   STATEMENT 17: PILE = '1B67FB47FTE',TIME = 2160
   STATEMENT 14: PILE = 'B67FB47FTE',TIME = 2196
   STATEMENT 11: PILE = 'FB67FB47FTE',TIME = 2196
   STATEMENT 11: PILE = '7FB67FB47FTE',TIME = 2232
   STATEMENT 11: PILE = '57FB67FB47FTE',TIME = 2268
   STATEMENT 11: PILE = 'B57FB67FB47FTE',TIME = 2304
   STATEMENT 11: PILE = '1B57FB67FB47FTE',TIME = 2340
   STATEMENT 11: PILE = 'A1B57FB67FB47FTE',TIME = 2340
   STATEMENT 26: PILE = '',TIME = 2376
RESULTAT : 7476751
DONNEE : A+A/A
   STATEMENT 11: PILE = 'E',TIME = 2592
   STATEMENT 11: PILE = 'TE',TIME = 2628
   STATEMENT 11: PILE = 'FTE',TIME = 2664
   STATEMENT 11: PILE = '7FTE',TIME = 2664
   STATEMENT 11: PILE = '47FTE',TIME = 2700
   STATEMENT 11: PILE = 'B47FTE',TIME = 2736
   STATEMENT 11: PILE = '1B47FTE',TIME = 2772
   STATEMENT 11: PILE = 'A1B47FTE',TIME = 2772
   STATEMENT 14: PILE = 'A1B47FTE',TIME = 2808
   STATEMENT 11: PILE = 'TA1B47FTE',TIME = 2844
   STATEMENT 11: PILE = 'FTA1B47FTE',TIME = 2880
   STATEMENT 11: PILE = '7FTA1B47FTE',TIME = 2880
   STATEMENT 11: PILE = '47FTA1B47FTE',TIME = 2916
   STATEMENT 11: PILE = 'B47FTA1B47FTE',TIME = 2952
   STATEMENT 11: PILE = '2B47FTA1B47FTE',TIME = 2988
   STATEMENT 11: PILE = 'A2B47FTA1B47FTE',TIME = 2988
   STATEMENT 14: PILE = 'A2B47FTA1B47FTE',TIME = 3024
   STATEMENT 14: PILE = 'A2B47FTA1B47FTE',TIME = 3060
   STATEMENT 17: PILE = '2B47FTA1B47FTE',TIME = 3060
   STATEMENT 14: PILE = 'B47FTA1B47FTE',TIME = 3096
   STATEMENT 14: PILE = 'B47FTA1B47FTE',TIME = 3132
   STATEMENT 11: PILE = 'FB47FTA1B47FTE',TIME = 3168
   STATEMENT 11: PILE = '7FB47FTA1B47FTE',TIME = 3204
   STATEMENT 11: PILE = '67FB47FTA1B47FTE',TIME = 3204
   STATEMENT 11: PILE = 'B67FB47FTA1B47FTE',TIME = 3240
   STATEMENT 11: PILE = '2B67FB47FTA1B47FTE',TIME = 3276
   STATEMENT 11: PILE = 'A2B67FB47FTA1B47FTE',TIME = 3276
   STATEMENT 26: PILE = '',TIME = 3312
RESULTAT : 74174762
```

```
DONNEE : A*(A+A)*A
   STATEMENT 11: PILE = 'E',TIME = 3600
   STATEMENT 11: PILE = 'TE',TIME = 3600
   STATEMENT 11: PILE = 'FTE',TIME = 3636
   STATEMENT 11: PILE = '7FTE',TIME = 3672
RESULTAT : 7474174285751



NORMAL TERMINATION AT LEVEL  0
LAST STATEMENT EXECUTED WAS   22
SNOBOL4 STATISTICS SUMMARY.
          1512 MS. COMPILATION TIME
          4536 MS. EXECUTION TIME
           333 STATEMENTS EXECUTED,       9 FAILED
             0 ARITHMETIC OPERATIONS PERFORMED
           138 PATTERN MATCHES PERFORMED
             0 REGENERATIONS OF DYNAMIC STORAGE
         13.62 MS. AVERAGE PER STATEMENT EXECUTED
```

4. COMPILATION EN SNOBOL

1. ETUDE DES PROBLEMES DE COMPILATION

1.1. Analyse : Dans les chapitres précédents nous avons étudié les langages reconnus par les modèles. Nous avons également vu comment transformer un accepteur en analyseur syntaxique en insérant dans les modèles des points d'action réalisés par des fonctions à évaluation retardée.

Les problèmes d'analyse des langages de programmation sont un peu plus compliqués que ceux des langages algébriques : d'autre part on a l'habitude, pour des raisons d'efficacité, de distinguer plusieurs types d'analyse, par exemple, reconnaissance des identificateurs, des constantes, découpage du texte en instructions...

1.1.1. Analyse lexicographique. Les constantes et les identificateurs sont engendrés par des langages réguliers. On peut évidemment les engendrer au moyen d'une C-grammaire et en déduire le modèle pour les analyser.

Mais en Snobol 4 on dispose de modèles standard mieux adaptés à l'analyse des langages réguliers : SPAN(CH) accepte la plus grande chaîne ne contenant que des caractères de la chaîne CH, BREAK(CH) accepte la plus grande chaîne contenant tous les caractères à l'exception de ceux de CH, ANY(CH) accepte un caractère de CH, NØTANY(CH) accepte un caractère à condition qu'il n'appartienne pas à CH.

Exemple : Si LETTRE est une chaîne formée des lettres de l'alphabet, NU une chaîne formée des chiffres de 0 à 9. Dans la plupart des langages de programmation les identificateurs peuvent être reconnus par le modèle :

ANY(LETTRE) SPAN(LETTRE NU)

Un entier peut être reconnu par : SPAN(NU)
Un nombre décimal par SPAN(NU) '.' SPAN(NU).

Dans la partie analyse lexicographique on traite également les problèmes de formats d'instructions, c'est le cas des langages comme FØRTRAN, CØBØL... Les modèles standards TAB, LEN, RTAB, PØS, RPØS permettent de traiter agréablement ce point.

Exemple : (SPAN(NU) | NULL). ETI SPAN (' ') | NULL
TAB(6) permet d'analyser la zone étiquette en FØRTRAN, il y aura échec si l'étiquette dépasse la colonne 6.

1.1.2. Analyse syntaxique.

Nous savons construire des analyseurs. Dans les langages de programmation s'ajoutent d'autres contraintes que celles des langages algébriques : vérifier si une variable utilisée est bien déclarée, vérifier le type d'une opérande, la compatibilité des types des opérandes pour un opérateur donné, la taille d'une constante... Ces vérifications seront réalisées en insérant des points d'action dans les règles.

Exemple : IDENT = (ANY(LETTRE) SPAN(LETTRE NU)) $
X *VERIDCL(X) où la fonction VERIDCL vérifie si l'identificateur X se trouve dans la table des symboles, imprime un message en cas d'erreur, retourne la valeur vide ou le modèle FAIL suivant l'importance de l'erreur.

Mais cette fonction peut également noter des renseignements sur le type de la variable et fournir des arguments à la procédure de génération.

L'analyseur ainsi conçu a un fonctionnement inefficace, surtout s'il doit travailler sur des textes longs et des modèles complexes. Il convient donc de lutter contre son inefficacité en réduisant la longueur des textes à traiter.

Pour cela on effectuera un découpage du programme ; en Fortran, par exemple, on découpera le programme en instructions car les instructions peuvent se compiler de façon indépendante les unes des autres ; en Cobol le programme est découpé en division, etc...

On pourra également effectuer l'analyse par niveaux, en utilisant des grammaires de plus en plus fines : reconnaître une instruction, noter ses composants, puis les analyser ensuite de façon indépendante.

Exemple : Reconnaissance de l'instruction IF de Cobol dans son format le plus complet :

'IF' ARB . EXPB 'THEN' ARB . ARG1 'ELSE' ARB . ARG2

Par la suite on analysera EXPB, ARG1, ARG2.

12. Génération .

Les problèmes de génération sont facilités avec un langage de traitement de chaînes de caractères. En Snobol 4 la possibilité de manipuler des chaînes de longueur arbitraire, la faible priorité de l'opérateur de concaténation dans les expressions facilitent encore plus la programmation des problèmes de génération.

On peut toutefois regretter l'absence de fonctions d'aide à la génération pour tabuler les résultats (alignement des codes instructions, des opérandes ...)

La génération peut se faire de manière indépendante de l'analyse ou bien en même temps.

Si on utilise le premier type d'analyseur étudié au chapitre "Analyse Syntaxique", celui pour lequel les résultats sortis en cours de filtrage ne sont pas définitifs, on peut soit générer un résultat intermédiaire sous forme d'une chaîne contenant les numéros de règles ou, ce qui est équivalent, les numéros des procédures de générations correspondantes, soit générer directement le texte objet.

Cette deuxième solution est toutefois plus inefficace ; en effet, il est plus long de générer le texte objet correspondant à une règle que son numéro et si cette règle a été choisie à tort, le temps perdu sera donc plus important.

Avec le second type d'analyseur, celui qui utilise la technique d'insertion d'affectation conditionnelle à une variable générée par une fonction, on peut évidemment sortir un résultat intermédiaire ou générer directement le texte objet. En fait, dans le second cas, la génération est réalisée par le processus de filtrage lorsque le filtrage est terminé.

Si on choisit la solution du résultat interne, la chaîne résultat sera "égrénée" par filtrage et remplacement (par la chaine vide) ; le caractère extrait sera utilisé comme étiquette vers la séquence de génération correspondante.

Dans tous les cas on a une raison supplémentaire de découper le texte source : ne pas garder des résultats trop longs en mémoire .

1.3. Les tables.

Un compilateur utilise un certain nombre de tables qui sont consultées très fréquemment ; il est donc nécessaire de les représenter de façon à faciliter les accès.

Les objets de type TABLE font partie de la version 3 du langage. Nous allons exposer un moyen de les représenter avec ce qui est disponible dans les versions antérieures. On utilisera la table des symboles du système Snobol 4. En effet, comme il est possible de créer dynamiquement des variables (grâce à l'opérateur unaire $), ainsi que du code objet (grâce à la fonction CØDE), il est nécessaire que le

système Snobol 4 garde à l'exécution la table des symboles.
De plus, certaines implémentations ne faisant pas de différences entre les chaînes de caractères des identificateurs et les autres, car celles-ci peuvent devenir des identificateurs de variables créées dynamiquement au moyen de l'opérateur $, ces dernières entrent également dans la table des symboles.

Soit à construire une table de constantes associant à chaque constante l'adresse d'un mot qui la contiendra. On va construire, à partir de la chaîne C, notation de la constante, une variable $C.

Lors d'une première rencontre de la constante, la variable ainsi créée a la valeur vide, on affecte, alors, à cette variable l'adresse du mot qui la contiendra $C = adresse.

Lors des rencontres suivantes on pourra utiliser l'adresse du mot qui contiendra la constante par la référence $C.

Pour réaliser certaines tables il sera nécessaire de prendre des précautions ; en effet certains indicatifs peuvent être confondus avec des identificateurs du programme, ce qui n'est pas le cas des constantes; il faudra alors effectuer une première modification de l'indicatif, en plaçant, par exemple, le symbole ## devant lui. Il est clair que cette technique est coûteuse car la création d'une variable en Snobol requiert un bloc de mémoire pour contenir les attributs de la variable. Mais c'est une manière simple d'implémenter une table à accès rapide.

1.4. Les piles.

Cette structure d'information est très utilisée en compilation. Les piles ne font pas partie des objets du langage, mais on a beaucoup de possibilités pour les représenter.

1.4.1. Au moyen de tableaux, comme on le ferait en Fortran. Mais en Snobol 4 les objets d'une pile ainsi représentée peuvent avoir des types différents, en effet : un élément d'un tableau peut être entier, l'autre une chaine, un autre encore un réel... Cette représentation a toutefois un inconvénient : il faut avoir fait une déclaration dynamique de tableau et donc avoir fixé le nombre maximum des éléments de la pile.

1.4.2. Une pile étant une liste sur laquelle on n'effectue que certaines opérations : ajouter un élément, enlever le dernier, on peut la représenter de façon chaînée (voir ch 0).

Cette implémentation est toutefois un peu lourde et nous avons préféré les suivantes.

1.4.3. Enfin, une pile peut être représentée au moyen d'une chaîne.

L'extraction du sommet de pile sera réalisée par filtrage et remplacement (par la chaîne vide), l'adjonction d'un élément par concaténation à gauche de la chaîne, pour permettre l'extraction par filtrage et remplacement.

Si les éléments sont de longueur fixe, on opérera exactement comme nous venons de le décrire :

Exemple : Les éléments sont de longueur N.

```
DEFINE('EMPILE(X)')   ; DEFINE('DEPILE(X)')   : (FIN.DEF)

EMPILE PILE = X PILE          :  (RETURN)

DEPILE PILE  PØS(0) LEN(N)  .  DEPILE  : S(RETURN) F(FRETURN)

FIN . DEF
```

Si les éléments sont de longueur variable, on les séparera par une marque soit '##', dans la définition précédente on remplacera les lignes EMPILE et DEPILE par :

```
EMPILE PILE = '##' X  PILE : (RETURN)

DEPILE PILE  PØS(0) '##' BREAK ('##'). DEPILE = : S(RETURN)
                                                  F(FRETURN)
```

1.5. Conclusion

Un compilateur conçu en Snobol ne peut pas prétendre à être efficace. Dans les domaines de l'expérimentation il est rapide à construire et facile à modifier. Lorsqu'on est en train de concevoir un nouveau langage on recherche plus à obtenir rapidement un compilateur plutôt que d'avoir un compilateur performant en cours de réalisation.

Il ne faut pourtant pas être trop optimiste, en effet, si le langage en cours de conception est de la complexité des langages usuels, la taille du compilateur risque d'être beaucoup trop importante.

En revanche pour concevoir des langages d'applications particulières cela peut être intéressant.

Dans l'enseignement de la programmation on peut se permettre de programmer un compilateur dans son ensemble. On peut écrire un compilateur interprète, réécrire un analyseur sans utiliser le processeur de filtrage

(les problèmes de traitement de chaînes se résolvent plus facilement au Snobol), implémenter différemment les tables....

Enfin, citons un certain nombre de problèmes du domaine de la compilation et qui pourraient être aussi traités en Snobol 4 :

1) La réalisation d'un interprète de cartes de commandes traduisant un langage de commande dans un autre langage de commande, ou pour l'enseignement d'un simulateur d'interprète vérifiant la validité et la cohérence des paramètres.

2) Un précompilateur d'une extension d'un langage, par exemple, FORTRAN 4 auquel on aurait ajouté le traitement des chaînes de caractères.

3) De même la réalisation d'un précompilateur de macro-instructions de PL/I.

2. REALISATION D'UN COMPILATEUR EN SNOBOL 4.

2.1. Définition du langage à compiler.

Pour donner un exemple de programmation d'un compilateur en Snobol 4, introduisons un langage simple : AGILES. C'est une sorte de sous-ensemble de FORTRAN qui ne traite que d'un type de données : les entiers.

Il comprend les instructions d'Affectation, le Goto, le If arithmétique, les ordres de Lecture et d'Ecriture qui transfèrent un entier à la fois en format I9 et un Stop, d'où son nom. Les expressions arithmétiques comprennent les opérations +, -, * , / . Les étiquettes sont numériques et nous n'avons par gardé la tabulation de Fortran : l'étiquette d'une instruction, si elle existe, doit commencer en colonne 1, elle est séparée du reste de l'instruction par un blanc ; une instruction n'ayant pas d'étiquette commence par un blanc.

Pour simplifier l'écriture de l'analyseur, pour ne pas encombrer les modèles avec des éléments sans intérêt, on a interdit les blancs dans les expressions arithmétiques et IF, GØTØ, LIRE, ECRIRE, STØP, sont des mots réservés.

2.1.1. Exemple de programme AGILES

```
      LIRE  Y
      S = 1
      X = 0
1     X = X + 1
      IF (S-Y)2,2,2
2     S = S*X
      ECRIRE S
      GØTØ 1
      STØP
```

2.2. Le compilateur d'AGILES

Le langage objet est Symbol, langage d'assemblage du C.I.I. 10 070. Ce choix a été fait pour des raisons de lisibilité et non d'efficacité ; en effet il faudra faire une seconde traduction pour mettre en oeuvre le texte généré.

On en tire, néanmoins, un avantage supplémentaire : les étiquettes seront traitées par l'assembleur ; à la compilation on se contentera de transformer l'étiquette E en étiquette Symbol E' et de générer :

```
E'    EQU    $
```

La génération s'effectue en même temps que l'analyse, compte tenu de la remarque faite plus haut.

Pour réaliser le compilateur nous avons appliqué les deux méthodes de programmation décrite précédemment : pour les parties du langage dont la grammaire est très récursive (expressions arithmétiques) nous avons construit la traduction en partant de la grammaire, pour le reste nous nous sommes laissés guider par les résultats à obtenir.

Le programme généré est formé :

- d'un prologue qui contient un certain nombre de directives pour l'assembleur ainsi que des instructions d'initialisation.

- d'une suite d'instructions Symbol.

- d'une suite de directives pour la réservation des variables et la définition des constantes.

Ces dernières lignes de directives sont composées au moment de la rencontre d'une variable (resp. d'une constante), elles sont écrites sur un fichier temporaire.

2.3. Analyse :

Initialisation du compilateur ; écrire le prologue ;
jusqu'à fin texte source faire : lire carte, trait 1 (carte)
fait

Recopier les lignes "RES" et "DATA" à partir du fichier temporaire.

```
écrire prologue :
sortie = composer (,'CSECT', '1')
sortie = composer (,'SYSTEM', 'SIG7F')
sortie = composer (,'SYSTEM', 'SIRIS7')
sortie = composer (,'SYSTEM', 'FØRTLIB')
sortie = composer ('$RG', 'BAL,6', '9INITIAL')
```

sortie est une variable de sortie pour le texte objet

```
trait 1(ch) :
si ch← éti blancs où éti est une étiquette,
suite le reste de la chaîne commençant au
premier caractère non blanc après éti
alors sortie = composer ('$' éti, 'EQU', '$')fsi.
traitcorps (suite) ; retour ;
```

composer une fonction qui aligne les différents champs des lignes Symbol

← signifie se décompose en

```
traitcorps (ch) :
si ch ← devant '=' derrière où devant
et derrière sont quelconques alors si
devant est un identificateur
alors traitexpr (derrière) ;
sortie = composer (,'STW,' IREG-1,devant);
ireg   = 1 ; retour
```

ireg est un compteur qui gère les registres utilisés dans la génération. La gestion des registres est très simplifiée.

```
fsi :
si ch ← 'GOTO' blancs éti où éti est une
étiquette alors sortie = composer (,'B','$'éti);
retour fsi


si ch ← 'IF('cond')'e1','e2','e3 où cond est
une expression arithmétique
e1, e2, e3 des étiquettes
alors traitexp (cond) ;
      sortie = composer (,'CI,'IREG - 1,0);
      Ireg   = 1
      sortie = composer (,'BL', '$' e1)
      sortie = composer (,'BEZ', '$' e2)
      sortie = composer (,'BEZ', '$' e3)

fsi


si ch ← es blancs ident
   où es est soit LIRE soit ECRIRE
   et Ident un identificateur
alors trident (ident);
      sortie = composer (,'LI,14,1);
      sortie = composer (,'BAL,15',es);
      sortie = composer (,'INTG', ident) ;
      retour ;
fsi.


si ch = 'stop' alors sortie = composer (,'CAL1,9',,'1')
                    retour   fsi
lister = ch 'instruction non reconnue'

- traitexpr(ch) : /* définie à partir du modèle expression arith */
          alpha = 'abcdefghijklmnopqrstuvwxyz'
             nu = '1234567890'
           alnu = alpha nu
         entier = span (nu)
          ident = any  (alpha) (span (alnu) |'')
           elem = ident . vid .  trident (vid)  entier  . vent  .  trc
                  (vent)
              f = elem |'(' *E ')'
              T = T '*'  *F  *g(2) | *T '/'  *F  *g(3) |  *F
              E = E '+'  *T  *g(0) | *E '-'  *T  *g(1) |  *T
           ireg = 1
             CH   E : S(RETURN) F(FRETURN)
```

```
- trident (x)  :  vy = '##' x                    $vy : voir technique de
                  si $vy = 0 alors $vy = 1;      représentation des
  sortemp = composer (x,'RES',1)fsi               tables
                  empiler(x)  ; retour.


- trc (x) : si $x = 0 alors $x = 1 ;
                  soit temp = composer ('$' x ,'DATA',x)
            fsi
            empiler(x) ; retour.


-g(x) :/   aiguillage  allera x
       0   code.op = 'aw' ; goto   génération . commune
       1   code.op = 'sw' ; goto   génération . commune
       2   code.op = 'mw' ; goto   génération . commune
       3   code.op = 'dw' ; goto   génération . commune

      génération . commune

          op1 = depile
          op2 = depile
        sortie= composer (, 'LW, 'ireg, ØP2)
        sortie= composer (, code.op','ireg,ØP1)
       empiler  (ireg) ;

si ireg <15 alors ireg = ireg + 1 ; retour
sinon lister = "débordement d'indice".


- Recopier "RES" et "DATA".
        rebobiner sortemp ;
        jusqu'à fin de fichier sortie = relire.
```

Pour passer de l'analyse à la programmation, nous allons regrouper le différents modèles nécessaires au début du programme car ils n'ont besoin d'être exécutés qu'une seule fois.

Les fonctions définies pendant l'analyse ne seront pas forcément définies au moyen d'une fonction dans le programme.

LE COMPILATEUR :

```
*                  ****************************
*                  *                          *
*                  * COMPILATION DE AGILES    *
*                  *                          *
*                  ****************************
*
*  DEFINITION DE SORTTEMP , RELIRE  VARIABLES D'ENTREE SORTIE
*  POUR CREE LE FICHIER INTERMEDIAIRE DES RESERVATIONS DES VARIABLES
*  ET DEFINITIONS DE CONSTANTES.
*
          OUTPUT('SORTTEMP',101,'(20A4)')                                    1
          INPUT('RELIRE',101,80)                                             2
          &ANCHOR = 1                                                        3
*
*  FONCTIONS DE MANIPULATION DE PILE
          DEFINE('EMPILER(X)')    ;  DEFINE('DEPILE(X)')   :(FIN.PILE)       4
          DEFINE('EMPILER(X)')                                               6
EMPILER   PILE = X '#' PILE    :(RETURN)                                     7
DEPILE    PILE BREAK('#') . DEPILE '#' =     :(RETURN)                      8
FIN.PILE
* DEFINITION DE COMPOSER
       DEFINE('COMPOSER(X,Y,Z)L,FIL1,FIL2')     :(FIN.FORM)                  9
                                                                            10
COMPOSER L = '                '                                            11
          L  LEN(SIZE(X)) TAB(15) . FIL1   :F(ERREUR)                       12
          L  LEN(SIZE(Y)) TAB(15) . FIL2   :F(ERREUR)                       13
          COMPOSER  = X FIL1 Y FIL2 Z L        :(RETURN)                    14
FIN.FORM
*
*  DEFINITION DE SORTIE
          OUTPUT('SORTIE',107,'(20A4)')                                     15
*   LES MODELES                                                            16
          ALPHA = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'                              17
          NU    = '0123456789'                                              18
          ALNU  = ALPHA NU                                                  19
          ETI   = SPAN(NU)                                                  20
          BLANCS = SPAN(' ')                                                21
          IDENT  = ANY(ALPHA) (SPAN(ALNU) | '')                             22
          ENTIER = SPAN(NU)                                                 23
          ELEM = IDENT . VID . *TRIDENT(VID) | ENTIER . VENT . *TRC(VENT)   24
          F    =  ELEM | '(' *E ')'                                         25
          T    =  *T '*' *F . *GE(2) | *T '/' *F . *GE(3) | *F               26
          E    =  *E '+' *T . *GE(0) | *E '-' *T . *GE(1) | *T               27
          DEFINE('TRAITCORPS(X)G')   :(FIN.CPS)                             28
TRAITCORPS                                                                  29
A         X  BREAK('=') . DEVANT '=' REM . DERRIERE      :F(G)              30
          DEVANT IDENT           :F(FRETURN)                                31
          IREG = 1                                                          32
          DERRIERE POS(0) E RPOS(0)      :F(FRETURN)                        33
          SORTIE = COMPOSER(,'ST',' IREG = 1,DEVANT)     :(RETURN)          34
                                                         :F(I)              35
G         X  'GOTO' BLANCS ETI . VETI                     :(RETURN)          36
          SORTIE = COMPOSER(,'B',,'$E' VETI)                                37
I         IREG  =  1                                                        38
          X  'IF(' E ')' ETI . E1 ',' ETI . E2 ',' ETI . E3  :F(LE)         39
          SORTIE = COMPOSER(,'BL',,'$E' E1)                                 40
          SORTIE = COMPOSER(,'BEZ',,'$E' E2)
          SORTIE = COMPOSER(,'BGZ',,'$E' E3)                :(RETURN)
```

```
LE      X ('LIRE' | 'ECRIRE') . ES BLANCS     IDENT . VID   :F(S)          41
        TRIDENT(VID)                                                       42
        SORTIE = COMPOSER(,'LI,14',,'1')                                   43
        SORTIE = COMPOSER(,'BAL,15',,ES)                                   44
        SORTIE = COMPOSER(,'INTG', VID)                 :(RETURN)          45
S       X 'STOP'                                         :F(INSNR)         46
        SORTIE = COMPOSER(,'CAL1,9',,'1')                :(RETURN)         47
INSNR   OUTPUT = 'INST NON REC. '                        :(RETURN)         48
FIN.CPS                                                                    49
*   DEFINITION DES FONCTIONS DE GENERATION POUR LES EXP ARITH
*                  TRIDENT ,TRC,G
      DEFINE('TRIDENT(X)') ; DEFINE('TRC(X)') ; DEFINE('GE(X)') :(FID)     50
TRIDENT VY = '#' X                                                         53
        TRIDENT = 'BIDON'                                                  54
        EQ($VY,0)     :F(PLIDENT)                                          55
        $VY = 1                                                            56
        SORTTEMP  = COMPOSER(X,'RES',,'1')                                 57
PLIDENT SORTIE = COMPOSER(,'LW,' IREG,X)                                   58
        EMPILER(IREG)                                                      59
        IREG = LT(IREG,15) IREG + 1    :S(NRETURN)                         60
        OUTPUT = ' DEBD REGISTRES'     :(RETURN)                           61
TRC     TRC = 'BIDON'                                                      62
        EQ($X,0)              :F(PLC)                                      63
        $X = 1                                                             64
        SORTTEMP = COMPOSER( '$' X,'DATA', X)                              65
PLC     SORTIE = COMPOSER(,'LW,' IREG,'$' X)                               66
        EMPILER(IREG)                                                      67
        IREG = LT(IREG,15) IREG + 1    :S(NRETURN)                         68
        OUTPUT = ' DEBD REGISTRES'     :(RETURN)                           69
GE      GE = 'BIDON'          :($X)                                        70
0       CODP = 'AW'     :(TRCOM)                                           71
1       CODP = 'SW'     :(TRCOM)                                           72
2       CODP = 'MW'     :(TRCOM)                                           73
3       CODP = 'DW'     :(TRCOM)                                           74
TRCOM   OP1  = DEPILE()                                                    75
        OP2  = DEPILE()                                                    76
        SORTIE = COMPOSER(,'LW,' IREG,OP2)                                 77
        SORTIE = COMPOSER(,CODP ',' IREG,OP1)                              78
        EMPILER(IREG)                                                      79
        IREG =  LT(IREG,15) IREG + 1    :S(NRETURN)                        80
        OUTPUT = 'DEBD REGISTRES'       :(NRETURN)                         81
FID                                                                        82
* GENERATION DU PROLOGUE
        SORTIE = COMPOSER(,'CSECT','1')                                    83
        SORTIE = COMPOSER(,'SYSTEM','SIG7F')                               84
        SORTIE = COMPOSER(,'SYSTEM','SIRIS7')                              85
        SORTIE = COMPOSER(,'SYSTEM','FORTLIB')                             86
        SORTIE = COMPOSER('$RG','BAL,6','9INITIAL')                        87
* TRADUCTION DES INSTRUCTIONS
BOUCLE  CARTE = TRIM(INPUT)       :F(FIN)                                  88
        PILE = ''                                                          89
        CARTE  ETI . VETI BLANCS REM . SUITE    :F(NIET)                   90
        SORTIE = COMPOSER('$E' VETI,'EQU','$')                             91
PRECED  TRAITCORPS(SUITE)      :(BOUCLE)                                   92
NIET    CARTE SPAN(' ') REM . SUITE    :(PRECED)                          93
*                                                                          94
FIN     ENDFILE(101) ; REWIND(101)
*RECOPIE
        SORTIE = COMPOSER(,'REF','9INITIAL,LIRE,ECRIRE')                   96
        SORTIE = COMPOSER(,'CSECT','0')                                    97
BCOP    SORTIE = RELIRE        :S(BCOP)                                    98
        SORTIE = COMPOSER(,'END')                                          99
```

```
      END                                                    100
NO ERRORS DETECTED DURING COMPILATION

NORMAL TERMINATION AT LEVEL  0
LAST STATEMENT EXECUTED WAS   99
SNOBOL4 STATISTICS SUMMARY-
          3924 MS. COMPILATION TIME
          5508 MS. EXECUTION TIME
           461 STATEMENTS EXECUTED,       28 FAILED
            19 ARITHMETIC OPERATIONS PERFORMED
           154 PATTERN MATCHES PERFORMED
             2 REGENERATIONS OF DYNAMIC STORAGE
         11.95 MS. AVERAGE PER STATEMENT EXECUTED
```

```
            CSECT       1
            SYSTEM      SIG7F
            SYSTEM      SIRIS7
            SYSTEM      FORTLIB
            9INITIAL
$RG         BAL,6       $0
            LW,1        X
            STW,1       Y
            LW,1        1
            LI,14       LIRE
            BAL,15      Y
            INTG        $1
            LW,1        X
            STW,1       $1
            LW,1        S
            STW,1       $
$E1         EQU         X
            LW,1        $1
            LW,2        1
            LW,3        2
            AW,3        X
            STW,3       S
            LW,1        Y
            LW,2        1
            LW,3        2
            SW,3        $E2
            BL          $E2
            BEZ         $E3
            BGZ         $
$E2         EQU         S
            LW,1        1
            LI,14       ECRIRE
            BAL,15      S
            INTG        S
            LW,1        X
            LW,2        1
            LW,3        2
            MW,3        S
            STW,3       $E1
            B           $
$E3         EQU         1
            CAL1,9      9INITIAL,LIRE,ECRIRE
            REF         0
            CSECT       0
$0          DATA        1
Y           RES         1
$1          DATA        1
X           RES         1
S           RES         1
            END
```

Programme objet généré correspondant au programme source suivant :

```
      X=0
      LIRE Y
      X=1
      S=1
1     X=X+1
      IF(S-Y)2,2,3
2     ECRIRE S
      S=S*X
      GOTO 1
3     STOP
```

5. IMPLEMENTATION DE SNOBOL 4

1. GENERALITES.

1.1. Les difficultés de la compilation de Snobol 4 ne résident pas dans l'analyse syntaxique : la structure du langage est très simple et on peut compiler ligne à ligne. Les expressions sont les seuls objets dont la syntaxe est complexe, mais le problème de leur analyse syntaxique n'est pas plus difficile que celui des expressions arithmétiques des autres langages, il suffit, en effet, de modifier les algorithmes existants en ajoutant deux opérateurs avec les priorités adéquates. Les difficultés viennent essentiellement du caractère extrêmement dynamique des programmes.

Rappelons, en effet, que :

a) il n'y a pas de déclaration et les valeurs d'une variable n'ont pas de type imposé : une variable peut changer de type au cours de l'exécution du programme et prendre successivement, par exemple, des valeurs de type entier, tableau de chaînes de caractères, modèle, ou même de type défini par le programmeur.

b) dans les tableaux il n'existe pas de contrainte d'homogénéité de type des éléments ; certains peuvent, par exemple, être des modèles, d'autres des chaînes ou des réels, etc...

c) les chaînes et les modèles ont une longueur variable en cours d'exécution sans qu'il soit possible de déterminer une taille limite.

d) des variables peuvent être créées au moment de l'exécution à partir d'une chaîne non vide grâce à l'opération $. c'est un problème différent de celui qui est posé par les variables contrôlées en PL/I, car au moment de la compilation il n'est pas possible de connaître le nom de telles variables.

e) des séquences de code objet peuvent être générées à l'exécution.

f) la définition des tableaux, des fonctions, des fonctions de création d'objets composés définies par le programmeur n'ont lieu qu'à l'exécution.

Un grand nombre d'objets sont créés à l'exécution ou sont amenés à changer de taille.

L'emplacement de tous les objets ne peut donc être connu à la compilation; par conséquence, l'allocation de mémoire doit se faire de façon dynamique.

Au cours de l'exécution, des objets qui ont été créés peuvent devenir inutiles car le langage ne contrôle pas leur durée de vie.

Il faudra prévoir une procédure de récupération de l'espace mémoire inutile (ramasse miettes).
Les deux principaux problèmes posés par la compilation de Snobol 4 sont donc la représentation des objets du langage et la gestion de l'espace mémoire.

1.2. Un compilateur portable.

Il existe un langage d'implémentation de SNØBØL 4 [6] SIL, formé de macro-instructions. Il suffit donc d'écrire le texte des macro - instructions pour réaliser une implémentation sur n'importe quel ordinateur.

Cette implémentation est un outil d'aide à la définition de Snobol 4 : elle facilite la mise à jour du compilateur au fur et à mesure des modifications du langage. Les objets du langage y sont représentés par des blocs de mémoire contigus de longueur variable.

Cette représentation permet d'ajouter facilement de nouveaux types d'objets, même s'ils demandent, pour être représentés, des blocs de taille différente des objets implémentés.

Une telle représentation facilite l'allocation mémoire, dans le cas de l'adjonction de nouveaux éléments, mais l'accès aux informations que contiennent les blocs n'est pas nécessairement bien adapté au traitement.

C'est le cas pour les chaînes de caractères qui sont alors représentées de façon compacte.

Nos objectifs étant différents de ceux de l'auteur du langage, nous avons recherché à rendre le compilateur efficace en choisissant une représentation des objets qui facilite le traitement et économise la mémoire. Nous n'avons donc pas utilisé ce mode d'implémentation.

1.3. Compilateur à code exécutable ou interprétable ?

La concaténation de deux chaînes de caractères de longueurs quelconques, le filtrage, le filtrage et remplacement, sont des opérations qui ne trouvent pas leur correspondant dans les langages des machines actuelles qui restent encore orientées vers le calcul numérique.

Les opérations y travaillent principalement au niveau du mot : c'est le cas pour le CII 10 070, qui permet cependant d'atteindre le caractère et possède des opérations de transfert et de comparaison de chaînes représentées de façon compacte. Le répertoire des instructions sur les chaînes de caractères n'est cependant pas suffisamment riche pour nous faire choisir un compilateur à code exécutable, d'autant plus que le caractère très dynamique du langage gêne beaucoup la génération de code exécutable.

Ainsi, pour compiler l'affectation X = $Y + 1 on ne pourra pas associer à $Y l'adresse d'un emplacement en mémoire ; de même on ne pourra pas faire la liaison entre un appel de fonction et la définition correspondante qui est faite à l'exécution...

Pour ces deux raisons on est plus ou moins amené à réaliser un interprète, ou, ce qui revient au même à générer surtout des appels de sous-programmes. C'est ce choix que nous avons fait et nous avons réalisé un compilateur dont le langage objet est interprété.

Cependant, postérieurement à l'étude menée, un compilateur à code exécutable, SPITBØL, a été réalisé [6] . Il est plus efficace (3 à 10 fois plus rapide et en moyenne 8 fois) que le compilateur écrit avec SIL. Il restreint cependant le langage.

Notre but était, au contraire, d'implémenter le langage tel qu'il était défini dans [1] . D'ailleurs, si on regarde SPITBØL, son efficacité ne réside pas essentiellement dans son choix de générer du code exécutable. La compilation d'une affectation dont le premier membre est une variable simple et non une expression et le second membre une expression arithmétique est certainement plus efficace en SPITBØL, mais ce compilateur ne peut pas être plus efficace dans des cas plus généraux, et en particulier dans le filtrage. On trouve en Snobol 4 des fonctions qu'on peut appeler de type système, qui ralentissent l'efficacité de l'exécution car elles se répètent très souvent. Il s'agit, par exemple, de compter les instructions exécutées et de s'assurer que le maximum permis n'est pas dépassé, de contrôler si la variable à laquelle on veut affecter une valeur n'est pas en demande de trace....

Ces fonctions sont implémentées de façon efficace en SPITBØL.

La détection du nombre maximum d'instructions exécutées est réalisée grâce à l'interruption de dépassement de capacité en mémoire.

De telles améliorations, que l'on pourrait généraliser aux variables associées aux E/S, aux variables en demande de trace, peuvent aussi bien être réalisées dans un système compilateur - interprète.

2. REPRESENTATION DES OBJETS.

2.1. Les objets à représenter.

Il y a divers objets à représenter : les entiers et les réels, les chaînes de caractères, les identificateurs si on ne veut pas leur donner la même représentation que les chaînes, les modèles, les variables avec leurs différents attributs (E/S, fonctions, étiquettes ....,), les tableaux, les objets composés définis par le programmeur, et enfin le code objet qui peut être créé dynamiquement par la fonction CØDE. Notre objectif étant de donner une représentation efficace des objets les plus importants nous allons étudier les différentes solutions envisageables.

2.2. Représentations classiques des chaînes de caractères.

Une chaîne de caractères est une liste au sens de [11]. On peut donc la représenter en mémoire de façon contiguë, chaînée ou mixte c'est-à-dire comme une suite de sous-chaînes représentées de façon compacte et chaînées entre elles.

La représentation compacte est de prime abord plus économique en mémoire, mais elle oblige dans les traitements (par exemple, la concaténation) à effectuer des recopies, d'où des pertes de temps et de place en mémoire.

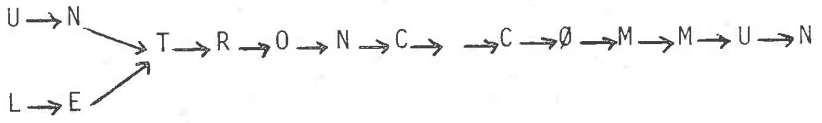
Exemple : La suite d'affectations suivante :

```
TRC  =  'TRØNC CØMMUN'
CH1  =  'UN '  TRC
CH2  =   'LE ' TRC
```

donnera en mémoire :

TRØNC CØMMUNUN TRØNC CØMMUNLE TRØNC CØMMUN

La représentation chaînée permet au contraire le partage des seconds termes des concaténations, d'où une représentation en mémoire des chaînes créées par les affectations de l'exemple précédent :



Mais on ne peut pas effectuer de partage de début de liste, par exemple, les affectations suivantes :

```
CH3  = TRC '  SIMPLE'
CH4  = TRC '  UNIQUE'
```

obligent à recopier T→R→O→N→C→ →C→Ø→M→M→U→N.

D'autre part la place prise par le lien est importante. On peut diminuer la place prise par les liens par rapport à la place prise par les caractères en chaînant non plus des caractères mais des sous-chaînes (de longueur fixe ou variable) représentées de façon compacte.

Lorsque l'on veut travailler à l'intérieur des sous-chaînes on retrouve alors les inconvénients de la représentation compacte.

2.3. Les chaînes de caractères en Snobol 4.

Les chaînes traitées en Snobol ont deux origines. Il y a les chaînes constantes du programme et les chaînes obtenues à l'exécution par lecture ou par conversion de valeurs arithmétiques. Ces chaînes ont à un certain moment une représentation compacte, nous les appelerons des chaînes initiales.

Il y a les chaînes obtenues par calcul à partir des chaînes initiales ou de leurs sous-chaînes. Faut-il donner à ces chaînes une des représentations précédentes ?

La représentation compacte est intéressante pour deux raisons : d'une part pour l'édition des résultats (nécessité de ranger la chaîne dans une mémoire tampon), d'autre part elle facilite l'accès aux caractères en séquence dans l'opération de filtrage. Nous avons signalé ses inconvénients.

Avant de répondre à la question posée il serait bon de regarder les problèmes de représentation des autres éléments du langage et en particulier des modèles et du code objet.

Les modèles, d'ailleurs, ont une grande parenté avec les chaînes : une chaîne peut être un modèle, l'opérateur de concaténation est aussi un opérateur sur les modèles.

2.4. Problèmes posés par la représentation des modèles.

2.4.1. Modèles et opérations sur les modèles.

D'une façon externe le modèle se présente sous forme d'une expression

dont les opérandes peuvent être : des chaînes de caractères, des identificateurs, des modèles standard et fonctions standard , des fonctions, des prédicats, ainsi que des expressions arithmétiques que nous considérons pour simplifier comme formant un tout.

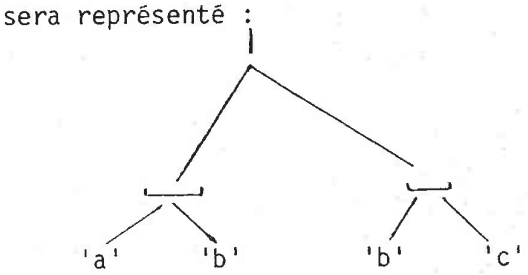
Ces expressions contiennent des opérateurs binaires : l'alternative |, la concaténation " ", les affectations immédiates $ et conditionnelles; . ainsi que des opérateurs unaires comme l'évaluation retardée *, l'opérateur d'affectation de la position de repère de filtrage dans la chaîne sujet, etc...

Du point de vue de la syntaxe, les expressions généralisent les expressions arithmétiques, dans le programme objet il est alors naturel de les représenter sous forme de ramifications ayant des opérandes aux feuilles et des opérateurs aux noeuds.

Exemple : A la compilation le modèle :

'a' 'b' | 'b' 'c'

sera représenté :



A l'exécution en revanche, les expressions ayant pour valeur des modèles ne se traitent pas comme les expressions arithmétiques.

En effet, dans une expression ayant pour valeur un modèle, tous les opérateurs ne peuvent être traités de la même manière selon qu'on définit le modèle (évaluation de l'expression ayant pour valeur un modèle, pendant l'exécution d'une instruction d'affectation, par exemple) ou qu'on l'utilise dans un filtrage.

En effet, les opérateurs binaires |, $ et ., ainsi que les opérateurs unaires* et @ ne sont traités que par l'opérateur de filtrage ; l'opérateur de concaténation peut être traité à l'évaluation si ses deux opérandes sont des chaînes de caractères, mais dès que l'un des opérandes est un modèle standard la concaténation ne peut être interprêtée que par l'opérateur de filtrage ; les opérateurs arithmétiques peuvent être traités au moment de l'évaluation.

2.4.2. A l'évaluation faut-il traiter la concaténation dans les modèles ?

Si on décide de le faire lorsque c'est possible, le traitement dépend du type des opérandes, ce qui est gênant.

Quel intérêt y a-t-il à concaténer les chaînes dans les modèles ? Si les chaînes sont représentées de façon compacte, on facilite l'accès aux caractères en séquence pour le filtrage. Mais, s'il y a des alternatives on n'est pas certain que le processeur de filtrage utilise la chaîne ainsi créée ; d'autre part, pour faire la concaténation on a dû recopier les deux opérandes d'où une perte de temps et de place en mémoire.

Si les chaînes de caractères sont représentées avec des chaînages alors on ne gagne rien du point de vue de l'accès aux caractères mais on perd du temps et éventuellement de la place pour faire la concaténation. D'autre part, lorsqu'une expression ayant pour valeur une chaîne de caractères est précédée de l'opérateur * on n'effectue pas les concaténations.

Pour toutes ces raisons on choisit de ne pas traiter la concaténation à l'évaluation. En revanche on évalue les expressions arithmétiques qui ne sont pas précédées de l'opérateur * car ceci permet de remplacer une sous-ramification par un entier ou un réel, ce qui gagne de la place.

En résumé, à la compilation, un modèle est représenté par une ramification et pendant l'évaluation on effectue des opérations qui transforment la ramification en une autre ramification, en remplaçant, par exemple, une sous-ramification qui représente une expression arithmétique par sa valeur entière ou réelle.

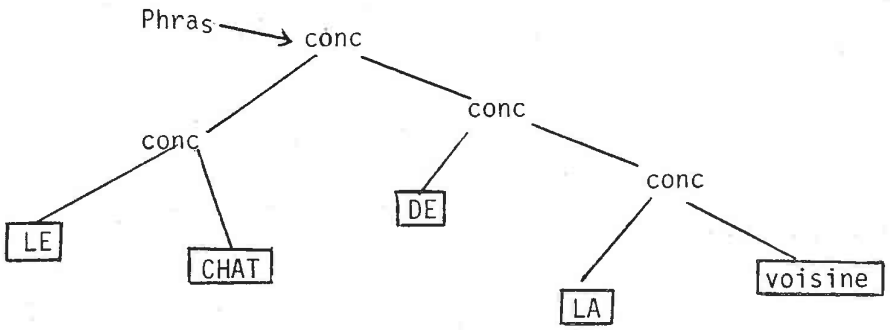
2.4.3. Application à la représentation des chaînes de caractères.

Une chaîne pouvant être un modèle, on décide de considérer que toutes les chaînes créées à l'exécution seront représentées comme les modèles par des ramifications dont les feuilles sont des chaînes initiales ou des parties de chaînes (cf 2.3.).

Exemple : la chaîne de nom PHRAS, créée à la suite de l'exécution des affectations suivantes :

```
ART1 = 'LE' ; ART2 = 'LA'
CONJ = 'DE' ; HUMAIN = 'VOISINE' ;
ANIMAL = 'CHAT' ; ØBJET = ART2 HUMAIN ;
SUJET = ART1 ANIMAL
PHRAS = SUJET CØNJ ØBJET
```

sera représentéede la façon suivante :

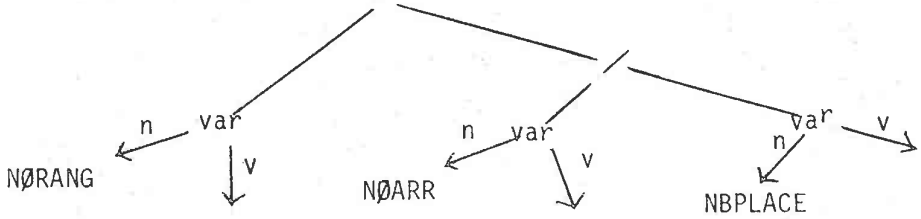


2.5. Le programme objet.

Les expressions occupent une très grande place dans les instructions du programme source et elles sont représentées par des ramifications dans le programme objet. Il est alors naturel de représenter également les instructions par des ramifications. Le programme objet sera alors une ramification à plusieurs racines, une par instruction et par champ GØTØ.

Exemple :

1) l'affectation suivante : NØRANG = NØARR / NBPLACE sera représentée de la manière suivante :

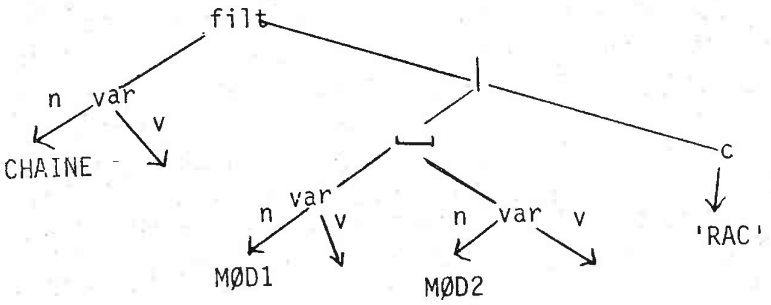


Les flèches N et V qui partent du noeud "var"(qui représente une variable) représentent des accès respectivement au nom et à la valeur.

2) Le filtrage suivant :

CHAINE ␣ MØD1 ␣ MØD2 | 'RAC'

est représenté :

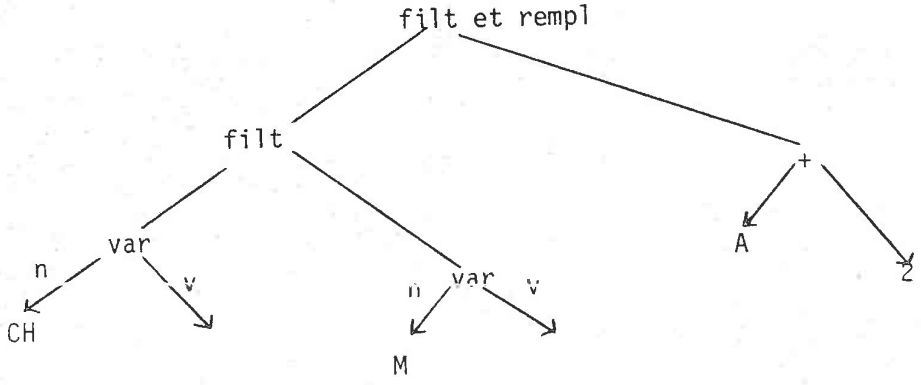


c représente un descripteur de chaînes de caractères.

3) Le filtrage et remplacement suivant :

CH ␣ M = A + 2

par



En représentant les modèles, les chaînes de caractères et les instructions du programme objet, par des ramifications, on facilite les problèmes d'allocation de mémoire.

2.6. Représentation choisie pour les ramifications.

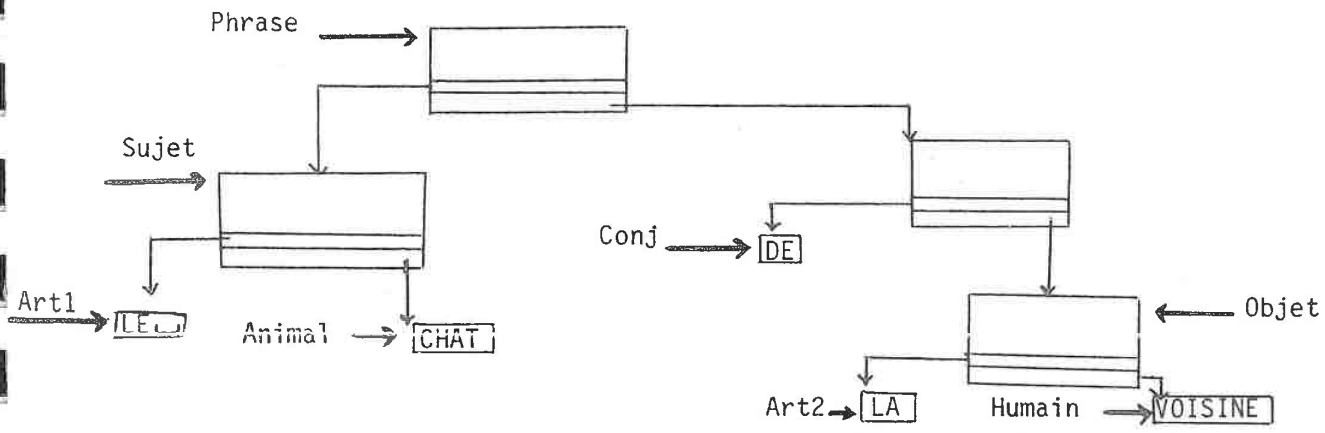
Parmi les représentations possibles des ramifications, nous écarterons la représentation postfixée qui est mal adaptée au calcul des fonctions. En effet, en Snobol 4 l'évaluation des arguments d'une fonction est réalisée par la fonction elle-même, c'est l'appel par nom d'Algol 60. Il est préférable d'accèder à la fonction avant d'accèder à ses opérandes ce qui n'est pas aisé lorsqu'on a choisi la représentation postfixée.

La représentation préfixée est économique en mémoire, mais nous lui préférerons la solution qui consiste à représenter les noeuds de la ramification par des blocs de mémoire et les liens entre un noeud et ses successeurs par des pointeurs vers les blocs qui représentent les successeurs.

Cette solution permet de faire du partage de valeurs à tous les niveaux, ce qui n'est pas le cas avec la représentation préfixée.

Si on chaîne les différentes racines de la ramification qui représente le programme objet on peut définir facilement la concaténation de deux séquences de code objet.

Exemple : Dans l'exemple du (§ 2.4.3.) la représentation des valeurs des différentes variables donne :



Les variables sont représentées par des lettres italiques et l'accès à leur valeur par des flèches ⟶.
Les autres flèches sont les liens des noeuds vers leurs successeurs.

2.7. Représentation des autres objets.

2.7.1. Les variables.

Dans la plupart des langages de programmation, les identificateurs des variables sont traités à la compilation et ne sont pas utilisés à l'exécution.Une exception est faite pour les éléments de la liste d'un ordre GET DATA en PL/I.

En Snobol, on peut faire référence à une variable de façon statique, par son nom, comme dans les autres langages, c'est le cas de ØPERATIØN dans l'instruction suivante :

```
ØUTPUT  =  'TITRE : ' ØPERATIØN
```

mais également de façon dynamique, au moyen de l'opérateur $, comme dans l'exemple précédent où la troisième instruction a même effet que l'instruction ci-dessus :

```
Exemple :  MØNUMENT  = 'ØPERA'
           TERM      = 'TIØN'
           ØUTPUT    = 'TITRE : '  $(MØNUMENT TERM)
```

Les variables seront représentées par un bloc de mémoire à partir duquel on peut accèder au nom de la variable et à sa valeur.

A la première occurrence de la variable, qu'elle existe dans le texte ou qu'elle soit créée dynamiquement il faudra générer un bloc et l'initialiser ; lors des occurrences suivantes il faudra obtenir l'adresse du bloc représentant la variable.

Pour savoir si une variable est déjà représentée et dans ce cas pour connaître l'adresse du bloc qui la représente on organise les variables en table au sens de [11] .

Les accès étant fréquents nous allons utiliser pour représenter la table une technique de rangement dispersé. (hashing)

Rappelons que cette technique consiste à partager la table en sous-tables au moyen d'une fonction d'adressage non injective, définie sur les identificateurs.

Une table, appelée table majeure, permet d'accèder à chacune des sous-tables. Les variables appartenant à une même sous-table seront chaînées ; pour accélérer la recherche dans une sous-table on pourra ranger la sous-table par ordre des longueurs croissantes des identificateurs. Pour plus de détails sur les techniques de rangement dispersé on aura intérêt à se reporter à [11].

Le nom d'une variable peut provenir du texte source ou d'un calcul ; on le représentera comme les autres chaînes.

Une variable peut avoir des attributs supplémentaires, comme celui, par exemple, d'être une étiquette. Il faudra donc prévoir un emplacement pour coder les attributs, le type de la variable et un pointeur pour l'étiquette.

Une variable sera donc représentée par un bloc de mémoire contenant un pointeur vers le nom, un accès à la valeur (dans le cas où la valeur est une chaîne ou un modèle, il s'agira d'un pointeur), un pointeur pour l'organisation de la table, un pointeur pour son rôle éventuel d'étiquette, divers indicateurs de type, d'attribut.

2.7.2. Les entiers et les réels.

On représente les entiers et les réels sur un mot. Dans les compilateurs des autres langages de programmation on évite d'avoir deux exemplaires d'une même constante représentés en mémoire ; pour cela on les organise en table.

Les valeurs numériques ne sont pas très fréquentes en Snobol et il est alors vain de s'efforcer à ne représenter qu'un exemplaire de la même constante dans le programme. D'autre part, comme on utilise dans tous les cas un pointeur pour repérer les valeurs, on pourra toujours remplacer le pointeur vers la valeur par la valeur elle-même dans le cas des constantes numériques.

2.7.3. Les tableaux, les objets composés définis par le programmeur, les fonctions.

Il s'agit de représenter des variables qui ont des attributs spéciaux. En effet, on peut déjà avoir utilisé un bloc pour représenter une

variable simple de nom N, par exemple, lorsqu'on va exécuter l'une des instructions suivantes :

N = ARRAY('-8 : 3'), **DEFINE**('N(X)'), **DATA**('N(E,R)')

et en plus N peut être une étiquette.

Que faut-il représenter en plus de la variable, pour chaque type d'attribut ?

Pour les tableaux : les caractéristiques (nombre de dimensions et bornes), ainsi que les valeurs des différents éléments, c'est-à-dire une liste de mots contenant soit des valeurs, soit des pointeurs.
Cette liste pourra être représentée dans des blocs chaînés.

Pour les fonctions: la liste des paramètres et des variables locales, c'est-à-dire une liste de pointeurs vers des blocs représentant des variables.Cette liste pourra également être représentée dans des blocs chaînés.

Pour les objets composés : Les différents champs (liste de pointeurs vers des blocs de variables).
On pourra encore adopter la même représentation que pour les tableaux et les fonctions.

2.8. Conclusion.

A la compilation le programme objet sera représenté par une ramification (à plusieurs racines), elle-même représentée par un chaînage des racines. Une ramification représente soit le corps d'une instruction, soit le champ GØTØ s'il existe. Les noeuds des ramifications représentent des opérateurs (l'affectation, le filtrage, le filtrage et remplacement étant considérés comme des opérateurs) ou des fonctions.

Pour les opérateurs, le noeud contient un ou deux accès vers le(s) sous-ramification(s) qui représente(nt) le(s) opérande(s) selon qu'il est monadique ou dyadique.

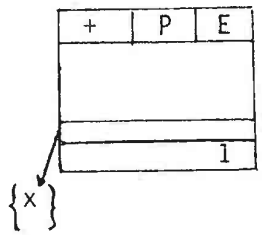
Pour les fonctions, le noeud contiendra un accès vers la liste des arguments, chacun pouvant être à son tour une ramification.

Les feuilles représentent les opérandes élémentaires. Lorsqu'il s'agit d'un entier ou d'un réel, il figure dans le noeud correspondant à

l'endroit prévu pour représenter l'accès vers la valeur.

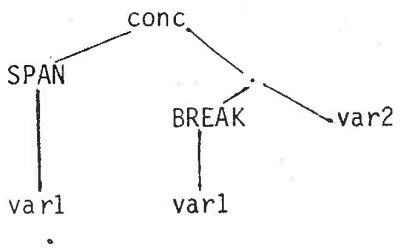
Exemple : X + 1

sera



Les modèles seront représentés par des ramifications.

Exemple : SPAN(CH) BREAK(CH) . V sera représenté par :



Var1 et Var2 désignant les variables nom CH et V.

Les chaînes de caractères représentées sont des chaînes initiales, étiquettes, identificateurs et suite de caractères délimités par deux apostrophes ou deux guillemets.

On les représentera de manière à économiser de la mémoire, c'est-à-dire de façon contiguë, ou à défaut de façon mixte au moyen de blocs chaînés. Un bloc contenant un pointeur de tête et un pointeur de queue, repérera la chaîne initiale.

Exemple : le bloc suivant représente la chaîne "aventure".

Les variables sont organisées en table, un bloc représentant une variable contient un pointeur vers la chaîne de caractères de l'identificateur. La valeur sera initialisée à 0.

Une étiquette sera traitée comme une variable, mais contiendra en plus un indicateur indiquant qu'elle a l'attribut "étiquette" et un pointeur vers la racine de la ramification qui va être générée.

Ainsi, soit le programme suivant :

```
          RAC = 'RØSA'
         DESI = 'S' | 'M' | 'E'
       MØDELE = RAC  DESI
B          CH = TRIM(INPUT)     :F(END)
           CH   MØDELE     :S(E1)
       ØUTPUT= CH 'ERREUR'     :(B)
E1     ØUTPUT= CH 'SUCCES'     :(B)
END
```

Le programme objet peut être représenté par la figure p. 5.16.
Pour des raisons de clarté on a omis le pointeur vers la valeur initialisée à 0 (code de la chaîne vide) et les chaînages de l'organisation en table des variables.

Les attributs rencontrés sont : e pour étiquette, f pour fonction, I/O pour entrée - sortie.

Certaines variables et certains attributs existent avant la compilation; c'est le cas par exemple de INPUT et ØUTPUT avec l'attribut I/O, TRIM avec l'attribut fonction, END avec l'attribut étiquette.

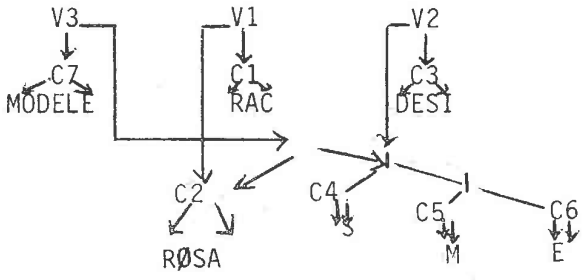
Quelles seront les modifications apportées à ce schéma au cours de l'exécution des différentes instructions du programme ?

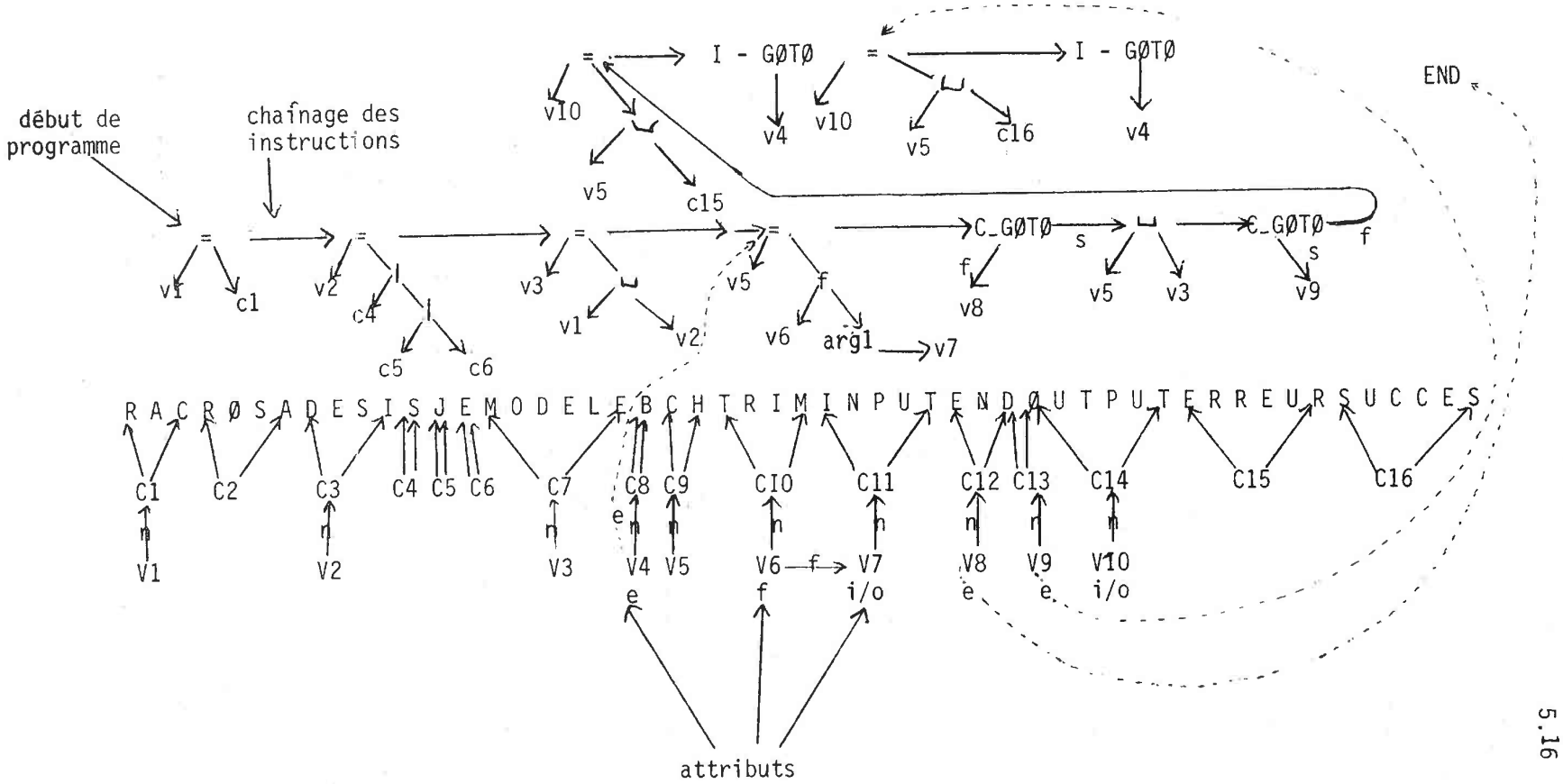
Une affectation implique deux opérations qui sont successivement :

- l'évaluation de l'expression du second membre, ce qui consiste à recopier la ramification qui représente l'expression en remplaçant les pointeurs vers les variables par des pointeurs vers les valeurs correspondantes, en calculant la valeur des sous-ramifications qui représentent des expressions arithmétiques et en exécutant les fonctions,

- l'établissement d'un pointeur de la variable à la ramification obtenue.

Ainsi après exécution des trois premières instructions du programme, aura-t-on :





Vi représente les variables
Ci sont les descripteurs de chaînes
v : les pointeurs vers les V1 correspondants
c : C1 figure

—n→ désigne l'accès de la variable au nom
- -e- -> repère l'instruction étiquettée
—f→ pointeur vers la procédure correspondante

lorsqu'on exécute l'instruction CH = TRIM (INPUT), une carte est lue (INPUT) et TRIM élimine les caractères blancs à droite dans la chaîne lue, les caractères restants figureront à la suite des caractères de la chaîne C16, un descripteur C17 sera créé. Puis si on boucle sur l'instruction des descriptions C18,C19 ... pour de nouvelles chaînes lues.

Dans ce programme seuls la dernière chaîne lue et son descripteur sont utilisés. Nous verrons un peu plus loin comment récupérer ces zones de mémoire.

3. GESTION DE LA MEMOIRE.

Le problème de gestion de mémoire se décompose en problème d'allocation et de récupération des espaces qui ne sont plus utilisés.

3.1. Allocation.

En Snobol, l'allocation est essentiellement dynamique. On dispose d'un espace mémoire, appelé espace libre et à un certain moment de l'exécution on a besoin d'une ou plusieurs zones de mémoire pour représenter un objet (ramification lors de l'évaluation d'une expression, chaîne lue...).

Parmi les objets à représenter qui nécessitent une allocation dynamique de mémoire on trouve les chaînes initiales et des objets qui sont représentés par des blocs éventuellement chaînés entre eux.

Pour faciliter la gestion de la mémoire on choisit des blocs de taille unique pour représenter ces objets.

Les blocs les plus fréquents sont ceux qui représentent les ramifications; leur taille nous servira à déterminer une taille unique de blocs.
Si les chaînes initiales sont représentées de façon contiguë, leur allocation de mémoire doit se faire de façon contiguë, ce qui n'est pas nécessaire pour les blocs entrant dans la représentation des autres objets. D'autre part il est plus naturel et plus efficace de disposer d'un espace libre chaîné pour les blocs.

On pourrait donc employer deux espaces libres, un espace libre contigu pour les chaînes initiales et un espace libre chaîné pour les blocs, ce qui obligerait à diviser la zone dynamique de la mémoire en deux.

Cette solution présente un inconvénient : un espace libre peut être saturé tandis que l'autre ne l'est pas. Il est possible de prélever des mots de l'espace contigu pour les donner à l'espace libre chainé, mais la réciproque n'est pas souvent possible.

Aussi, choisissons-nous plutôt une solution qui consiste à n'avoir plus qu'un seul espace libre.

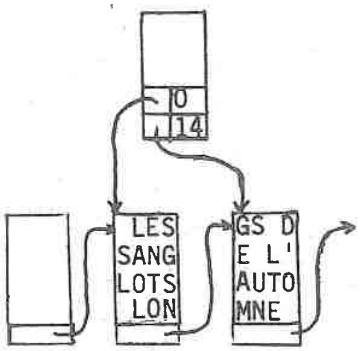
La gestion des blocs avec un espace libre contigu complique la procédure de restitution, à cause des décalages et des mises à jour de pointeurs.

Nous choisissons donc un seul espace libre chaîné. Nous représenterons, finalement, les chaînes initiales dans des blocs chaînés de longueur fixe.

Un pointeur vers le début d'une chaîne initiale sera un couple formé de l'adresse du bloc qui contient le début de la chaîne et de la position du premier caractère de la chaîne dans le bloc.

Exemple :"·LES SANGLØTS LØNGS DE L'AUTØMNE"

sera représentée de la façon suivante :



3.2. Restitution.

3.2.1. Restitution des blocs ne contenant pas les chaînes initiales.

Lorsque la liste libre est vide, il faut rechercher à la reconstituer avec des blocs qui ont été alloués mais qui ne font plus partie d'aucune information.

En partant de la table des variables et en parcourant les chaînages, on détermine les blocs occupés. Puis on applique une procédure de ramasse-miettes (garbage collector) pour rendre les blocs inutilisés à la liste libre. On initialise l'espace libre en découpant l'espace dynamique en blocs de taille fixe et en les chaînant.

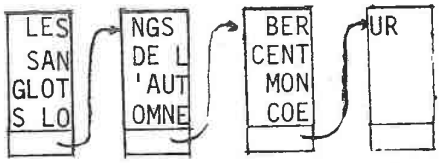
3.2.2. Restitution des blocs contenant les chaînes.

Si la procédure précédente ne suffit pas on peut libérer l'espace occupé par des chaînes contenues dans des blocs ; pour libérer des blocs on effectuera un tassage.

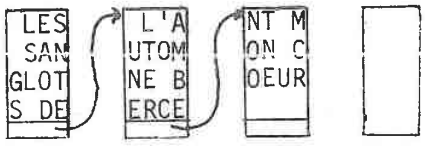
Exemple : Dans la chaîne suivante :

" LES SANGLØTS LØNGS DE L'AUTØMNE BERCENT MON COEUR"

représentée ainsi :



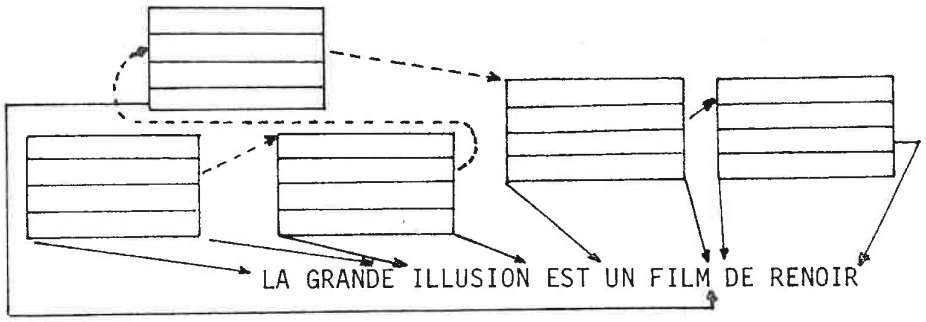
on a déterminé que la sous-chaîne LONGS est inutilisée. On effectue une translation des caractères en suivant les chaînages ce qui donne :



On a accès aux chaînes par les descriptions de chaînes ; lorsqu'on a cherché à libérer les blocs on a marqué les descripteurs de chaînes.

occupés. Mais les blocs non occupés ne suffisent pas à déterminer les chaînes qui doivent être libérées, à cause du partage. Pour cela on ordonne les descripteurs de chaînes en fonction du rang du début de la chaîne qu'ils définissent dans la liste des chaînes initiales.

Exemple :



Pour clarifier on n'a pas représenté les chaînes initiales dans des blocs chaînés.

Avec cette organisation des descripteurs, la restitution est plus simple qu'une restitution classique à un espace libre contigu.

Enfin on pourrait certainement utiliser avantageusement une mémoire virtuelle pour représenter les chaînes de caractères qui pourraient alors être rangées de façon compacte dans un certain nombre de pages de mémoire.

4. LE COMPILATEUR REALISE.

Les principes de compilation de Snobol 4 que nous venons de décrire en évitant d'entrer dans les détails ont été dégagés lors des études qui nous ont amenés à construire un compilateur en collaboration avec la SESA, sur ordinateur CII 10070, sous système BPM.

Un certain nombre de ces idées ont été retenues pour la solution finale qui constitue un compromis avec la solution proposée par la SESA.

Toutefois il est bon de noter certaines différences avec ce que nous avons écrit plus haut.

La mémoire a été découpée en sept parties qui contiennent chacune une table, par exemple une pour les entiers et une pour les réels.

Les identificateurs sont considérés comme des chaînes particulières et ont une représentation différente des autres chaînes. La représentation du code objet et des modèles sous forme de ramifications représentées par des blocs chaînés a été retenue.

La fonction CODE n'a pas été implémentée ; nous pensons que la représentation du programme objet sous forme d'une table des instructions ne facilite pas cette implémentation.

5. UNE MEILLEURE REPRESENTATION POUR LES MODELES.

5.1. Introduction.

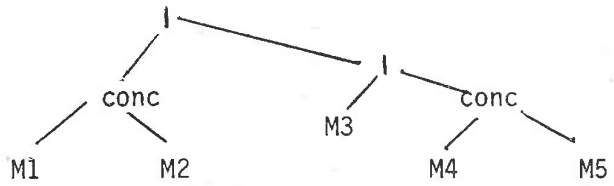
Dans la solution décrite plus haut, nous avons cherché à donner une représentation unique à trois types d'objets: les chaînes, les modèles et le code objet, qui occupent une place importante dans le langage et qui nécessitent une allocation dynamique.

Il serait aussi intéressant de se donner pour but de représenter les modèles de façon à faciliter leur parcours par le processeur de filtrage, la représentation sous forme de ramification ne répondant pas bien à cet objectif.

En effet, étant donné les modèles élémentaires M1,M2,M3,M4,M5 le modèle suivant :

M1 M2 | M3 | M4 M5

est représenté par la ramification :



Si M1 échoue le filtrage doit alors examiner M3. Pour passer de M1 à M3 il faut alors remonter à la racine et parcourir deux chaînages.

D'où l'intérêt de représenter pour chaque modèle élémentaire M deux accès : l'accès vers le modèle qui sera examiné après M si M réussit et l'accès vers le modèle qui sera examiné après M si M échoue.

5.2. Terminologie.

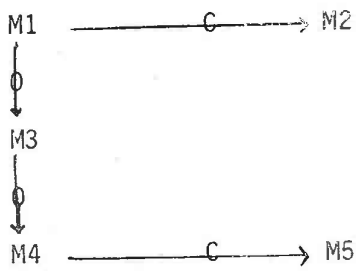
Le modèle M1 qui est examiné après M en cas de succès de M est appelé C-successeur de M.
Le modèle M2 qui est examiné après M en cas d'échec de M est appelé O-successeur de M.

Les modèles élémentaires qui ne possèdent pas de C-successeurs sont appelés C-terminaux, ceux qui ne possèdent pas de O-successeurs sont appelés O-terminaux.

On représentera les modèles en représentant les modèles élémentaires et en établissant des liaisons C-successeur et O-successeur, on appelera C-lien une liaison entre un modèle élémentaire et son C-successeur et O-lien une liaison entre un modèle élémentaire et son O-successeur. Un modèle élémentaire vers lequel n'arrive aucun lien est la racine du modèle.

5.3. Représentation des modèles.

On représente les modèles élémentaires, les O-liens et C-liens. Ce qui donne pour le modèle précédent :



M2,M3,M5 sont C-terminaux.

M4,M2,M5 sont O-terminaux.

Cette représentation nous amène à étudier comment effectuer la concaténation et l'alternative.

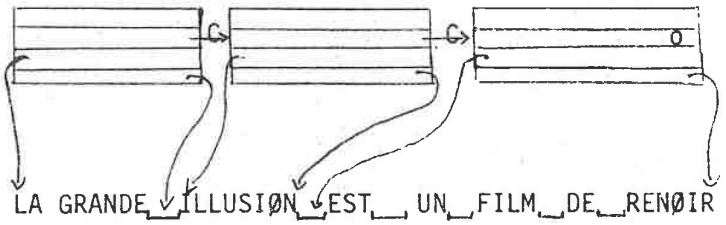
On sait effectuer ces opérations lorsque les modèles sont élémentaires. Soient M1 et M2 deux modèles évalués ; on évalue le modèle M1 M2 en établissant un C-lien entre chaque C-terminal de M1 et la racine de M2 ; on évalue le modèle M1 | M2 en parcourant les O-liens depuis la racine de M1 jusqu'à la rencontre d'un O-terminal et en établissant un O-lien entre ce O-terminal et la racine de M2.

5.4. Représentation des chaînes.

Si on considère les chaînes comme des modèles particuliers et si on traite la concaténation comme pour les modèles les chaînes seront représentées par chaînage des descripteurs de chaînes initiales au moyen de C-lien.

Exemple :

L'information suivante :



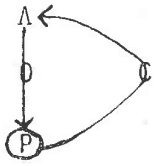
représente la chaîne valeur de PHRAS obtenue à la suite de l'exécution des instructions suivantes :

```
QUALIFICATIF  =  'LA GRANDE'
          NOM  =  'ILLUSION'
          FIN  =  'EST UN FILM DE RENOIR'
        PHRAS  =  QUALIFICATIF NØM FIN
```

Cette représentation des chaînes avait été trouvée au cours des études pour la réalisation du compilateur Snobol 4 et avaient été baptisées "listes supports de valeurs", réf. 9.

5.5. Conclusion.

Nous avons là la meilleure représentation pour les modèles ; il est également assez naturel de représenter ainsi les chaînes, l'accès aux caractères en séquence est alors plus facile que dans une ramification.
Cette solution présente un autre intérêt : un certain nombre de fonctions modèles peuvent être représentées à partir de modèles plus simples : c'est le cas, par exemple,de ARBNØ.
ARBNØ(P) est équivalent au modèle Λ | P | P ..... qui peut être représenté par :



CONCLUSION

La première partie de ce travail étudie de façon plutôt formelle la puissance des modèles et de l'opération de filtrage ; on y montre des exemples assez simples d'application dans différents domaines.

On pourrait poursuivre cette étude en calcul algébrique : il serait nécessaire pour cela d'écrire une bibliothèque de sous programmes Snobol qui réalisent la plupart des fonctions du langage spécialisé FØRMAC [25]. Nous avons d'autre part à peine abordé les applications linguistiques ; toutefois, nous sommes convaincus que Snobol doit permettre de traiter par l'informatique des problèmes plus ambitieux que ceux qui le sont actuellement.

En effet, nous avons vu qu'on peut construire des accepteurs et des analyseurs pour des langages engendrés par des grammaires de Chomsky, mais aussi qu'on peut rendre compte de caractères contextuels au moyen d'affectations et de fonctions dans les modèles. L'hébreu, par exemple, semble une langue très intéressante à analyser, le mot peut en effet y contenir un très grand nombre d'informations (langue agglutinante) et son analyse devrait faire intervenir toute la puissance des modèles.

Une autre application de Snobol pourrait être l'aide à la recherche de la syntaxe d'une langue donnée (l'ancien français, par exemple). On pourrait commencer par construire une grammaire approximative et en déduire le modèle correspondant. En analysant le texte, on aurait un certain rebut qu'on tenterait de réduire en modifiant la grammaire et ainsi de suite...

Pour une telle application d'"inférence grammaticale", un compilateur conversationnel s'impose. Un tel compilateur ne doit pas être très difficile à réaliser. En effet, le langage est très dynamique (fonction CØDE) et on se rappelle que la table des symboles existe à l'exécution. D'ailleurs, le système d'entrée-sortie d'un compilateur Snobol non conversationnel étant modifié et un dispositif de récupération du contrôle en cas d'erreur fatale étant trouvé (ØN ERRØR de PL/1), on pourrait très facilement programmer en Snobol un système Snobol conversationnel.

Enfin, en prolongement des méthodes de programmation en Snobol 4, il serait intéressant de donner des méthodes de preuves pour les programmes utilisant des modèles ; dans les cas les plus simples, elles reviendraient à démontrer qu'un langage dont on donne une spécification est bien engendré par une certaine grammaire.

REFERENCES

Sur SNOBOL

[1] R.E. GRISWOLD, J.F. POAGE et I.P. POLONSKY.
The Snobol 4 Programming Language. 2ème édition.
Prentice-Hall.Inc. Englewood Cliffs, N.J. 1971.

[2] R.E. GRISWOLD.
String processing and the Snobol 4 Language.
Int. Summer School on Fundamental Aspects and Current Developments in Computer Science. Copenhague 1969.

[3] R.E. GRISWOLD.
The Macro Implementation of Snobol 4. Freeman San Francisco 1972.

[4] P. WEGNER.
The structure of Snobol 4. Technical Report 68-9,
Dept. of Computer Science, Cornell. Univ. 1968.

[5] R.D. TENNENT
Mathematical Semantics of Snobol 4.
Dept. of Computing and Information Science
Queen's University Kingston Ontario 1974.

[6] Robert B.K. DEWAR
"Spitbol Version 2.0" Snobol 4
Project Document S4 023.
Illinois Institute of Technology, Chicago 1971.

[7] J.F. GIMPEL
"A User Manual for BLOCKS Version 1.4"
SNOBOL 4 Project Document S4D11c Bell Telephone Laboratories,
Inc. Holmdel, New Jersey 1970.

[8] J.F. GIMPEL
A theory of discrete patterns and their implementation in Snobol 4.
CACM 16 2 91-100. 1973.

[9] J. JARAY et H. PISTRE
Représentations particulières des chaînes de caractères, généralisations et
application à l'Implémentation de Snobol 4. Congrès AFCET - SICOB 1971.

[10] D.J. FARBER, R.E. GRISWOLD et J.P. POLONSKY
SNOBOL, a string manipulation language
JACM Vol. 11 n° 1 (1964).

et enfin :

$ [11] J.F. GIMPEL
Catalog of Snobol Publications.
S4D21d. Bell Labs, Holmdel, N.J. 1973.

Autres ouvrages

[11] C. PAIR
Structures de données et algorithmes fondamentaux (à paraître).

[12] C. PAIR
Analyse syntaxique. Cours C.E.A. E.D.F. I.R.I.A. 1973.

[13] S.A. GREIBACH
A New Normal Form Theorem for Context Free Phrase Structure Grammars.
JACM 12, p 42-52, 1965.

[14] M. SINTZOFF
Introduction à la description d'Algol 68. Rev. Française d'Informatique et de Rech. Op. 1969, 3.16.

[15] D.E. KNUTH
The Art of Computer Programming. Addison-Wesley 1968.

[16] M. SINTZOFF
Existence of a Van Wijngaarden syntax for every recursive enumerable set,
Ann. Soc. Sci. de Bruxelles, 81 (1967).

[17] A. VAN WIJNGAARDEN
Report on the algorithmic language ALGOL 68.
Num. Math. 14 29-218 (1969)

[18] D. CROWE
Generating parsers for affix grammars.
Comm. ACM 15, 728-734.

[19] C.H.A. KOSTER
Affix grammars. In : (J.E.L. Peck ed.) Algol 68 implementation.
Amsterdam : North-Holland 1971.

[20] C.H.A. KOSTER
A compiler compiler.
Mathematish Centrum Amsterdam, Report MR 127, 1972.

[21] D. GRIES
Compiler Construction for Digital Computers.
New-York : John Wiley and Sons 1971.

[22] A.V. AHO, J.D. ULLMAN
The theory of parsing, translation and compiling.
Englewood Cliffs (N.J.) : Prentice-Hall 1972.

[23] E.C. RUSSEL
Meta 5 Manual version for XDS Sigma 7. Under batch processing Monitor.

[24] I.B.M. System/360 PL/1 Reference Manual.
Form C28-8201-1 International Business Machine Corporation 1968.

[25] R. TOBEY, J. BAKER, R. CREWS, P. MARKS, K. VICTOR
PL/1 FORMAC
Interpreter user's reference manual 1967.

ANNEXES

PROGRAMME : Développement de la formule

$$X''_{i+2} + X_{i+2} = \sum_{i=0}^{n} X_i \sum_{k=0}^{n-i} X_k \, X_{n-i+k}$$

```
* FABRICATION D'UNE CHAINE DE TRIPLETS D'INDICES.
* ON ORDONNE LES INDICES DANS LES TRIPLETS.
ORIGINE N = LT(N,7) N + 1 :F(END)                                    1
GDBOUC I = -1                                                        2
BO  I = LT(I,N) I + 1 :F(FININD)                                     3
    K = -1                                                           4
B1  K = LT(K,N - I) K + 1 :F(BO)                                     5
    NS = N - (I + K)                                                 6
*TRI
    N1 = LT(I,K) I :F(KI)                                            7
    N2 = K :(SUIT)                                                   8
KI  N1 = K ; N2 = I                                                  9
SUIT TERM = LT(NS,N1) 'X' NS 'X' N1 'X' N2 '+'   :S(FINTRI)          11
     TERM = GT(NS,N2) 'X' N1 'X' N2 'X' NS '+'   :S(FINTRI)          12
     TERM = 'X' N1 'X' NS 'X' N2 '+'                                 13
FINTRI CH = CH TERM :(B1)                                            14
FININD                                                               15
*FONCTION DE COMPTAGE
BOUCLE    IDENT(CH)      :S(FIN)                                     16
          CH BREAK('+') . CHOC '+' = :F(ERREUR)                      17
          NB = 1                                                     18
BCPT      CH CHOC '+' =    :F(NGEN)                                  19
          NB = NB + 1    :(BCPT)                                     20
NGEN      NCH = EQ(NB,1) NCH '+' CHOC   :S(BOUCLE)                   21
          NCH = NCH '+' NB CHOC  :(BOUCLE)                           22
ERREUR    OUTPUT = ' ANOMALIE'   :(ORIGINE)                          23
FIN       NCH '+' =                                                  24
    OUTPUT = 'X'  N + 1 '"+X' N + 1  '=' NCH                         25
          NCH = ''                                                   26
  CH = '' ; TERM = '' ; NB = 0    :(ORIGINE)                         27
END                                                                  30
```

RESULTAT :

$$X2''+X2=3X0X0X1$$
$$X3''+X3=3X0X0X2+3X0X1X1$$
$$X4''+X4=3X0X0X3+6X0X1X2+X1X1X1$$
$$X5''+X5=3X0X0X4+6X0X1X3+3X0X2X2+3X1X1X2$$
$$X6''+X6=3X0X0X5+6X0X1X4+6X0X2X3+3X1X1X3+3X1X2X2$$
$$X7''+X7=3X0X0X6+6X0X1X5+6X0X2X4+3X0X3X3+3X1X1X4+6X1X2X3+X2X2X2$$
$$X8''+X8=3X0X0X7+6X0X1X6+6X0X2X5+6X0X3X4+3X1X1X5+6X1X2X4+3X1X3X3+3X2X2X3$$

PROGRAMME : Dialogue MAITRE - ELEVE.

```
        ANIMAL = FAIL ; VEGETAL = FAIL ; MINERAL = FAIL                    1
       DEFINE('COMPL(X)')  :(FIN.DEF)                                      4
COMPL  CHREGNE = ' ANIMAL | VEGETAL | MINERAL | FAIL '                     5
       &ANCHOR = 0  ; CHREGNE  X ' |' =  ; &ANCHOR = 1                     6
       COMPL = EVAL(CHREGNE)    :(RETURN)                                  9
FIN.DEF                                                                    10
         ALPHA = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'                              11
         BL = SPAN(' ')                                                    12
    ATTR = ('ANIMAL' | 'VEGETAL' | 'MINERAL')                              13
         PREFQUEST = 'EST-CE' BL 'QUE' BL                                  14
   ART = ('LE' | 'LA')                                                     15
   MOT = SPAN(ALPHA)                                                       16
    VERBE = ('EST' BL 'UN')                                                17
   LOCUTION = ART BL MOT . SUJET BL VERBE BL ATTR . CLASSE                 18
****                                                                       19
      &ANCHOR = 1                                                          20
        R =   L'ELEVE :
* ENVOI
 OUTPUT = '                        ***************                 '       21
 OUTPUT = '                        *  DIALOGUE   *                 '       22
 OUTPUT = '                        ***************                 '       23
 OUTPUT = '                                                        '       24
      OUTPUT =   APRES TOUT CE QUE J'AI FAIT POUR TOI DISAIT LE MAITRE     25
     OUTPUT =   JE SAIS TU ME L'A DEJA DIT DISAIT L'ELEVE                  26
     OUTPUT =   JE NE T'EN DEMANDAIS PAS TANT                              27
 OUTPUT = '************************************************************'   28
 OUTPUT = ' '                                                              29
LECTURE   CH  = TRIM(INPUT)      :F(FIN)                                   30
          OUTPUT = ' LE MAITRE :  ' CH                                     31
ANALYSE   CH PREFQUEST LOCUTION RPOS(0) :S(REPONSE)                        32
          CH LOCUTION RPOS(0)  :S(APPREND)                                 33
SYNTAXE    OUTPUT = ' ERREUR DE SYNTAXE '       :(LECTURE)                 34
****
****
VOIRRESTE SUJET   COMPL(CLASSE)  :F(SAISPAS)                               35
          OUTPUT = R ' NON IL APPARTIENT A UN AUTRE REGNE'   :(LECTURE)     36
REPONSE   SUJET  $CLASSE RPOS(0) :F(VOIRRESTE)                             37
          OUTPUT = R   OUI C'EST VRAI      :(LECTURE)                      38
SAISPAS OUTPUT = R   VOUS NE ME L'AVEZ PAS APPRIS          :(LECTURE)      39
****
****
APPREND   SUJET $CLASSE  RPOS(0)  :S(JESAIS)                               40
          $CLASSE  =  $CLASSE  | SUJET                                     41
          OUTPUT = R ' JE NOTE QUE  ' CH   :(LECTURE)                      42
JESAIS    OUTPUT  = R ' JE LE  SAVAIS DEJA ' :(LECTURE)                    43
****
FIN       OUTPUT =   JE SUIS UN BON ELEVE N'EST-CE PAS                     44
          OUTPUT =   POURTANT J'AI MAUVAISE MEMOIRE                        45
          OUTPUT =   LA PROCHAINE FOIS J'AURAI TOUT OUBLIE                 46
          OUTPUT = ' AU REVOIR ET MERCI'                                   47
END                                                                        48
```

RESULTAT :

```
                ***************
                *  DIALOGUE   *
                ***************

APRES TOUT CE QUE J'AI FAIT POUR TOI DISAIT LE MAITRE
JE SAIS TU ME L'A DEJA DIT DISAIT L'ELEVE
JE NE T'EN DEMANDAIS PAS TANT
**********************************************************

LE MAITRE :  EST-CE QUE LE CHAT EST UN ANIMAL
L'ELEVE :  VOUS NE ME L'AVEZ PAS APPRIS
LE MAITRE :  LE CHAT EST UN ANIMAL
L'ELEVE :  JE NOTE QUE  LE CHAT EST UN ANIMAL
LE MAITRE :  LE CHAT EST UN ANIMAL
L'ELEVE :  JE LE  SAVAIS DEJA
LE MAITRE :  EST-CE QUE LE CHAT EST UN ANIMAL
L'ELEVE :  OUI C'EST VRAI
LE MAITRE :  LE CHAT EST MUET
ERREUR DE SYNTAXE
LE MAITRE :  LE PORTUGAL EST UN MINERAL
L'ELEVE :  JE NOTE QUE  LE PORTUGAL EST UN MINERAL
LE MAITRE :  LA CALIFORNIE EST UN MINERAL
L'ELEVE :  JE NOTE QUE  LA CALIFORNIE EST UN MINERAL
LE MAITRE :  LE RUBIS EST UN MINERAL
L'ELEVE :  JE NOTE QUE  LE RUBIS EST UN MINERAL
LE MAITRE :  EST-CE QUE LE CHAT EST UN VEGETAL
L'ELEVE :  NON IL APPARTIENT A UN AUTRE REGNE
LE MAITRE :  EST-CE QUE LE CHAT EST UN MINERAL
L'ELEVE :  NON IL APPARTIENT A UN AUTRE REGNE
LE MAITRE :  EST-CE QUE LE RUBIS EST UN VEGETAL
L'ELEVE :  NON IL APPARTIENT A UN AUTRE REGNE
LE MAITRE :  LA CIGUE EST UN VEGETAL
L'ELEVE :  JE NOTE QUE  LA CIGUE EST UN VEGETAL
LE MAITRE :  EST-CE QUE LE CHIEN EST UN SENTIMAL
ERREUR DE SYNTAXE
LE MAITRE :  LE CHIEN EST UN ANIMAL
L'ELEVE :  JE NOTE QUE  LE CHIEN EST UN ANIMAL
LE MAITRE :  LA VACHE EST UN ANIMAL
L'ELEVE :  JE NOTE QUE  LA VACHE EST UN ANIMAL
LE MAITRE :  EST-CE QUE LA VACHE EST UN ANIMAL
L'ELEVE :  OUI C'EST VRAI
LE MAITRE :  EST-CE QUE LE CHIEN EST UN ANIMAL
L'ELEVE :  OUI C'EST VRAI
JE SUIS UN BON ELEVE N'EST-CE PAS
POURTANT J'AI MAUVAISE MEMOIRE
LA PROCHAINE FOIS J'AURAI TOUT OUBLIE
AU REVOIR ET MERCI
```

NOM DE L'ETUDIANT : JARAY Jacques

NATURE DE LA THESE : Spécialité en Mathématiques Appliquées

VU, APPROUVE

& PERMIS D'IMPRIMER

NANCY, le 24 Avril 1975

LE PRESIDENT DE L'UNIVERSITE DE NANCY I

UNIVERSITÉ DE NANCY I
Le Président
J.R. HEZLUT