UNIVERSITE DE NANCY I

U.E.R. DE MATHEMATIQUES

outils d'aide à la mise au point d'un compilateur écrit dans le langage face

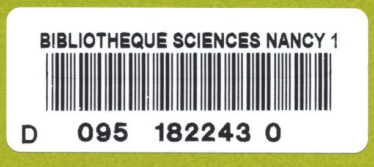
thèse

pour l'obtention du doctorat de spécialité mathématiques appliquées (informatique)

présentée le 22 juin 1976

par

anne_marie rasser_bouchet



jury

PRESIDENT C. PAIR

EXAMINATEURS F. BELLEGARDE

J.C. DERNIAME

M. GRIFFITHS

UNIVERSITE DE NANCY I
U.E.R. DE MATHEMATIQUES

outils d'aide à la mise au point d'un compilateur écrit dans le langage face

thèse

pour l'obtention du doctorat
de spécialité mathématiques appliquées
(informatique)

présentée le 22 juin 1976

par

anne_marie rasser_bouchet

jury

PRESIDENT C. PAIR

EXAMINATEURS F. BELLEGARDE
J.C. DERNIAME
M. GRIFFITHS

Je tiens tout d'abord à remercier Monsieur le Professeur C. PAIR, Président de l'Institut National Polytechnique, qui a inspiré puis dirigé ce travail et me fait l'honneur de présider ce Jury. Je lui exprime toute ma reconnaissance pour la formation qu'il m'a donnée au cours de son enseignement.

Je remercie Monsieur le Professeur M. GRIFFITHS qui s'est intéressé à cette recherche et a accepté d'être membre de ce Jury.

Mes remerciements vont également à Monsieur J.C. DERNIAME qui m'a initiée à l'informatique et a bien voulu juger ce travail.

Je voudrais exprimer toute ma reconnaissance à Madame F. BELLEGARDE qui n'a cessé de suivre et conseiller mes travaux de recherche en une amicale collaboration. Elle a assisté à la création du projet FACE et l'anime toujours avec ferveur.

Au sein de l'équipe FACE, je remercie également Messieurs J. MAROLDT et H. PISTRE pour leurs travaux d'implémentation de ce langage.

Je tiens aussi à exprimer mes remerciements à Madame L. BEAURAIN pour sa frappe diligente et Mademoiselle F. LE MARECHAL pour son amicale aide matérielle.

SOMMAIRE

INTRODUCTION

L'écriture d'un compilateur représente un travail important à la fois au niveau de la conception, de l'écriture et de la mise au point. Ce travail peut être allégé en utilisant un langage d'écriture de compilateurs [6] [2] [8].

Face fait partie des langages d'écriture de compilateurs, dits "dirigés par la syntaxe" : ils permettent d'écrire des compilateurs définis par une grammaire algébrique,et la génération du texte-objet est obtenue par un certain nombre d'actions associées à la grammaire et activées par l'analyse syntaxique du texte source. Ainsi le programmeur n'a en général pas à décrire l'analyse syntaxique qui est effectuée automatiquement ; pour la plupart des langages d'écriture de compilateurs, l'algorithme d'analyse est fixe.

Le premier objectif du langage Face était de libérer complètement l'utilisateur de la méthode d'analyse syntaxique [3] [15], celle-ci n'ayant aucune influence sur la description de la génération du texte-objet. L'algorithme d'analyse peut même être choisi automatiquement en fonction du langage étudié [15]. La forme du résultat de l'analyseur est choisie de manière à faciliter la génération du texte-objet, et celle-ci est indépendante de l'algorithme d'analyse.

Un autre objectif de Face est de permettre une description aisée de la génération du texte-objet par l'intermédiaire de procédures associées aux règles de grammaire ; l'ordre d'appel des procédures, dépendant du résultat de l'analyse syntaxique, n'est pas spécifié par l'utilisateur, il est effectué automatiquement [3].

La plupart des langages d'écriture de compilateurs dirigés par la syntaxe ne permettent de décrire qu'un passage

à la fois du compilateur. Le langage Face permet à l'utilisateur de décrire des compilations en un ou plusieurs passages, séparés ou non de l'analyse syntaxique, sans être tenu pour chacun d'eux à refaire cette analyse et même en pouvant utiliser des grammaires différentes pour chacun d'eux [3]. Par exemple, une analyse morphologique et lexicographique peut être guidée par une grammaire beaucoup plus pauvre que celle guidant les autres passages.

Enfin le langage Face fournit des outils spécifiques à l'utilisateur : il peut disposer de tables, traiter de listes si nécessaires [3].

L'écriture d'un compilateur Algol 60 et la mise au point du compilateur Face, tous deux écrits en Face, nous permet d'affirmer que la conception et la programmation en Face d'un compilateur se font de façon modulaire et sont agréables pour le programmeur. Mais la mise au point reste un travail fastidieux : si Face, comme langage d'écriture de compilateur, facilite leur écriture, il n'en facilite pas la mise au point : l'utilisateur perd le contrôle sur ce qui se passe (cf. § 1-1-2),ce qui rend difficile une mise au point modulaire.

Nous avons cherché à rendre cette tâche plus facile : le premier objectif est de ne pas attendre la construction complète du compilateur pour le mettre au point. Le programmeur doit pouvoir tester des parties du compilateur au fur et à mesure qu'il les conçoit, ce qui réduit considérablement les tests au niveau du compilateur entier.

Le deuxième objectif est d'automatiser la production des jeux d'essai nécessaires à cette mise au point. L'idée est d'utiliser la grammaire pour guider la génération d'un ensemble de jeux d'essai. Ceci doit permettre une plus grande sécurité en fournissant un ensemble de tests plus complet. C'est l'objet de ce travail.

Dans le chapitre I, après des rappels sur le langage Face, nous montrons comment l'utilisateur de Face peut concevoir un compilateur et l'écrire de façon modulaire, nous en déduisons les tests de mise au point qu'il doit faire. Dans la suite de ce travail, nous apportons des outils supplémentaires pour la mise au point d'un compilateur en Face, en particulier pour la mise au point de la grammaire (chapitres I et IV), pour la mise au point des parties du compilateur (chapitre III), ces parties appelées '<u>paragraphe</u>' seront définies aux chapitres I et II, et pour la génération des jeux d'essai (chapitre IV).

CHAPITRE 1

PROBLEMES POSES PAR LA MISE AU POINT D'UN COMPILATEUR ECRIT EN FACE

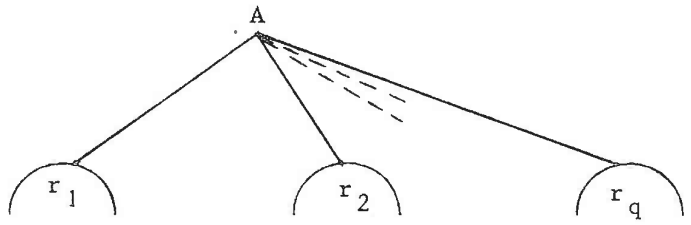
I-1. RAPPELS DE FACE

1-1-1. Processus de compilation en Face

Face s'appliquant à des langages dont la syntaxe est définie par une grammaire algébrique, un programme écrit en Face est essentiellement constitué d'une ou plusieurs grammaires complétées d'une description de la traduction du langage, grâce à des procédures attachées aux règles de grammaire.

L'analyse syntaxique transforme le texte source en une ramification à une racine [11] [12], dont les noeuds sont étiquetés par les numéros de règles de la grammaire pour laquelle est faite l'analyse.

Justifions comment Face permet la traduction du langage source en utilisant ce résultat. Soit r la ramification résultat de l'analyse syntaxique. La compilation du texte source peut être définie par une procédure comp [3], qui admet la ramification r en paramètre. La ramification r a une racine A et un certain nombre de ramifications composantes r_1, r_2, ... , r_q



On peut écrire $r = A \times (r_1 + r_2 + \ldots + r_q)$ [11] [12].

On peut alors définir la procédure comp récursivement par

$$comp(r) = F(comp(r_1), comp(r_2), \ldots, comp(r_q))$$

où F est une procédure qui utilise les résultats de $comp(r_1)$, $comp(r_2)$, ..., $comp(r_q)$, et qui est associée à la règle de grammaire qui a engendré cette partie de la ramification.

Prenons comme exemple la grammaire d'axiome E :

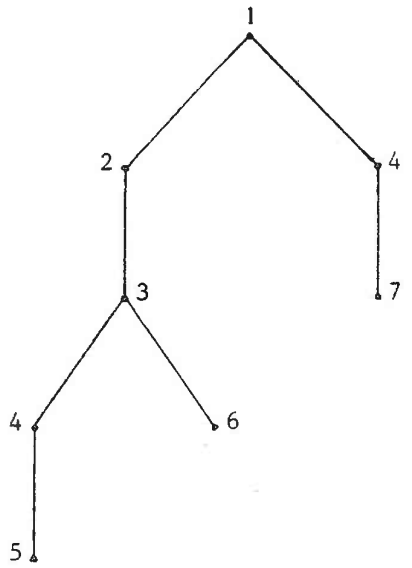
$$<E> ::= <E> + <T> \,|^{1}\, <T> \,|^{2}$$

$$<T> ::= <T> * <F> \,|^{3}\, <F> \,|^{4}$$

$$<F> ::= a \,|^{5}\, b \,|^{6}\, c \,|^{7}\, (<E>) \,|^{8}$$

où 1, 2, ..., 8 sont les numéros de règles.

La phrase a * b + c appartient au langage engendré par la grammaire, son analyse syntaxique fournit la ramification :



Cette ramification peut se décrire par composition des sous-ramifications qu'elle contient :

$$r = 1 \times ((2 \times 3 \times (4 \times 5 + 6)) + (4 \times 7))$$

Si on appelle F_i, la procédure associée à la règle i :

$$comp(r) = F_1\ (F_2(F_3(F_4(F_5), F_6)), F_4(F_7)).$$

Les procédures F_5, F_6 et F_7 associées aux règles conduisant aux terminaux n'ont pas d'arguments provenant d'autres procédures F_i, elles fournissent des résultats (pour l'exemple ci-dessus, ce pourraient être les adresses et les types des différentes variables a, b, c).

La procédure F_4 utilise comme argument les résultats de F_5 dans son premier appel, et ceux de F_7 dans le second.

La procédure F_3 utilise les résultats de F_4 et de F_6, et ainsi de suite, si bien que l'expression comp(r) peut être calculée par une suite d'appels d'instructions procédures :

$$G_5 \; ; \; G_4 \; ; \; G_6 \; ; \; G_3 \; ; \; G_2 \; ; \; G_7 \; ; \; G_4 \; ; \; G_1$$

où chaque procédure G_i exécute F_i en prenant ses arguments sur une pile et y replace ses résultats : à la fin, la pile contient le résultat de comp(r).

L'ordre d'appel de ces procédures G_i, correspond à la représentation postfixée de la ramification fournie par l'analyse syntaxique. Soit pour l'exemple : 5, 4, 6, 3, 2, 7, 4, 1.

Le langage Face utilise ce principe, mais il reste transparent à l'utilisateur. En effet Face offre une procédure standard APPL qui réalise le programme de commande du compilateur, c'est-à-dire l'appel des différentes procédures regroupées en ce que Face appelle les rangées de procédures, et ceci dans l'ordre spécifié par une rangée d'entiers qui sont les indices successifs des procédures de la rangée. Cette rangée d'entiers correspond à la représentation postfixée de la ramification qui est le résultat de l'analyse syntaxique, ceci est réalisé par une autre procédure standard ANALYSE. Les procédures standard analyse et appl peuvent être combinées en une autre procédure standard ANAPPLYSE qui effectue l'appel aux procédures au fur et à mesure de la construction du résultat d'analyse (tout au moins quand l'analyse est déterministe).

1-1-2. Nature d'un programme Face

Rappelons la structure d'un programme Face, qui décrit un compilateur, car nous serons souvent amenés à donner des exemples de programmes Face ou de 'parties' de programmes Face.

Un programme Face est essentiellement composé de 3 parties :

- La première partie comprend la description d'une ou plusieurs grammaires du texte source nécessaires aux procédures analyse ou anapplyse.

- Cette description des grammaires est suivie d'une partie de programme liste de déclarations des objets globaux du compilateur, ce peuvent être des éléments (entiers ou chaînes de caractères), des repères qui sont des pointeurs vers ces éléments, ou des regroupements d'éléments ou repères dans des tableaux à une dimension qui sont les rangées d'éléments ou de repères. On peut aussi y déclarer des procédures auxiliaires servant à l'ensemble du compilateur. C'est aussi dans cette deuxième partie que le programmeur déclare la ou les rangées de procédures, arguments des procédures standard appl ou anapplyse.

- Enfin le compilateur est défini par une expression fermée qui termine le programme.

Nous verrons mieux, sur un exemple, ces différentes parties. L'exemple est tiré du compilateur Algol 60, où l'on traite de la structure de bloc, des déclarations de type et de l'expression conditionnelle. La compilation se fait en deux passages par appels successifs des procédures anapplyse et appl.

Au premier passage sera associée la rangée de procédures lex, au second la rangée de procédures gen.

Programme Face	Commentaires
BLOC : DEBUT,PARTIE DEC,FIN DE BLOC|5 ;	Partie de la grammaire Algol 60, comprenant les blocs et les déclarations de type, et l'expression conditionnelle
DEBUT : 'DEBUT'|6 ;	
PARTIE DEC : DEC||PARTIE DEC, ',' , DEC| ;	
DEC : DECTYPE, ',' ,IDEC|8|TYPE,IDEC|8 ;	
. . .	
EXPR : EXPR SIMPLE||PARTIE AVANT SINON, 'SINON',EXPR|86 ;	
PARTIE AVANT SINON : PROPOSITION SI, EXPR SIMPLE|85 ;	
PROPOSITION SI : 'SI',EXPR,'ALORS'|84 ;	
. . .	
REP ML:=1 ; NO:=0 ; NB:=0	Déclarations de variables globales avec leur initialisation ML : pointeur vers la zone locale NO : numéro du bloc en cours NB : nombre de blocs
. . .	
RANGEE(3) ELT RX = (1,2,3) ;	RX est un tableau qui contient les numéros de registres
. . .	
RANGEE(10) REP AD ; RANGEE(1000) REP CHAINE ;	AD : tableau qui contient l'adresse en zone locale de chacun des blocs CHAINE : résultat de la procédure analyse
. . .	

```
RANGEE(120) PROC LEX ;
LEX[5] = (2,0) (AD[NO]:= ML+1 ; ML:=P1 ;
         NO:=P2) ;
LEX[6] = (0,2) (P1:=ML ; P2:=NO ;
         NB:=NB+1 ; NO:=NB) ;
```

Déclaration de la rangée de procédures LEX.
LEX 5 correspond à la fin d'un bloc au premier passage : 2 arguments à prendre en pile, pas de résultats.
LEX 6 correspond au début d'un bloc, elle n'a pas d'argument mais donne 2 résultats, désignés par P1, P2 dans la procédure.

```
RANGEE(120) PROC GEN ;
GEN[5] = (1,0) (COURS[NO]:=0 ; NØ:= P1) ;
```

Déclaration de la rangée de procédures GEN.

```
PROC ELT TEST REG = (ELT ADR)
DECOMPACTER(1 ; ADR) = 'E' : 1 , 0 ;
```

Déclaration d'une procédure auxiliaire.
TEST REG est une procédure booléenne indiquant si l'adresse est un registre ou non. Elle utilise une procédure standard DECOMPACTER.

```
PROC TYPE = (ELT T)
T = 'R' ∨ T = 'E' : ( ) , ERREUR(3) ;
```

Procédure qui vérifie si T est réel ou entier, sinon positionne le code d'erreur à 3 grâce à la procédure ERREUR.

```
(ANAPPLYSE(1 ; CHAINE ; LEX) ;
APPL(CHAINE ; GEN))
```

Expression fermée finale, qui appelle l'analyse du programme source et l'exécution de la rangée LEX sur la chaîne obtenue. Puis l'appel toujours sur la même chaîne mais pour la rangée de procédures GEN.

Si nous voulons mettre au point ce compilateur C, du langage L, écrit en Face, il va y avoir essentiellement deux étapes :

a) la compilation du programme C : le compilateur Face génère un programme assembleur M, équivalent à C.

b) l'exécution de M avec des programmes de test P_i, en langage L, pour données.

Toute erreur du compilateur C n'est trouvée que par une erreur décelée dans un texte généré G_i ; ces erreurs remettent en question l'écriture initiale de C en Face et il est difficile de trouver de quelle procédure d'une rangée elles proviennent sans repenser à la conception globale du compilateur. Dans l'exemple ci-dessus, une mauvaise adresse générée peut aussi bien provenir d'une procédure auxiliaire de calcul d'adresse (par exemple TESTREG), ou d'une mauvaise gestion de bloc (par exemple dans LEX[5] ou GEN[5]), ou d'une mauvaise incrémentation d'un compteur d'adresses globales à l'intérieur d'une procédure de rangée (par exemple ML ou NB), ou même d'une ramification erronée, provenant d'une grammaire mal écrite (par exemple génération des expressions arithmétiques générées à l'envers).

Il est difficile de faire une mise au point modulaire du compilateur C, il faut repenser globalement au compilateur en cas d'erreur, puis le recompiler totalement.

I-2. CONCEPTION ET ECRITURE MODULAIRES D'UN COMPILATEUR EN FACE

Etant donné un langage source LS à compiler, l'utilisateur peut commencer par écrire une grammaire de ce langage, ce qui lui donne la première partie de son programme Face et une vue d'ensemble du langage source.

Il vient d'écrire la partie purement syntaxique de son compilateur, il cherche maintenant à modulariser le travail de traduction, le découpage est fait selon deux directions : définir la traduction en plusieurs passages sur le texte source, puis modulariser le travail de chacun des passages.

1-2-1. Conception modulaire des passages du compilateur

En fonction du langage source LS et de l'implémentation (place disponible en mémoire, langage objet...) l'utilisateur peut déterminer le nombre de passages nécessaires à la compilation.

Par exemple, pour un compilateur Algol 60, si on dispose de suffisamment de place, on choisira plus facilement deux passages : le premier passage permet

i) de construire une table des symboles où l'identificateur est associé au numéro du bloc où il est défini dans le programme source.

ii) de réaliser l'allocation statique de mémoire correspondant aux objets déclarés pour ces divers blocs.

Le deuxième passage permet [9]

i) d'effectuer la génération proprement dite du texte objet.

ii) de réaliser l'allocation statique d'un bloc et de prévoir la gestion dynamique des blocs et des appels de procédures lors de l'exécution.

Par contre, si l'on implémente ce même compilateur sur une machine disposant de très peu de place en mémoire, on aura intérêt à multiplier le nombre de passages.

Chaque passage est défini par ses résultats. Dans l'exemple précédent les résultats du premier passage sont une table des symboles et une rangée de repères ALLOC où ALLOC[i] est la taille de la zone statique du bloc de numéro i ; le résultat du deuxième passage étant le programme objet généré.

Pour chaque passage, il s'agit de définir les résultats à partir d'un langage source LS ; ce dernier est décrit par une grammaire et celle-ci peut être différente à chaque passage et en particulier différente de la grammaire de LS, écrite au départ par le programmeur.

Dans l'exemple le premier passage ne s'intéresse qu'aux règles de la grammaire qui définissent les déclarations, les débuts et fins de blocs ; la grammaire de ce premier passage peut être plus différente que celle guidant le deuxième passage, qui s'intéresse à tout le texte source.

La définition des résultats de chacun des passages permet à l'utilisateur de trouver une grammaire nécessaire à chacun d'eux et de voir en fonction de l'enchaînement des passages s'il doit recommencer l'analyse syntaxique du texte source ou s'il peut utiliser le résultat de l'analyse du passage précédent.

Dans le premier cas, s'il a une grammaire différente de celle du passage précédent, il doit appeler la procédure anapplyse pour définir le passage et dans le second cas, s'il peut garder la même grammaire pour deux passages consécutifs, il lui suffit d'appeler appl (cf. ch. I-1-1).

Si nous reprenons le compilateur Algol 60 à définir en deux passages et si nous préférons ne pas recommencer l'analyse syntaxique du texte source au deuxième passage, une même grammaire sert de guide aux deux passages ; le programme Face ne comptera qu'une description de grammaire.

Si nous appelons lex la rangée de procédures correspondant au premier passage et gen la rangée de procédures correspondant au deuxième passage, l'expression finale est :

(anapplyse(1;chaîne;lex) ; appl(chaîne;gen)).

Ayant défini le nombre voulu de passages, leurs résultats et leur enchaînement, en particulier si on recommence une analyse syntaxique entre chacun, le programmeur peut écrire l'expression fermée finale du compilateur.

1-2-2. Conception modulaire au niveau des règles de chacun des passages

Etant donnés la grammaire et les résultats voulus pour un passage, il reste à l'utilisateur à écrire la rangée de procédures correspondant à chacun des passages, c'est-à-dire une procédure de cette rangée par règle de grammaire.

Si le programmeur écrit le deuxième passage d'un compilateur Algol 60 qui doit générer le texte objet et s'il considère les deux règles :

<Expression simple>::=<terme>|<expression simple>+<terme>| [20]

Il sait que pour la première règle il n'a rien à générer, pour cette règle il n'y aura pas de procédure dans la rangée gen, il n'associe aucun numéro à cette règle.

Par contre pour la deuxième il doit effectuer la génération d'une addition ; s'il associe le numéro 20 à la règle, la procédure gen[20] utilise des arguments qui sont des résultats provenant des composantes de racine 'expression simple' et 'terme', par exemple les renseignements : type, adresse, booléen indiquant la nature du calcul généré, résultat intermédiaire ou non.

Ainsi gen[20] aura six arguments qui sont dans l'ordre : type, adresse, marque de résultat intermédiaire pour l'opérande gauche, type, adresse et marque de résultat intermédiaire pour l'opérande droit. Elle doit fournir trois résultats qui sont les type, adresse, marque de résultat intermédiaire du résultat de l'addition lors de l'exécution du texte objet.

La procédure génère l'addition, après avoir généré éventuellement des conversions de type sur les opérandes. Ceci est l'apport de cette procédure au résultat global du passage décrit par la rangée de procédures gen et en fait un effet annexe de la procédure.

Proposons une procédure pour gen[20], qui génère un langage intermédiaire à deux adresses :

Exemple	Commentaires
GEN[20]=(6,3)(REP TYPE,ADRESSE;	G20 choisit un type compatible en fonction des opérandes, effectue les conversions et génère les opérations. Replace le résultat en pile.
TYPE:=TESTTYPE(P1;P4);	TESTTYPE : procédure qui choisit le type du résultat en fonction des types des opérandes.
P1≠TYPE:CONVERSION(P1;P2;P3), P2≠TYPE:CONVERSION(P4;P5;P6),();	CONVERSION : procédure qui effectue la conversion et remet les résultats en P1, P2 et P3.
ADRESSE:=RECHADR(P2;P3;P5;P6)	RECHADR : procédure qui cherche une adresse pour le résultat en fonction des adresses des opérandes.
TYPE='REEL':GENERER(ADR;P2; P3;P4;P5;ADRESSE), GENERER(ADE;P2;P3;P4;P5; ADRESSE); P1:=TYPE; P2:=ADRESSE; P3:=1)	GENERER : procédure qui génère un ordre du langage intermédiaire. On positionne la marque de résultat intermédiaire (P3).

Une procédure de la rangée est donc un module pour la conception du passage.

Remarquons que c'est lors de l'écriture d'une procédure de rangée, que l'on sent le besoin de créer des procédures auxiliaires, soit parce que les procédures servent à plusieurs procédures de rangée, soit pour alléger l'écriture d'une procédure.

Dans l'exemple ci-dessus nous avons rencontré les procédures auxiliaires conversion, testtype, rechadr qui serviront à toutes les procédures de rangée s'occupant d'une expression arithmétique, et la procédure générer qui servira à tout le passage de génération.

I-3. TESTS MODULAIRES D'UN COMPILATEUR ECRIT EN FACE

Nous venons de voir que le langage Face est bien adapté à une définition et conception modulaire de l'écriture d'un compilateur, si bien que l'écriture en est relativement rapide. Cependant il reste à mettre au point le compilateur écrit, ce qui est la phase la plus lourde et la plus rébarbative, et nous a amenés à nous poser la question suivante : comment modulariser les tests de mise au point d'un compilateur écrit en Face ?

Le programmeur doit tester que ce qu'il vient d'écrire définit bien les résultats voulus et que le programme objet final obtenu définit bien la même fonction que le programme source (cf. § 1-4-2 iii et § 2-3-1). Il doit tester que la ou les grammaires écrites engendrent bien le langage désiré et que les ramifications engendrées par la grammaire, conviennent pour les appels des procédures de rangée (cf. 1-3-1). Il doit aussi tester que chaque rangée de procédures définit les résultats pour le passage qu'elle décrit (cf. § 1-3-2) et ceci ne se réalise que si chaque procédure de la rangée définit les résultats voulus à partir de ses arguments (cf. 1-3-3) et des procédures simples qu'elle utilise (cf. 1-3-4).

1-3-1. Test de la grammaire

Pour tester la grammaire le programmeur effectue des tests de mise au point pour vérifier que le langage source LS est égal au langage L engendré par la grammaire qu'il écrit, ce qui se décompose en deux sortes de tests :

1) Vérifier que le langage source LS est inclus dans L (cf. § 1-3-1-2).
2) Vérifier que L, langage engendré par la grammaire, est inclus dans LS (cf. § 1-3-1-3).

Il est évident que la grammaire doit être acceptée par l'analyseur (cf. § 1-3-1-1).

Ces tests de mise au point ne démontrent pas que L soit égal à LS, ils sont une vérification pour le programmeur ; toute démonstration serait impossible pour deux raisons :

- on n'a pas en général de grammaire définissant le langage source souhaité LS ;
- dans le cas où LS est défini par une grammaire à contexte libre et l'équivalence de cette grammaire et de la grammaire écrite par le programmeur est indécidable [9].

1-3-1-1. Une grammaire écrite dans le programme Face doit permettre l'analyse syntaxique d'un certain langage source LS. Elle doit être acceptée par un des algorithmes d'analyse disponibles [15].

La partie préanalyseur du système Face vérifie que la description de la grammaire donnée est syntaxiquement correcte et qu'elle peut faire l'objet d'une analyse et au moyen d'un des algorithmes disponibles. Il choisit parmi eux un algorithme d'analyse syntaxique adapté à la grammaire et transforme si besoin est la représentation de la grammaire en vue de l'analyse choisie.

Exemple :
Si dans une grammaire du langage Face, nous trouvons les deux règles :

<expression fermée>::=<souvrir><premier repère>
<fin decloc de repère><liste de déclarations locales>
<point virgule><fin d'exp.fermée>|[116]

<liste de déclarations locales>::=<vide>|
<liste de déclarations locales><point virgule>
<déclaration locale de repères>|

Elles ne satisfont pas aux conditions de déterminisme [15] pour un algorithme descendant avec lecture d'un caractère à l'avance, appartenant aux algorithmes d'analyse syntaxique du langage Face : l'algorithme ne sait pas choisir de façon déterministe à la lecture du point virgule, s'il existe encore une déclaration locale de repères dans la règle LISTE DE DECLARATIONS LOCALES ou s'il doit revenir dans la règle EXPRESSION FERMEE.

Le préanalyseur ne choisira pas cet algorithme pour cette grammaire et si l'on ne possède pas d'autre algorithme dans le système, la grammaire sera refusée.

Pour tester cela il suffit d'écrire un programme Face avec la grammaire seule et une expression fermée finale vide (cf. 1-1-2). Le compilateur Face appelle le préanalyseur, pour vérifier que la grammaire est bien acceptée [15] par un des algorithmes d'analyse disponibles. Ce dernier imprimera la cause du refus et le numéro de la ligne source où se trouvent la règle et le numéro de cette règle, s'il existe.

Le préanalyseur ou le compilateur Face aurait pu aussi vérifier que la grammaire est bien écrite, par exemple

qu'elle est bien réduite inférieurement, c'est-à-dire que tout non terminal engendre bien au moins un mot du langage ; mais il est rare qu'un utilisateur écrive des grammaires non réduites inférieurement, sauf par oubli de règles, ce qui est testé par le compilateur Face : en effet il vérifie que tout non terminal cité en deuxième membre de règle est bien premier membre d'une règle de grammaire.

1-3-1-2. Bien que la grammaire soit en elle-même une définition de la syntaxe du langage source LS étudié, le programmeur peut faire des erreurs et écrire une grammaire qui n'engendre pas exactement le langage voulu, et ceci soit en transcrivant la syntaxe de ce langage en une grammaire Face, soit en modifiant la grammaire en vue d'un passage particulier ou même par inattention. Le programmeur veut vérifier partiellement que le langage source qu'il veut décrire est bien inclus dans le langage L engendré par la grammaire qu'il donne (LS ⊂ L) et ceci tout simplement en donnant à analyser des "phrases-types" de son langage source.

Il lui suffit pour cela d'écrire un programme Face réduit à la grammaire à tester suivie d'une expression fermée contenant un ou plusieurs appels à la procédure standard ANALYSE pour ces "phrases-types".

Exemple :

Programme	Commentaires
GRAM	
programme:étiquette,bloc|bloc	Description de la grammaire à tester.
⋮	
RANGEE(2000)REP RAMIF ;	Déclaration du tableau recevant le résultat de l'analyse, c'est-à-dire la représentation postfixée d'une ramification.
(ANALYSE(1;RAMIF));	Expression fermée contenant l'appel à l'analyseur.

De plus l'utilisateur peut vérifier que la ramification est correcte, c'est-à-dire que les actions sont engendrées dans l'ordre voulu, si on lui propose une procédure permettant d'écrire cette ramification, procédure standard qui écrit une rangée de repère (cf. annexe O).

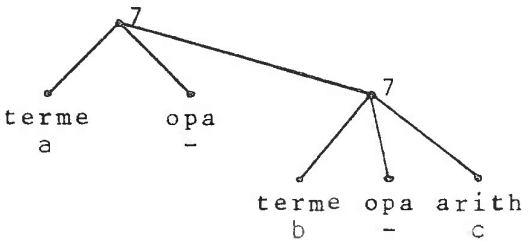
Par exemple dans un compilateur Algol 60, pour les règles de grammaire suivantes, concernant les expressions arithmétiques, il peut être utile de constater si les opérations sont bien générées de gauche à droite et suivant la priorité voulue :

<facteur>::=<primaire>|[1]<primaire><puissance><facteur>|[2]

<terme>::=<facteur>|[3]<facteur><op m ><terme>|[4]

<arith>::=<terme>|[5]<opa><terme>|[6]<terme><opa><arith>|[7]

A la "phrase-type" a - b - c correspondrait la ramification résultat :



L'utilisateur s'apercevrait alors que les opérations sont générées de droite à gauche, il changerait sa grammaire en transformant la récursivité droite en récursivité gauche.

Ce test des "phrases-types" appartenant, ou non, au langage source LS voulu ne suffit pas ; en effet considérons l'exemple de deux règles d'une grammaire du langage Face :

<suite de règles>::=<règle>;<suite de règles>|<règle>

<règle>::=<non terminal>:<second membre>;|

Ces règles vont engendrer deux points-virgules entre chaque règle de la grammaire Face, au lieu d'un seul voulu. Une erreur est détectée par l'analyseur car celui-ci refusera des phrases correctes (avec un seul point-virgule) ; par exemple pour une phrase :

non terminal 1 : second membre 1 | ;
non terminal 2 : second membre 2 | ;

l'analyseur indiquerait : "non terminal 2 incorrect, terminal attendu".

Ce n'est pas la vraie erreur qui est détectée et l'utilisateur a du mal à retrouver quelle est l'erreur de la grammaire, surtout dans un cas encore plus complexe.

1-3-1-3. Considérons maintenant l'exemple d'une grammaire ayant simplement une règle :

<élément de liste pour>::=<expression>tant que<expression>|...

Le programmeur s'est trompé : <expression> derrière le 'tant que' doit être une expression simple de type booléen ; en analysant des "phrases-types" justes il ne verra pas l'erreur et a peu de chance de la retrouver.

C'est pourquoi il a semblé nécessaire de permettre à l'utilisateur de vérifier que le langage engendré par la grammaire donnée n'engendre que des phrases du langage source voulu ($L \subset LS$). Pour réaliser cette génération de phrases engendrées par une grammaire donnée, nous avons écrit une procédure standard de mise au point Face, GENPHRASE qui sera l'un des objets du chapitre IV. Cette procédure fournit aussi les ramifications correspondantes engendrées par la grammaire donnée.

Ainsi le préanalyseur, l'analyseur et le générateur de phrases, réalisés par les procédures standard Face PREAN et ANALYSE et par la procédure standard de mise au point Face GENPHRASE, permettent à l'utilisateur de réaliser les vérifications possibles des grammaires qu'il écrit en Face.

1-3-2. Test des différents passages

Il est nécessaire que l'utilisateur mette au point séparément chacun des passages, surtout s'ils sont nombreux.

Pour tester le premier passage du compilateur, il suffit d'écrire un appel de appl sur la rangée de procédures qui le décrit, l'appel de APPL doit être précédé ou combiné avec un appel d'ANALYSE qui fournit une ramification sur une phrase, jeu d'essai du langage source.

Par exemple, si nous testons le premier passage du compilateur Algol 60 qui est associé à la rangée LEX, le programme Face obtenu serait :

```
GRAM
programme:bloc|]étiquetage,bloc|
bloc :
        .
        .
        .
REP ML,NO:=1 ;
RANGEE(15) REP ALLOC;
RANGEE(1000)REP CHAINE;
        .
        .
        .
```

Commentaires

Description de la grammaire Algol 60 pour l'analyse lexicographique.
Déclarations globales
ML : pointeur en zone locale
NO : numéro du bloc en cours
ALLOC contient pour chacun des blocs l'adresse en zone locale

```
PROC CHARGEREG(ELT M;REP REG;
ELT AI,ADR,RI,RESIN)
(REG:=RECHREG(ADR);
CHARGER(REG;AI,ADR,RI))
          :
          :
RANGEE(120)PROC LEX;
LEX[1]=( )
          :
          :
(ANAPPLYSE(1;CHAINE;LEX))
```

Procédure simple nécessaire aux procédures de la rangée suivie des autres procédures.

Description des procédures de la rangée LEX qui remplit la table d'identificateurs et la rangée ALLOC.

Pour les passages suivants, il faut fournir au programme de test les résultats des passages précédents ; ces résultats dépendent de la phrase à compiler , ils correspondent essentiellement à des initialisations de variables globales, et à des adjonctions dans les tables (par exemple tables d'identificateurs), ce qui se fait facilement dans l'expression fermée du programme Face, juste avant l'appel d'analyse, suivi de APPL ou d'ANAPPLYSE pour la rangée de procédures et la grammaire définissant le passage.

Par exemple, si nous considérons maintenant la mise au point du deuxième passage d'un compilateur Algol 60, associé à la rangée de procédures GEN, il faudra simuler les résultats de la rangée LEX ci-dessus, c'est-à-dire avoir rentré en table d'identificateurs, les identificateurs utilisés par la phrase de jeu d'essai, et construit le tableau ALLOC (cf. exemple précédent). Nous obtiendrons le programme de test suivant :

```
GRAM
programme:bloc|[étiquetage,bloc]
bloc :
          :
          :
```

Commentaires

Description de la grammaire propre au deuxième passage.

REP ML,NO:=1 RANGEE(1000)REP CHAINE; RANGEE(15)REP ALLOC;	Déclarations de variables globales.
PROC CHARGEREG( )	Déclarations des procédures suivies de leurs corps, utilisées dans la rangée GEN.
RANGEE(120)PROC GEN GEN[1]= ( )	Rangée de procédures décrivant le deuxième passage.
(RANGEE(2)REP INFO,INDIC; INFO[1]:=COMPACT(1,'R',1,0); INFO[2]:=1;INDIC[1]:= COMPACT(A, , ,); INDIC[2]:= COMPACT(, , ,); ENTREE(TABLE;INDIC;INFO;FILTRE) ALLOC[1]:=1 ; NB:=1; ANAPPLYSE(1;CHAINE;GEN))	On initialise INFO contenant les informations de l'identificateur puis les caractères de l'identificateur dans INDIC, et on l'entre dans la table. On initialise ALLOC et NB le nombre de blocs. Tout ceci pour simuler le passage précédent. Puis appel de ANAPPLYSE.

La phrase de jeu d'essai aurait pu être :

DEBUT REAL A ; A:=1 FIN.

On peut constater que la tâche de l'utilisateur serait facilitée en proposant des procédures d'initialisations de tables et d'écritures de résultats, repères ou rangées de repères. De fait on lui proposera, dans la bibliothèque de mise au point, des procédures d'écritures(cf. annexe 0).

Le test direct du premier passage est encore réalisable, car il n'y a pas de problèmes d'initialisations, mais les tests des autres passages deviennent rapidement lourds, si nous considérons des phrases importantes ; dans l'exemple ci-dessus le test du deuxième passage du compilateur Alogl 60 est pratiquement le test complet du compilateur.

Ces mises au point globales d'un passage seront utiles pour l'enchaînement des procédures d'une même rangée, mais devront être précédées de tests plus fins, par exemple sur chaque procédure de rangée.

1-3-3. Tests des procédures d'une rangée

Il est possible de tester seule la procédure F_i de numéro i, d'une rangée de procédures, associée à la règle i de la grammaire.

Soit par exemple la règle d'une grammaire Algol 60 définissant un bloc :

<bloc>::=<début><en tête de bloc>
<fin d'instruction composée>|^{98}

Nous voulons tester la procédure 98 de la rangée LEX. Cette procédure a deux arguments :

- le numéro du bloc précédemment en cours
- la valeur du compteur ML d'allocations en zone dynamique du bloc englobant.

Il suffit de créer une procédure fictive permettant de fournir ces deux renseignements et d'initialiser une rangée d'entiers permettant l'appel de cette procédure

fictive et de la procédure testée, voici le programme Face qui permettrait ce test :

Programme	Commentaires
REP ML,NO; RANGEE(98)PROC LEX;	Déclaration du compteur en zone dynamique et du numéro du bloc.
LEX[1]=(0,2)(P1:=2; P2:=10)	LEX1 est la procédure fictive initialisant les deux paramètres à 1 et à 10.
LEX[98]=(2,0)(ML:=P2; NO:=P1)	LEX98 est la procédure à tester.
(RANGEE(2)REP CHAINE:=(1,98); APPL(CHAINE;LEX));	

Naturellement une telle procédure n'aurait pas lieu d'être testée seule car elle est trop fine pour une mise en oeuvre trop lourde : il faut concevoir des programmes de tests pour toutes les procédures d'une rangée.

Il est utile de le faire pour des procédures plus importantes :

Si nous reprenons la règle de grammaire citée au paragraphe 1-2-2 :
Expression simple : terme]]expression simple,'+',terme]^{20};
nous obtenons un programme de test semblable :

Programme	Commentaires
PROC REP TESTTYPE=(REP X,Y) (.....)	Description de la procédure qui calcule le type du résultat en fonction des types X et Y.

```
PROC CONVERSION=(REP A,B,C,TYPE)
(.....              )
          :
          :
          :
RANGEE(2)PROC GEN;
GEN[1]=(0,6)(P1:='REEL'; P2:=5;
P3:=0; P4:='ENTIER'; P5:=10;
P6:=1)
GEN[20]=(6,3) (.... cf. § 1-2-2)
(RANGEE(2)REP CHAINE:=(1,20);
APPL(CHAINE;GEN))
```

CONVERSION : procédure qui effectue la conversion de la variable définie par A,B,C, suivant le type TYPE, et replace les résultats en A,B,C.
GEN[1] est une procédure fictive qui fournit les six paramètres.

Initialisation de la rangée d'entiers CHAINE et appel de APPL.

Il serait intéressant d'avoir une procédure standard de mise au point qui permette d'écrire les trois résultats fournis par la procédure GEN[20] ; c'est-à-dire une procédure standard qui écrive la pile gérée par APPL (cf. § 1-1-1).

Il est aussi pénible d'être obligé de changer la procédure fictive pour donner des valeurs diverses aux arguments de la procédure à tester ; nous remédierons à tout ceci en concevant un langage de mise au point (cf. chapitre III).

Dans la plupart des cas, le découpage du compilateur au niveau d'une procédure d'une rangée est trop fin, pour le test, aussi nous allons chercher un moyen terme entre le test complet d'un passage et le test d'une procédure d'une rangée par un découpage en paragraphes.

1-3-4. Test des procédures auxiliaires

Le programmeur est amené à écrire des procédures auxiliaires (cf. 1-2-2), il vaut mieux les avoir testées avant de les utiliser. L'écriture d'un programme Face peut permettre ce test.

Considérons une procédure auxiliaire, sans résultat, TYPE. Cette procédure vérifie si le type donné en argument est réel ou entier et sinon appelle la procédure standard Face ERREUR.

Programme	Commentaires
PROC TYPE=(ELT T)	Corps de la procédure TYPE.
T='REEL'∪T='ENTIER':(),ERREUR(3);	Expression fermée finale qui comprend : initialisation d'un élément au type 'REEL' et appel de la procédure TYPE.
(ELT X='REEL';TYPE(X))	

Ici aussi cette procédure est très simple, mais le test serait le même pour une procédure quelconque, il suffit dans une expression fermée d'initialiser les arguments, d'écrire un ou des appels de la procédure pour diverses valeurs d'essai des arguments et de pouvoir imprimer les résultats si ils existent.

I-4. DECOUPAGE DU COMPILATEUR EN PARAGRAPHES

Nous venons de voir que l'utilisateur peut commencer par écrire la grammaire du langage source, puis il définit le nombre et les résultats des passages du compilateur, puis il passe à l'écriture de la rangée de procédures définissant un de ces passages. Pour ceci il peut écrire les procédures de cette rangée une à une dans n'importe quel ordre en considérant la règle de numéro correspondant (cf. § 1-2-2).

Il peut faire une mise au point de sa grammaire (cf. § 1-3-1) mais la mise au point séparée de toutes les procédures d'une rangée est trop lourde (cf. § 1-3-3).

D'autre part il est un peu théorique de croire que le programmeur écrit les procédures d'une rangée dans un ordre quelconque : il écrit successivement certaines procédures de la rangée qui définissent des résultats formant un tout par leurs appels à l'intérieur du passage. C'est cet ensemble de procédures et leurs règles de grammaire associées que nous appellerons un PARAGRAPHE.

1-4-1. Paragraphe d'un compilateur. Exemples

Dégageons ce que peut être un paragraphe en prenant des exemples.

1-4-1-1. Mettons-nous à la place du programmeur d'un compilateur Algol 60 en Face, lors de l'écriture d'un passage de génération du texte objet, s'il considère la règle de grammaire:

<expr>::=<expr.simple>|si<expr.simple>alors<expr.simple>
sinon<expr>

Au moment où il veut écrire la procédure associée à la deuxième règle, il s'aperçoit que le découpage de la règle n'est pas suffisant car il a des actions à faire au niveau du 'alors' à savoir : vérifier si l'expression simple derrière le 'si' est bien de type booléen et générer le texte objet correspondant au saut si l'expression booléenne est fausse ; de même au niveau du 'sinon' il doit générer le saut de partie sinon, et sortir l'étiquette de cette partie.

Il va donc redécrire la règle, de façon descendante en partant de la règle précédemment écrite :

<expr>::=<expr.simple>|<partie avant sinon>sinon<expr>|86

<partie avant sinon>::=<proposition si><expr.simple>|85

<proposition si>::=si<expr>alors|84

Une fois redéfinie la grammaire, il écrira les procédures associées et de façon ascendante cette fois-ci, car la procédure 84 fournit des résultats aux procédures 85 et 86, et qu'il a tendance à générer le texte objet dans le même sens que le texte source.

Remarque :

Nous trouvons d'une manière générale cette démarche descendante pour le choix des règles, puis la démarche ascendante pour l'écriture des procédures.

1-4-1-2. Prenons maintenant, toujours dans un compilateur Algol 60, des règles de grammaire concernant un appel de procédure :

<appel de procédure>::=<identificateur><fin d'appel>

<fin d'appel>::=<debparam><liste de par.effectifs>)|64

<debparam>::=(|63

<liste de par.effectifs>::=<expr.simple>|65

<liste de par.effectifs><sépar><expr.simple>|65

<sépar>::=,|66

Le programmeur,lors de l'écriture du passage de génération,écrit ensemble les quatre procédures concernant ces règles, car elles s'enchaînent entre elles : les résultats des unes sont les données pour les autres, et elles génèrent des instructions du texte objet qui forment un tout.

1-4-1-3. Si nous considérons les règles des déclarations de tableau dans une grammaire Algol 60 :

<déclaration de tableau>::=<type tab><liste tab>|[18]

<type tab>::=TABLEAU|[31]

<liste tab>::=<tableau>|[28]<liste tab>;<tableau>

<tableau>::=<ident>[<liste de pdb>]|[29]

<ident>,<tableau>|[30]

<liste de pdb>::=<pdb>|[21]<liste de pdb>,<pdb>|[27]

<pdb>::=<expression>:<expression>|[26]

Pour le premier passage nous avons envie d'écrire les procédures de la rangée LEX, associées à ces règles, en même temps, car elles contribuent à fournir un résultat commun : construire et adjoindre les entrées des tableaux dans la table des symboles et allouer en zone statique la place nécessaire au vecteur de renseignements des tableaux [13].

Elles utiliseront les mêmes procédures simples pour réaliser leur adjonction en table des symboles.

De plus, une fois écrites les procédures du premier passage , pour ces règles, on se rend facilement compte des actions qu'elles devront faire au passage suivant de génération ; si bien que pour un tel ensemble de règles, on a envie d'écrire ensemble les procédures des rangées associées, pour tous les passages du compilateur qu'on écrit.

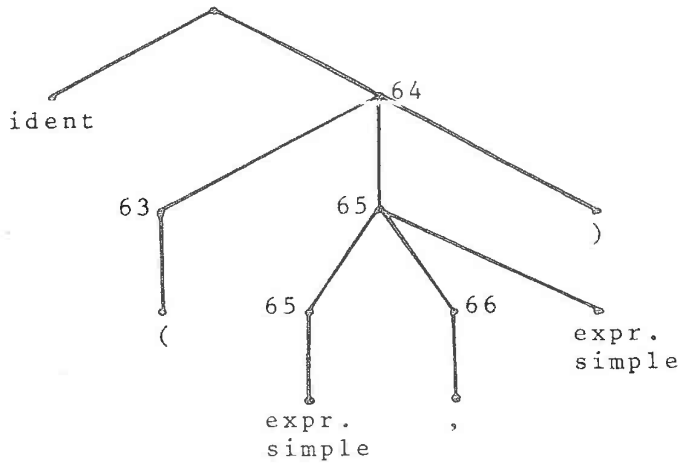
1-4-2. Caractéristiques d'un paragraphe

Ainsi au moment de l'écriture des rangées de procédures, on distingue des 'parties' du compilateur, qu'on appellera PARAGRAPHES et dont on peut dès à présent dégager les caractéristiques suivantes :

i) Les règles, guidant l'ensemble des procédures de la rangée ou des rangées du paragraphe, forment une 'sous-grammaire' où l'on distingue un ou plusieurs axiomes (dans les exemples,ce sont respectivement <expr>, <appel de procédure>, <déclaration de tableau>) et un ensemble de non-terminaux d'où ne dérivent pas de mots terminaux (dans l'exemple du paragraphe 1-4-1-1 le non-terminal <expr.simple> n'apparaît pas en premier membre d'une des règles).

ii) Si l'on considère ces non-terminaux comme des terminaux, la sous-grammaire considérée engendre des ramifications correspondant à des 'sous-ramifications' desramifications engendrées par la grammaire complète du compilateur.

Si nous prenons l'exemple du paragraphe 1-4-1-2 dont l'axiome était <appel de procédure>, la phrase suivante est engendrée par la grammaire décrite : 'ident(expr.simple,expr.simple)' et la ramification correspondante est :



Cette ramification sera une sous-ramification des ramifications engendrées par la grammaire complète pour toutes les phrases qui contiennent un appel de procédure à deux paramètres effectifs.

Remarquons que l'utilisateur conçoit la sous-grammaire en y considérant implicitement un axiome, et ceci est lié à la conception descendante de la sous-grammaire .

Pour mettre au point les procédures de la ou des rangées, il faut les appeler dans le même ordre d'appel pour la phrase de test du paragraphe, que pour un programme Face complet. Nous voyons donc que l'ordre d'appel entre les procédures de la ou des rangées, écrites pour le paragraphe, est respecté, mais l'effet des appels correspondant aux composantes manquantes devra être simulé, ceci sera étudié dans le chapitre suivant.

iii) De par la formation descendante de la sous-grammaire ou de par le regroupement de ses règles, les procédures de rangée du paragraphe coopèrent à la définition d'un même résultat.

Par exemple, les procédures associées à la déclaration de tableaux pour le premier passage (cf. § 1-4-1-3) s'occupent de l'adjonction à la table des symboles des tableaux trouvés et travaillent sur la même table.

Ou les procédures associées au paragraphe de l'expression conditionnelle (cf. § 1-4-1-1) vont générer, pour le deuxième passage, un morceau de texte-objet définissant la même fonction que le programme source, c'est ainsi que le programmeur vérifie son texte-objet généré.

iiii) Nous remarquerons que le partage du compilateur en paragraphes n'est pas exhaustif ; le résultat du paragraphe peut être plus ou moins complexe.

Dans l'exemple 1-4-1-3 cité ci-dessus pour la déclaration des tableaux, nous aurions pu envisager un paragraphe plus grand, qui permettrait de regrouper toutes les actions associées à l'ensemble des déclarations d'Algol 60.

Il est normal que le programmeur veuille tester ces paragraphes, lorsqu'il les écrit ; mais comme la sous-grammaire est incomplète, il va manquer des informations pour faire ce test en Face, aussi nous avons cherché à faciliter le travail de test en définissant un langage, permettant à l'utilisateur de décrire le paragraphe, et de donner les informations nécessaires au test, ce seront les buts des chapitres II et III.

1-4-3. Test d'un compilateur écrit en Face

Nous pouvons donc maintenant conclure comment l'utilisateur va concevoir, écrire et mettre au point un compilateur dans le langage Face. Voici les différentes étapes proposées :

1) Ecrire la grammaire du langage source dans son ensemble.

2) Vérifier la grammaire : tout d'abord il faut vérifier que la grammaire soit bien acceptée par le préanalyseur en la faisant compiler par Face ; puis il faut faire analyser des "phrases-types" du langage source par l'analyseur, et finalement demander de générer des phrases du langage engendré par la grammaire (cf. chapitre IV).

3) Concevoir le compilateur, c'est-à-dire établir le nombre et les résultats de chacun des passages du compilateur en fonction du langage source et de l'implémentation.

4) Ecrire et tester les procédures auxiliaires communes à un ou à plusieurs passages.

5) Il s'agit alors d'écrire et de tester chacun des passages : on regroupe les règles de grammaire qui définissent un même travail, qui formeront un PARAGRAPHE. On écrit alors les procédures d'une ou des rangées associées aux règles de grammaire du paragraphe, et on teste ces procédures rangée par rangée. Pour certains paragraphes, on testera en même temps que les procédures de la rangée et les procédures auxiliaires liées au paragraphe.

6) On peut alors mettre au point complètement chacun des passages, en commençant par le premier, puis ensemble les deux premiers et ainsi de suite, ce qui évitera pour un passage de simuler les passages précédents. Le compilateur sera alors entièrement mis au point.

CHAPITRE 2

DEFINITION DU PARAGRAPHE

II.1. LE PARAGRAPHE. GENERALITES

Nous venons de voir à la fin du chapitre I que l'utilisateur concevait le compilateur qu'il écrit en Face en le découpant en ce que nous avons appelé des paragraphes. Par exemple, le paragraphe de l'expression conditionnelle va comprendre sa grammaire :

<expr>::=<expr.simple>|<partie avant sinon>sinon <expr>|[86]

<partie avant sinon>::=<proposition si><expr.simple>|[85]

<proposition si>::= si<expr>alors|[84]

et les procédures d'une ou plusieurs rangées de procédures attachées aux règles 84, 85,86.

Le paragraphe est composé d'un ensemble de règles de grammaire formant une "sous-grammaire" et d'un ensemble de procédures de rangées, associées aux différentes règles de sa grammaire.

Nous voulons maintenant permettre à l'utilisateur une mise au point du compilateur par paragraphe, ainsi le paragraphe deviendrait aussi un module de mise au point d'un compilateur écrit en Face.

Pour cela il doit vérifier que la fonction effectuée par le paragraphe fournisse des résultats corrects à partir de jeux d'essais donnés par l'utilisateur. Nous avons vu (cf. § 1-1-1) que nous pouvons considérer un passage du compilateur comme une fonction COMP, sur une ramification $r=A\times(r_1+r_2+..+r_p)$ engendrée par la grammaire du paragraphe, COMP s'écrit :

$$COMP(r)=(x_1:=COMP(r_1);\ \ldots\ ;\ x_p:=COMP(r_p);\ \varphi_A(x_1;x_2;..;x_p))$$

où φ_A est une procédure du paragraphe associée à la règle A.

De la même façon, nous pouvons définir une fonction du paragraphe i : PAR_i, ayant pour argument une ramification engendrée par sa sous-grammaire. PAR_i est une composante de la fonction COMP, elle a la même fonction que COMP, mais sur un ensemble de définition plus petit : les ramifications engendrées par la sous-grammaire. Elle va définir certains des résultats de COMP, donc empiler des résultats intermédiaires à la fonction COMP.

Nous allons étudier comment nous pouvons définir la composante PAR_i du paragraphe automatiquement pour permettre à l'utilisateur une mise au point du compilateur écrit, au niveau de chaque paragraphe qu'il conçoit (cf. § 1-4-1).

II.2. LA GRAMMAIRE DU PARAGRAPHE

2.2.1. La grammaire

La sous-grammaire relative au paragraphe est une grammaire où des non-terminaux cités en deuxième membre des règles de grammaire, n'apparaissent pas en premier membre.

Si G est la grammaire complète du compilateur

$G = (N , T , ::= , X)$

où N est l'ensemble des non-terminaux
T est l'ensemble des terminaux
::= est la relation de production
X est l'axiome.

On peut définir pour le paragraphe une grammaire G'

$G' = (N' , T' , ::=' , X')$

où T' est un sous-ensemble de T
::=' est la restriction de ::= aux règles de grammaire du paragraphe
X' est un ensemble d'axiomes ($X' \subset N'$) et en général $X' \neq \{X\}$
$N' = N_1 \oplus N_2$ avec

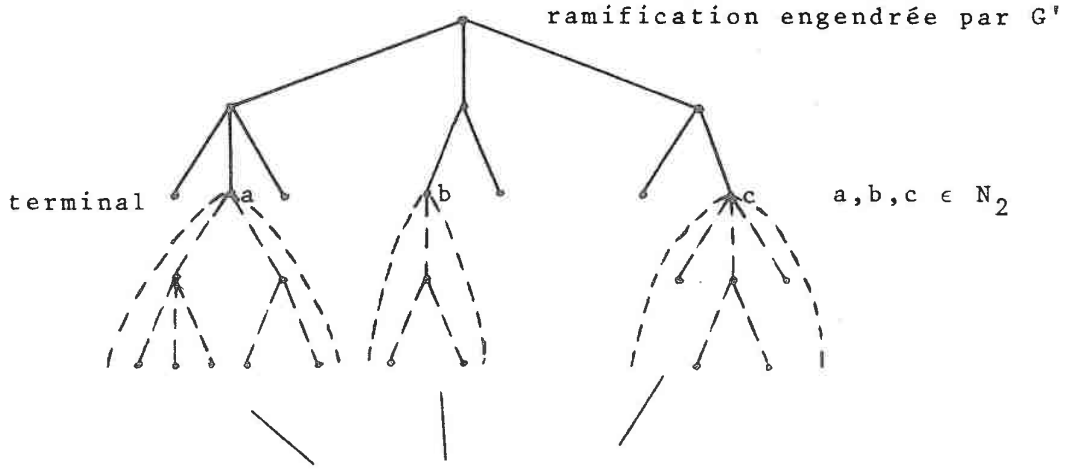
N_1 est un ensemble de non-terminaux qui apparais-

sent en premier membre de règles de grammaire de G'.

N_2 est un ensemble de non-terminaux de G, qui ont une occurrence dans un deuxième membre d'une règle de G'.

G' n'est donc pas réduite inférieurement.

Les éléments de N_2 engendrent dans la grammaire G des ramifications qui s'enracinent par ces éléments aux ramifications engendrées par G' :



Or la fonction PAR du paragraphe doit avoir comme arguments les résultats de la fonction COMP sur ces ramifications enracinées. Ces arguments seront définis par des "procédures manquantes" qui seront associées à des "règles manquantes" que l'on va ajouter à G'.

Des règles de grammaire de premier membre élément de N_2 seront ajoutées à G' pour former une grammaire du paragraphe.

Règles ($G_{paragraphe}$) = Règles (G') $\cup$ R_m

où R_m est l'ensemble des règles manquantes, dont les premiers membres appartiennent à N_2.

Reprenons l'exemple du paragraphe 1-4-1 sur l'expression conditionnelle :

<expr>::=<expr.simple>|<partie avant sinon>sinon<expr>|[86]

<partie avant sinon>::=<proposition si><expr.simple>|[85]

<proposition si>::=si<expr>alors|[84]

Dans cet exemple, nous avons :

T' = {sinon, si, alors}

N_1 = {expr, partie avant sinon, proposition si}

N_2 = {expr.simple}

Ici il n'y a qu'un axiome : <expr>

Et il faudra ajouter une règle manquante de premier membre : expr.simple.

2-2-2. Les règles manquantes

Les règles manquantes sont à ajouter à la grammaire du paragraphe et à chacune d'entre elles correspondra une procédure manquante, que nous étudierons par la suite.

Par exemple, s'il s'agit d'un paragraphe sur la déclaration de tableaux dans la grammaire Algol 60

<liste de déclarations de tableaux>::= TABLEAU <liste de tableaux>|[10] TABLEAU<liste de tableaux>; <liste de déclarations de tableaux>|[11]

<liste de tableaux>::=<tab>|[12]<liste de tableaux>;<tab>|[13]

<tab>::=<identificateur>|[15]<identificateur><dimension>|[14]

L'ensemble N_2 est {identificateur,dimension}.

Au premier abord, nous aurions envie de créer les règles manquantes ainsi :

identificateur::= |[1]

dimension::= |[2]

Mais si nous voulons nous servir de l'analyseur sur la grammaire du paragraphe, il y aura ambiguïté puisque, des deux règles de premier membre tab, dérive le vide ; de plus les phrases qu'il faudrait donner à analyser ne seraient pas très parlantes. Une"phrase-test" pour l'exemple ci-dessus serait : TABLEAU , ;TABLEAU ;

Nous allons donc placer en deuxième membre des règles manquantes, une des chaînes terminales qu'elles engendrent. Dans l'exemple nous aurions pu choisir :

identificateur::='IDENT'|[1]

dimension::='[10]'|[2]

Il n'y a alors plus aucune ambiguïté et comme la grammaire du paragraphe devra générer un ensemble de mots du langage, ces mots engendrés seront plus lisibles pour l'utilisateur.

Ainsi chaque règle manquante dans la grammaire du paragraphe devra avoir :

- un numéro de règle, qui permettra de lui faire correspondre sa procédure associée ;
- en deuxième membre de règle, une des chaînes terminales qu'elle engendre dans la grammaire complète du compilateur.

II.3. LES PROCEDURES MANQUANTES

Ce sont les procédures associées aux règles manquantes et qui vont permettre de simuler la fonction du passage du compilateur, sur les ramifications enracinées aux ramifications du paragraphe, comme nous l'avions vu au paragraphe 2-2-1.

2.3.1. Renseignements associés aux procédures manquantes : constitution des jeux d'essai des procédures manquantes

Considérons l'expression de la fonction comp, associée à un passage du compilateur sur la ramification $r = A\times(r_1 + \ldots + r_p)$

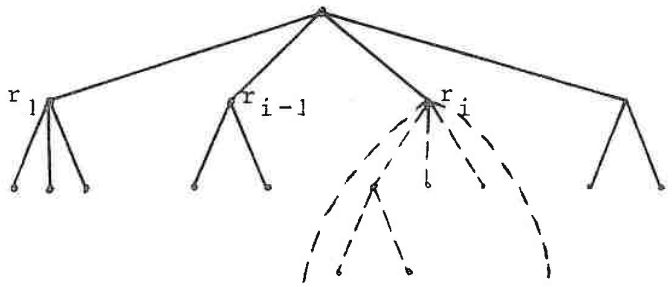
$$comp(r)=(x_1:=comp(r_1);x_2:=comp(r_2);\ \ldots\ ;x_p:=comp(r_p);\varphi_A(x_1;\ldots;x_p))$$

Si r_i est une sous-ramification dont la racine est un élément de N_2, c'est-à-dire un non-terminal Z, premier membre de règle manquante (cf. § 2-2-1), r_i est vide pour le paragraphe, ou plutôt réduite à la chaîne terminale donnée en règle manquante et au numéro de la règle manquante, cependant comp doit empiler l'ensemble des résultats qu'aurait eu r_i.

Une procédure φ_i, qu'on appelle procédure manquante, doit donc apporter au paragraphe un nombre β d'arguments, qui sont les résultats qu'aurait eus la fonction comp appliquée à la ramification r_i, de racine Z.

Si l'on veut que la vérification du paragraphe soit bonne, il faudrait envisager plusieurs jeux d'essai pour ces β données, résultats de comp sur plusieurs ramifications engendrées par Z dans la grammaire totale du compilateur.

Il se peut aussi, si r_i est toujours une sous-ramification dont la racine est dans N_2, que $comp(r_i)$ se serve des résultats produits par les (i-1) premiers appels $comp(r_j)$ $1 \leq j \leq i-1$.



La procédure manquante devra alors prendre des arguments sur la pile, avant de fournir des résultats pour les procédures suivantes.

Les procédures, des rangées de procédures, associées aux règles manquantes auront donc des paramètres comme les autres procédures (α ,β) : α nombre de paramètres en entrée pris sur le sommet de la pile et β nombre de paramètres en sortie nécessaires aux procédures suivantes, et qui seront des jeux d'essai du paragraphe.

Exemple :

Reprenons le cas de l'expression conditionnelle où l'utilisateur a découpé ainsi la grammaire :

<expression>::=<expr.simple>|<partie avant sinon><sinon><expression>|86

<partie avant sinon>::=<proposition si><expr.simple>|85

<proposition si>::= si<expression>alors|84

<expr.simple>::= EXP.SIMP |1

<sinon>::= SINON |2

} règles manquantes

Pour la génération, la règle associée au non-terminal <sinon> devait lui permettre de sortir l'étiquette prévue E1 pour la partie sinon et de créer E2 qui devra être sortie après la partie sinon. Le schéma du texte à générer étant

```
                Branchement si expression fausse à E1
                    :
                    :     partie alors
                    :
                    -
                Aller à E2
E1                  :
                    :     partie sinon
                    :
                    -
E2              suite
```

Ainsi gen[2] doit prendre un élément sur la pile, et en remettre un autre ; elle pourrait être :

gen[2] = (1,1) (P1='E2')

Par contre, pour gen[1] associée à l'expression simple, il suffit qu'elle monte les renseignements liés à une expression simple : type de l'expression, son adresse, une marque de résultat intermédiaire...

Ainsi les procédures manquantes élimineront de la pile les α arguments qu'elles doivent prendre, en les ignorant, et empileront les β résultats qu'elles doivent apporter et qui lui seront donnés par l'utilisateur. Nous étudierons dans le chapitre III un langage permettant de donner les jeux d'essai pour ces β résultats.

Remarquons que ces procédures agissent aussi par effet de bord (side effect) sur les résultats du passage testé : modification de tables, ou de repères globaux, génération de texte-objet... Nous ne ferons pas intervenir ces effets dans les procédures manquantes, car cela aurait beaucoup alourdi le langage, et de toute façon nous pouvons tester les procédures du paragraphe, ce qui est ici notre but; seulement les résultats du paragraphe sur les tables, repères, rangées de repères, texte généré seront un peu moins réalistes. En particulier pour l'exemple ci-dessus il manquerait le morceau de texte objet généré :

"allera E2;E1 :" . Mais dans cet exemple comme dans la plupart des cas, cela provient d'une mauvaise conception du paragraphe par l'utilisateur, qui doit définir le paragraphe en fonction des résultats voulus (cf. § 1-4-2 iii).

2-3-2. Indépendance entre eux des jeux d'essai des procédures manquantes

Considérons une règle manquante i et sa procédure associée pour un passage φ_i, et supposons que le nombre β de paramètres à empiler ne soit pas nul.

Pour la mise au point du paragraphe, il faut prévoir les cas possibles pour ces β paramètres, d'où les jeux d'essai possibles pour la procédure φ_i ; désignons-les par $\varphi_{i,1}$, $\varphi_{i,2}$, ... , φ_{i,k_i} .

Prenons maintenant p règles manquantes avec leurs divers jeux d'essai : $\varphi_{i,1}$, ... , φ_{i,k_i} . $(i = 1,p)$

Nous supposerons que les jeux d'essai pour ces p procédures sont indépendants et nous prendrons donc toutes les combinaisons de ces jeux d'essai pour le test de la rangée. En fait les jeux d'essai des procédures manquantes ne sont pas vraiment indépendants, mais le fait de les considérer comme indépendants n'est pas un obstacle, le programmeur saura que le test est fait sur toutes les combinaisons des jeux d'essai, ce qui lui permettra de détecter éventuellement les résultats dans le cas d'erreurs de compilation, ce qu'il a aussi à déceler dans toute bonne mise au point.

Nous allons mieux voir ces cas sur deux exemples :

ⓐ Considérons la sous-grammaire réduite à la seule règle :
<deuxième membre>::= si<expression booléenne>alors
<expr.simple>sinon<expression>

Les règles de premier membre <expr.simple> , <expression booléenne> et <expression> sont des règles manquantes, les jeux d'essai pour les procédures associées à ces règles sont indépendants entre eux, ils sont par exemple

pour <expression booléenne> : - son adresse
- indicateur de résultat intermédiaire ou non (pour savoir la zone où il se trouve)

pour <expression simple> : - son type : entier ou réel
- son adresse
- indicateur de résultat intermédiaire

pour <expression> : - son type : booléen, entier ou réel
- son adresse
- un indicateur de résultat intermédiaire

Faire varier l'indicateur de résultat intermédiaire n'est pas important ici ; nous pouvons prendre comme jeu d'essai de <expression booléenne>: (5,0) une adresse quelconque et pas de marque de résultat intermédiaire; pour <expression simple> deux jeux d'essai : ('ENTIER',12,0) et ('REEL',12,0) et pour <expression> trois jeux d'essai : ('BOOLEEN',10,0) ; ('ENTIER',10,0) et ('REEL',10,0). Ces jeux d'essai sont totalement indépendants entre eux.

(b) Prenons maintenant la grammaire réduite aussi à la règle :
<primaire indicé>::=<identificateur><indiçage>
<identificateur>et<indiçage>sont aussi premiers membres de règles manquantes.

Supposons que pour l'étude lexicographique <identificateur> ait quatre paramètres : type de l'identificateur, type, adresse, marque de résultat intermédiaire. Les jeux

d'essai possibles pour la procédure associée à <identificateur> sont ('TABLEAU','ENTIER',5,0), ('TABLEAU','REEL',5,0), ('ELEMENT','ENTIER',5,0), ('ELEMENT','REEL',5,0).

Nous voyons que, quels que soient les jeux d'essai pour la procédure associée à <indiçage>, seuls les deux premiers conviendront pour <identificateur> ; ceci est lié à l'existence des contraintes contextuelles que la grammaire à contexte libre ne prend pas en compte.

Cependant, il est nécessaire de considérer aussi les autres cas, pour permettre à l'utilisateur de bien prévoir tous les cas d'erreur dans son compilateur.

Ainsi si l'utilisateur fournit divers jeux d'essai par procédure manquante, nous les considèrerons comme indépendants dans l'interprétation de la fonction PAR du paragraphe.

Etudions maintenant comment sont utilisés ces jeux d'essai associés au paragraphe dans l'interprétation du paragraphe.

2-3-3. Etude de l'utilisation des jeux d'essais associés aux procédures manquantes

Grâce à l'addition des règles manquantes à la sous-grammaire, et des procédures manquantes aux rangées de procédures, il sera possible d'interpréter l'exécution de la fonction PAR du paragraphe, pour cela il faut avoir des arguments à lui fournir : des mots engendrés par la grammaire du paragraphe ou plutôt des ramifications sous forme postfixée sur les numéros de règles de cette grammaire. Elles permettent de tester les procédures φ_i des rangées de procédures du paragraphe en appliquant ces procédures dans l'ordre qu'elles indiquent (cf. APPL § 1-1-1). Nous les appellerons les "phrases de test" du para-

graphe. Cet ensemble de phrases de test sera choisi automatiquement grâce à la procédure standard GENPHRASE, procédure qui sera étudiée au chapitre IV.

Etudions l'utilisation des jeux d'essai associés aux procédures manquantes, pour une phrase de test donnée.

Reprenons le paragraphe de l'expression conditionnelle en y ajoutant les règles manquantes :

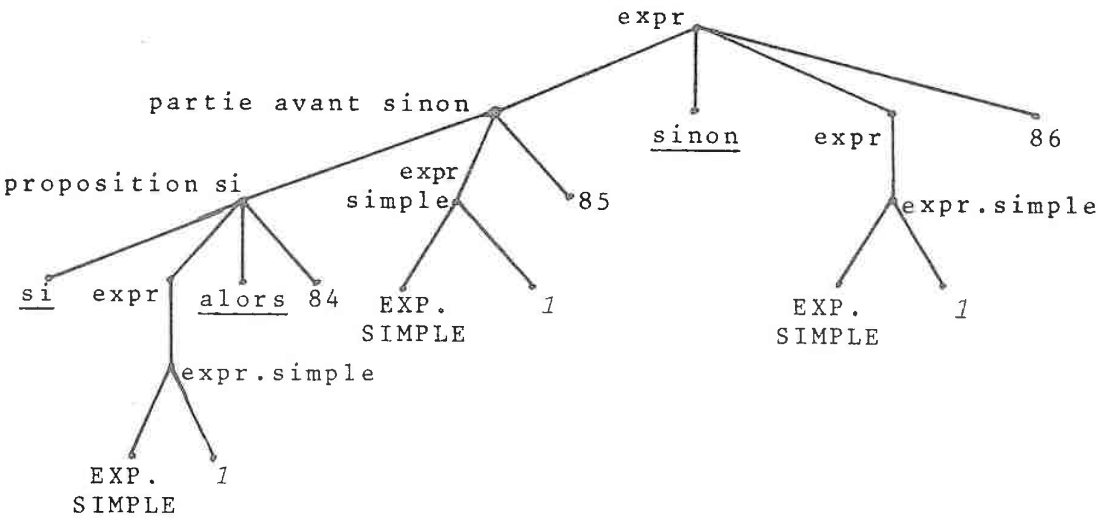
<expr>::=<expr.simple>|<partie avant sinon>sinon<expr>|86

<partie avant sinon>::=<proposition si><expr.simple>|85

<proposition si>::=si<expr>alors|84

<expr.simple>::=EXP.SIMPLE|1

Une des ramifications engendrées par cette grammaire est :



d'où la phrase de test : 1 84 1 85 1 86

associée au mot du langage :

SI EXP.SIMPLE ALORS EXP.SIMPLE SINON EXP.SIMPLE

Supposons que la procédure manquante φ_1 associée à la règle 1 doive empiler trois renseignements :

- le type : booléen, entier, réel
- l'adresse
- une marque de résultat intermédiaire

L'adresse et la marque de résultat intermédiaire ne servent qu'au calcul de l'adresse, calcul effectué par une procédure simple déjà testée. Aussi, l'utilisateur ne donne pas diverses valeurs d'adresses et de résultat intermédiaire au jeu d'essai, par contre le type peut prendre les trois valeurs, qui sont toutes trois nécessaires au test.

L'utilisateur peut proposer les trois jeux d'essai suivant :

	Jeu d'essai 1	Jeu d'essai 2	Jeu d'essai 3
type	'BOOLEEN'	'ENTIER'	'REEL'
adresse	5	5	5
marque de rés. intermédiaire	1	1	1

Si nous interprétons le paragraphe avec comme argument la phrase de test citée ci-dessus, nous devons appeler trois fois la procédure φ_1 manquante ; pour chacun de ces trois appels nous utilisons l'un ou l'autre des trois jeux d'essai, soit pour cette phrase 9 combinaisons possibles entre les jeux d'essai.

En définitive, pour une ramification, "phrase de test", nous considérons toutes les combinaisons des jeux d'essai associés aux procédures manquantes, entre les divers appels à des procédures manquantes à effectuer pour cette ramification.

Cependant on peut craindre aussi d'effectuer trop d'essais, par exemple dans la phrase :

si expr.simple alors expr.simple sinon expr.simple

pour le premier appel à expr.simple il aurait suffi de prendre le type 'BOOLEEN', puis 'ENTIER', car le type 'REEL' n'apporte rien de plus.

Nous essayerons d'y remédier, en faisant intervenir l'utilisateur en cours d'interprétation (cf. § III.6).

II.4. LE CONTEXTE DU PARAGRAPHE

Soit une ramification r engendrée par la grammaire du paragraphe, cette ramification r est une sous-ramification d'une ramification R de la grammaire complète du langage source.

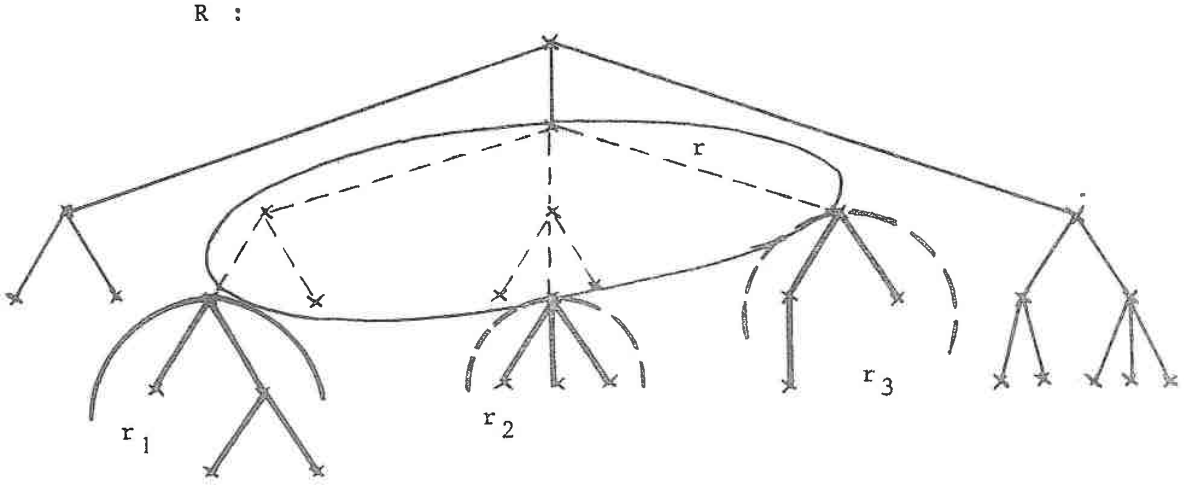


figure 1

Nous avons étudié dans le paragraphe 2.3.1 les arguments de la fonction PAR fournis par les ramifications r_i (r_1, r_2, r_3 du schéma), enracinées à la ramification r . Cependant pour exécuter la fonction PAR sur la ramification r, il faut replacer r à l'intérieur de R.

Ainsi nous aurons besoin des données suivantes pour interpréter le paragraphe :

i) les résultats globaux provenant des (k-1) passages précédents du compilateur sur la ramification R, si nous testons le $k^{ième}$ passage. Si nous notons $COMP_k$ la fonction définissant ce $k^{ième}$ passage, nous devons avoir les résultats globaux de $COMP_1$, $COMP_2$, ..., $COMP_{k-1}$.

ii) les résultats globaux et les paramètres des procédures de rangées provenant du même passage k, mais pour les sous-ramifications qui précèdent r à l'intérieur de R.

Tous ces résultats nécessaires au test du paragraphe forment ce que l'on appellera le contexte du paragraphe.

2-4-1. Etude du contexte

Les données nécessaires au test du paragraphe dues à l'apport du contexte sont :

i) Des "résultats globaux", valeurs désignées par des repères ou des rangées de repères du programme Face décrivant le compilateur ou l'état de tables à l'entrée du paragraphe. Pour définir ces résultats, il faut tenir compte des passages précédents et aussi de l'action du passage que l'on teste et ceci sur les ramifications qui précèdent celle du paragraphe (voir schéma 2 § 2-4-1 ii). Il suffit pour cela d'initialiser les repères et rangées de repères globaux utilisés dans le paragraphe et d'effectuer l'initialisation des tables par des adjonctions appropriées.

Par exemple, dans le paragraphe de l'expression conditionnelle du compilateur Algol 60, cité à plusieurs reprises (cf. § 2-3-3), nous avons besoin d'initialiser le pointeur en zone statique ML et aussi le niveau du bloc dans lequel on se trouve, à une valeur quelconque, 1 par exemple.

Ces résultats globaux pourront se définir par une expression fermée de Face, qui contiendra essentiellement des affectations et des appels de procédures pour initialiser les tables.

En outre, cette expression fermée peut contenir des actions concernant les procédures manquantes du paragraphe ; en effet les procédures manquantes, que nous créons, n'ont pas d'actions sur les repères, rangées de repères et les tables (cf. la remarque du § 2-3-1) ; et l'utilisateur peut, si cela l'intéresse, simuler aussi l'action des procédures manquantes sur les résultats globaux.

Voici l'exemple d'une expression fermée Face permettant d'initialiser le contexte du paragraphe d'une expression arithmétique qui utiliserait deux identificateurs A et B, déclarés auparavant :

Programme	Commentaire
(RANGEE[3]REP INFO:=('REEL',1,1); RANGEE[2]REP INDIC:=('A ' , ' '); ENTREE(TABLE;INDIC;INFO;FILTRE);	Entrée dans la table des symboles de A, qui a un caractère, et se trouve placé à l'adresse 1 de la zone variable.
INFO[1]:='ENTIER';INFO[2]:= 1; INFO[3]:=2; INDIC[1]:='B ';INDIC[2]:=' '; ENTREE(TABLE;INDIC;INFO;FILTRE); ML:=1 ; NIVEAU:=1)	Entrée de l'identificateur B, placé à l'adresse 2. Initialisation des deux pointeurs précédemment cités pour simuler le bloc

ii) Des "résultats empilés" des procédures, résultats empilés par la fonction COMP du passage considéré appliqué aux ramifications précédant la ramification du paragraphe ; ces résultats sont les arguments des procédures φ_i du paragraphe.

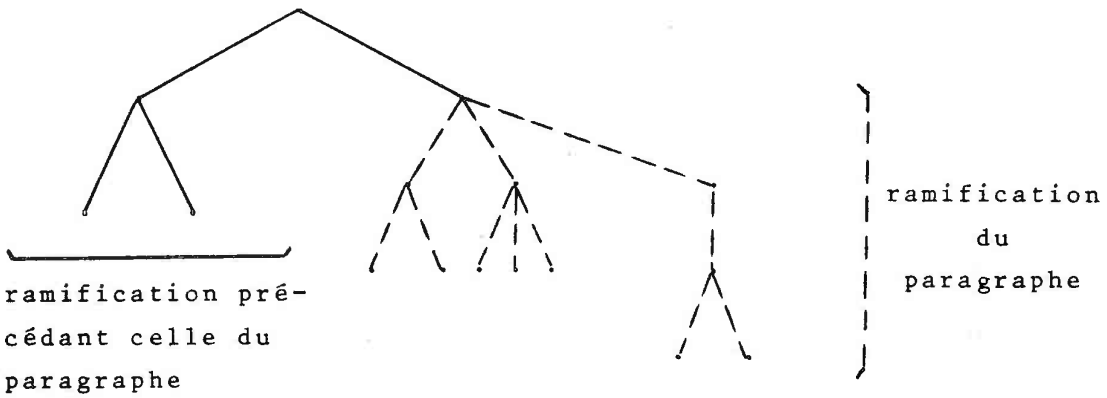


figure 2

Dans la plupart des cas, ces résultats empilés sont inexistants et l'utilisateur doit concevoir ses paragraphes pour qu'il en soit ainsi.

Si nous considérons l'exemple, déjà cité, sur la déclaration de tableau (cf. § 2-2-2), et l'utilisateur veut tester le paragraphe dont la sous-grammaire est réduite aux deux règles :

<liste de tableaux>::=<tableau>|[12]<liste de tableau>;
<tableau>|[13]

<tableau>::=<identificateur>|[15]<identificateur><dimension>|[14]

Les procédures associées aux règles 12 et 13 doivent faire les entrées dans la table des symboles, et pour cela il faut le type des tableaux déclarés, qui est un résultat d'une procédure associée à une règle n'appartenant pas au paragraphe, de même cette procédure doit initialiser un compteur du nombre d'identificateurs déclarés comme tableau et initialiser le Nil, base de rangée en pile. En fait si l'on conçoit un paragraphe qui vérifie l'adjonction en table des symboles, des identificateurs de tableaux, il est clair que cette règle devrait appartenir au paragraphe.

Pour obtenir les renseignements manquants devant être empilés, il suffit d'avoir une "procédure supplémentaire" qui fournirait ici trois paramètres :

. type : réel, entier ou booléen
. 0 : compteur d'identificateur
. 'NIL': base d'une rangée en pile.

Cette procédure supplémentaire doit empiler des valeurs-jeux d'essai ; son rôle est identique à celui d'une procédure manquante.

L'utilisateur doit fournir pour interpréter le paragraphe "les résultats globaux" et des "résultats empilés" et nous avons vu que les premiers pouvaient s'exprimer dans une expression fermée Face et les seconds comme les jeux d'essai d'une procédure manquante, par la procédure supplémentaire.

2-4-2. Procédures manquantes et contexte

En étudiant de façon plus précise comment l'utilisateur définit les résultats globaux à donner dans le contexte, nous allons voir que ceux-ci sont étroitement liés aux jeux d'essai associés aux procédures manquantes.

Prenons de suite un exemple, dont voici une règle de grammaire :

<primaire>::=<primaire indicé>|<identificateur>|[74]
<appel de procédure>

Nous voulons tester la procédure associée à la règle 74, lors de l'étude de la génération dans le cadre d'un compilateur Algol 60.

La procédure associée à la règle 74 doit faire une recherche de l'identificateur en table des symboles, pour obtenir les renseignements qui y ont été placés lors d'un passage précédent.

La procédure manquante associée à <identificateur> empile un résultat, qui est la chaîne des caractères de l'identificateur.

Nous allons considérer trois jeux d'essai pour cette procédure manquante : identificateur A, puis B, puis C. Pour générer des calculs avec des types différents, nous prendrons A de type réel, B de type entier, et pour provoquer la détection d'une erreur par le compilateur pour l'utilisation d'un identificateur non déclaré, nous prendrons C non en table. Le contexte devra englober les contextes des trois jeux d'essai : mettre A en table avec le type réel, B avec le type entier, il n'y a rien à faire pour C.

D'une manière générale, il faut prévoir à l'intérieur du contexte, tous les résultats des passages précédents, sur toutes les sous-ramifications manquantes prévues pour le test.

2.4.3. Procédures auxiliaires de test

Il s'agit ici de procédures auxiliaires, que l'utilisateur a définies mais qu'il n'a pas encore ni écrites en Face, ni testées, et qui sont citées dans les procédures d'une rangée du paragraphe.

L'utilisateur peut sentir le besoin d'écrire pour l'essai du paragraphe une "procédure fictive" fournissant uniquement un résultat valable pour un premier test du paragraphe.

Par exemple, dans le paragraphe de l'expression conditionnelle cité souvent, les procédures de rangées utilisent la procédure :
CHARGEREG() : procédure qui cherche un registre libre et donne en résultat le numéro du registre.

L'utilisateur peut envisager une procédure fictive CHARGEREG qui donnerait toujours le numéro 1, et écrire plus tard, lors de la mise au point d'autres parties, la procédure réelle CHARGEREG , car celle-ci n'a pas d'importance pour une mise au point d'un paragraphe.

II.5. MISE AU POINT D'UN PARAGRAPHE

2-5-1. Données nécessaires au test

Nous allons résumer ici les données nécessaires à l'exécution d'une fonction PAR associée au paragraphe, ces données se partagent en deux groupes :

Une partie Face appartenant au compilateur écrit en Face, comprenant essentiellement :

1) La sous-grammaire du paragraphe (non réduite).
2) Les procédures de rangées de procédures, ce sont elles que l'on veut en fait tester

Une partie Jeu d'essai, nécessaire à l'interprétation de la fonction PAR, elle comprend :

1) Les règles manquantes
2) Pour chaque procédure, dite procédure manquante, associée à une règle manquante, le nombre de paramètres en entrée et en sortie, et les divers jeux d'essai pour chacun des paramètres en sortie
3) Le contexte du paragraphe
4) Accessoirement, des procédures fictives simulant des procédures utilisées à l'intérieur des rangées et non écrites sous leur forme définitive.

2-5-2. Conclusion

Pour que l'utilisateur Face puisse mettre au point les paragraphes, il suffit de lui fournir un langage de mise au

point : c'est pour cette raison que nous avons conçu le langage MFACE, permettant de décrire la partie jeu d'essai, c'est-à-dire de décrire les données nécessaires au test pour interpréter la fonction PAR du paragraphe. La définition du langage MFACE et de son implémentation sont l'objet du chapitre suivant.

CHAPITRE 3

LE LANGAGE DE MISE AU POINT

ET SON IMPLEMENTATION

III.1. PROGRAMME DE MISE AU POINT

3-1-1. Généralités

Le langage de mise au point MFACE est un langage permettant à l'utilisateur de décrire la mise au point d'un paragraphe du compilateur écrit en Face. Il va prendre en compte le texte Face du compilateur et permet de décrire les jeux d'essai nécessaires à la mise au point d'un paragraphe.

Un programme MFACE sera composé de trois parties : la première rassemble le programme Face du paragraphe, qui sera ultérieurement stockée pour former le compilateur complet (cf. conclusion), la deuxième partie est formée des jeux d'essai du paragraphe et la troisième partie décrit les résultats à imprimer, résultats provenant de l'interprétation de la fonction définie par le paragraphe (cf. § II.1) sur les jeux d'essai de la deuxième partie. L'interprétation se fera à l'aide de deux procédures standard MFACE : GENPHRASE et APPLTEST que nous allons étudier.

3-1-2. Les procédures standard d'interprétation du paragraphe

Pour tester la (ou les) rangées de procédures du paragraphe, il faut appliquer les procédures dans un ordre correspondant aux représentations postfixées des ramifications engendrées par la grammaire. GENPHRASE va donc être une procédure générant un ensemble de tests pour le paragraphe, c'est-à-dire un ensemble de ramifications engendrées par la grammaire du paragraphe. Pour chacune de ces ramifications sera indiqué à l'utilisateur le mot du langage correspondant. GENPHRASE a donc pour argument un numéro d'une règle correspondant à l'axiome et pour résultat un ensemble de rangées d'entiers (représentations postfixées des ramifications) et un ensemble de rangées de chaînes de caractères (mots du langage correspondants).

APPLTEST va alors appliquer les procédures de la rangée suivant l'ordre de chacune des ramifications de l'ensemble créé par GENPHRASE ; elle a pour argument la rangée de procédures et communique à l'utilisateur à la fin de l'exécution de chacune des ramifications, les résultats pris sur la pile et les résultats demandés par l'utilisateur. De plus APPLTEST considère successivement les combinaisons des jeux d'essai des procédures manquantes citées dans les ramifications (cf. § II.2-3-3).

L'action de ces deux procédures GENPHRASE et APPLTEST peut se combiner grâce à une autre procédure standard GENAPPLTEST, qui, au fur et à mesure de la génération des ramifications, effectue APPL (procédure standard Face) avec l'impression des résultats, ceux pris sur la pile et ceux demandés par l'utilisateur .

3-1-3. Première partie : programme Face à tester

Elle débute par ATESTER et est constituée de la partie Face du paragraphe, à savoir :

. la grammaire du paragraphe, c'est-à-dire la sous grammaire à laquelle sont ajoutées les règles manquantes (cf. § 2-2) ;

. les procédures des rangées à tester : c'est-à-dire les rangées de procédures décrivant en Face les procédures φ_i associées aux numéros des règles de la sous-grammaire. Ce sont les procédures des rangées à mettre au point et définitivement écrites dans le langage Face ;

. les procédures auxiliaires du compilateur, écrites en Face, et nécessaires à l'exécution des procédures de la rangée. Il s'agit aussi bien des procédures déjà mises au point (cf. § 1-3-4) que des procédures fictives, simulées par le programmeur pour les besoins du test (cf. § 1-5-1) ;

. les déclarations de repères, et de rangées de repères, globaux du compilateur écrit en Face, utilisés dans les procédures simples et dans les procédures des rangées;

. une expression fermée de Face qui décrit le contexte du paragraphe (cf. § 2-4-1). Elle comprend essentiellement des initialisations de repères et rangées de repères et des appels de procédures.

3-1-4. Deuxième partie : jeux d'essai du paragraphe

Elle débute par JEU D'ESSAI et décrit le jeu d'essai correspondant aux procédures manquantes (cf. § 2-3-1). Pour cela l'utilisateur donne les valeurs (des entiers ou chaînes de caractères, c'est-à-dire des éléments de Face) qu'il attribue aux résultats empilés par les procédures manquantes, et qui seront arguments de l'interprétation du paragraphe.

Si nous reprenons l'exemple du paragraphe 2-3-1 sur l'expression conditionnelle dans le compilateur Algol 60 pour le passage de génération :

<expression conditionnelle>:=<partie avant sinon> sinon<expression>|86

<partie avant sinon>:=<proposition si><expr.simple>|85

<proposition si>:=si<expression>alors|84

<expression>:=EXPRESSION SIMPLE|1
<expr.simple>:=EXPRESSION|2
} règles manquantes

<expression> est un non-terminal manquant auquel correspond une procédure manquante de numéro 1, qui fournit trois arguments: le type, l'adresse relative à la zone, et une marque de résultat intermédiaire.

La procédure manquante associée au non-terminal <expr.simple> a deux résultats : le type, et l'adresse de l'expression simple.

Pour une ramification quelconque engendrée par cette grammaire nous allons donner des jeux d'essai pour les procédures manquantes ; pour l'expression nous allons

prendre tous les types possibles, avec une adresse quelconque et pas de résultat intermédiaire, et un cas avec un résultat intermédiaire, c'est-à-dire les jeux d'essai :

```
'BOOLEEN'   'REEL'   'ENTIER'   'REEL' : type
    10         10        10         5    : adresse relative
    0          0         0          1    : 0 si pas de résul-
                                             tat intermédiaire
                                          1 sinon
```

pour la procédure manquante associée à expr.simple, le cas :

```
'REEL'    'ENTIER'      : type
  15         15         : adresse relative
```

L'utilisateur pourra donner intégralement ses jeux d'essai par exemple

```
gen[2]=(0,3)('BOOLEEN',10,0);('REEL',10,0);('ENTIER',10,0);
       ('REEL',5,1)
```

mais il pourra aussi factoriser ses jeux d'essai, c'est-à-dire gen[2] deviendrait :

```
gen[2]=(0,3)(['BOOLEEN','REEL','ENTIER'],10,0);('REEL',5,1)
```

et pour la procédure associée au non-terminal <expr. simple> :

```
gen[1]=(0,2)(['ENTIER','REEL'],15)
```

Cette façon de factoriser les données permet de condenser ces données et correspond au mode de pensée de l'utilisateur : en concevant son jeu d'essai, il voit qu'il doit faire varier le type, avec un type d'adressage quelconque, donc il prend la même adresse pour tous les types choisis.

D'une façon générale, si une procédure manquante doit monter trois éléments sur la pile, donnés sous la forme :

$((\alpha_1,\alpha_2),(\beta_1,\beta_2,\beta_3),\gamma_1)$; $(\mu,(\nu_1,\nu_2),\pi)$

ceci correspondra aux différents jeux d'essai suivants :

α_1 α_2 α_1 α_2 α_1 α_2 μ μ
β_1 β_1 β_2 β_2 β_3 β_3 ν_1 ν_2
γ_1 γ_1 γ_1 γ_1 γ_1 γ_1 π π

où les α_i, β_j, γ, μ, V, π sont des éléments du langage Face.

3-1-5. Troisième partie : les résultats à imprimer

Elle débute par RESULTATS, si elle existe ; et l'utilisateur y indique une liste de repères ou rangées de repères globaux qui doivent être imprimés en fin d'interprétation du paragraphe, pour chacune des ramifications considérées et pour chacun des jeux d'essai. Ce sont des effets annexes que peut avoir le paragraphe sur des repères ou rangées de repères et que le programmeur veut contrôler. De façon implicite, les résultats du paragraphe qui sont empilés, et la génération de texte objet seront imprimés.

Il restera à imprimer les tables pour contrôler entièrement l'interprétation, ce qui pourrait être fait par une procédure standard appropriée à la mise au point (cf. remarque du § 3-3-2).

Nous allons maintenant préciser la syntaxe et la sémantique du langage de mise au point MFACE.

III.2. DESCRIPTION DU LANGAGE

3-2-1. Programme

<programme>::=A TESTER<pgme Face>JEU D'ESSAI<jeux d'essai donnés>
RESULTATS<liste de résultats>|
A TESTER<pgme Face>RESULTATS<liste de résultats>|
A TESTER<pgme Face>JEU D'ESSAI<jeux d'essai donnés>

Sémantique :

pgme Face en MFACE a la même syntaxe que la règle <programme> dans la grammaire Face [3], elle permet d'écrire ce que nous avons vu au paragraphe 3-1-3, soit la partie Face du paragraphe.
jeux d'essai donnés permettent de présenter les jeux d'essai des procédures manquantes, et liste de résultats décrit la liste des résultats à imprimer.

3-2-2. Jeux d'essai donnés pour les procédures manquantes

<jeux d'essai donnés>::=<pile de départ><liste de proc. manquantes>|<liste de proc.manquantes>

<pile de départ>::=<identificateur>=(<entier décimal>) <liste de jeux d'essai>

<liste de proc.manquantes>::=<proc.manquante>|<proc. manquante><liste de proc.manquantes>

<proc.manquante>::=<en-tête><liste de jeux d'essai>| <en-tête>

<en-tête>::=<identificateur>[<entier décimal>]= (<entier décimal>,<entier décimal>)

<liste de jeux d'essai>::=(<jeu d'essai>)|<liste de jeux d'essai>;(<jeu d'essai>)

<jeu d'essai>::=<valeurs d'essai>|<valeurs d'essai>, <jeu d'essai>

<valeurs d'essai>::=<élément>|[<liste>]

<liste>::=<élément>|<liste>,<élément>

Sémantique ::= l'identificateur désigne une rangée de procédures qui a été déclarée dans le <pgme Face> de la première partie.
pile de départ correspond à la procédure supplémentaire qui complète le contexte, pour cela elle initialise la pile s'il le faut (cf. § 2-4-2).

l'en-tête des procédures manquantes est de type (K,L) qui ont la même signification qu'en Face [3]. K est le nombre d'arguments que la procédure prendrait sur la pile et L le nombre de résultats qu'elle empile ; elle doit fournir L arguments au paragraphe, ceux-ci sont donnés par la liste de jeux d'essai. Jeu d'essai décrit les valeurs d'essai pour chacun des L paramètres, n'importe quel élément d'une de ces listes est compatible avec tout autre élément d'une autre liste à l'intérieur d'un même jeu d'essai (cf. § 3-1-4). Une liste ne peut contenir qu'un élément. L'élément est l'élément de la grammaire Face.

Exemple :

Pour l'exemple cité au paragraphe 3-1-4, qui n'a pas de pile de départ à initialiser, nous aurions la partie de programme MFACE suivante :

```
gen[2]=(0,3)(('BOOLEEN','REEL','ENTIER'),10,0);
       ('REEL',5,1)
gen[1]=(0,2)(('REEL','ENTIER'),15)
```

3-2-3. Les résultats

<liste de résultats>::=<identificateur>;|<identificateur>,
<liste de résultats>

Sémantique :

L'identificateur est syntaxiquement un identificateur Face [3] et désigne ici un repère ou une rangée de repères qui sont des objets globaux déclarés dans le <pgme Face> de la première partie.

III.3. EXEMPLE DE PROGRAMME MFACE

3-3-1. Présentation de l'exemple

Nous considérons un langage très simple d'expression arithmétique, qui n'a que le type entier sans déclaration. La compilation se fera en un seul passage.

Nous voulons ici faire la mise au point du paragraphe qui concerne les expressions arithmétiques, dont la grammaire serait :

<E>::=<E>+<T>|<T>
<T>::=<T>×<F>|<F>
<F>::=(<E>)|<entier>|<ident>

Nous appelons COMP la rangée de procédures permettant la compilation.

3-3-2. Programme de mise au point du paragraphe

Programme	Commentaires
A TESTER	
E : E,'+',T|10|T ;	} grammaire initiale
T : T,'×',F|12|F;	
F : '(',E,')'||ENTIER|15|IDENT|16;	
ENTIER : '10'|1;	} règles manquantes
IDENT : 'ID'|2;	
REP PC:=1, PV:=1;	Déclarations de repères et rangées de repères
RANGEE(3)REP OCCUPREG:=(1,0,0)	PC:pointeur en zone constante
PROC REGISTRE=(ELT P,Q;REP REG)	PV:pointeur en zone variable
	OCCUPREG:tableau d'occupations des 3 registres existants.
(TREG(1)=0:REG:=1,TREG(2)=0:	Registre : procédure qui cherche un registre libre, s'ils
REG:=2,TREG(3)=0:REG:=3,	sont tous occupés, en libère
LIB(REG));	un grâce à la procédure LIB.
PROC LIB=(REP REG)	
(.....);	

Suite du programme

```
PROC ELT FILTRE =
   (RANGEE REP PINF)(1);
```

```
PROC ELT FILTRE1 =
   (RANGEE REP PINF)
   (0; PC:=PC-1; RETOUR())
```

```
PROC ORDRE = (ELT ETI,CODE;
   REP REG;ELT R1,R2)(....
   ............);
```

```
RANGEE(16)PROC COMP;

COMP[10]=(4,2)
(REP REG;REGISTRE(P1;P2;REG);
ORDRE(' ';'LOD';REG;P1;P2);
ORDRE(' ';'ADD';REG;P3;P4);
P1:='REGISTRE';P2:=REG);
```

```
COMP[12]=(4,2)
(REP REG;REGISTRE(P1;P2;REG);
ORDRE(' ';'LOD';REG;P1;P2);
ORDRE(' ';'MUL';REG;P3;P4);
P1:='REGISTRE';P2:=REG);
```

Commentaires

Procédure Filtre, utile à la procédure standard ENTREE, nous la simulons en la positionnant toujours à FAUX[6], pour pouvoir toujours faire les adjonctions.

Procédure Filtre 1 : procédure appelée par ENTREE 2 quand on trouve l'indicatif cherché, alors retour au programme appelant, pour ne pas faire d'adjonction et remise à jour du pointeur.

Procédure Ordre : génère un ordre de texte objet, le premier opérande placé dans le registre, le deuxième dépend de R1 et R2.

Déclaration de la rangée de procédures.

Comp 10 : procédure qui génère l'addition. Pour cela prend les quatre renseignements sur la pile, deux par opérandes : le premier indique la zone où se trouve l'opérande : constante, variable, registre; le deuxième l'adresse relative dans cette zone ou le registre.

Comp 12 : génère la multiplication de la même façon.

Suite du programme	Commentaires
COMP[15]=(3,2) (RANGEE(2)REP INDIC; RANGEE(1) REP IND,INFO; INDIC[1]:=P1;INDIC[2]:=P2; IND[1]:=DEC(INDIC;P3); INFO[1]:=PC;PC:=PC+1; ENTREE(TABLE CONST;IND;INFO; FILTRE); P1:='CONSTANTE';P2:=PC-1);	Comp 15 : cas d'un entier. On le place dans TABLECONST avec comme indicatif la valeur décimale, s'il n'y est déjà. On utilise les procédures standard Face DEC, ENTREE [3]. On garde le type 'constante' et son adresse.
COMP[16]=(2,2) (RANGEE(1)REP INDIC,INFO; INDIC[1]:=P2; INFO[1]:=PV ; PV:=PV+1; ENTREE(TABLE;INDIC;INFO; FILTRE); P1:='VARIABLE'; P2:=PV-1);	Comp 16 : cas d'un identificateur. On le place dans TABLE s'il n'y est déjà, au rang PV. On garde le type 'variable' et son adresse.
(RANGEE(1)REP INDIC:=('ID '); RANGEE(1)REP INFO:=(1); ENTREE(TABLE;INDIC;INFO; FILTRE); INDIC[1]:='120 ';INFO[1]:=1; ENTREE(TABLECONST;INDIC;INFO; FILTRE);	Expression fermée décrivant le contexte : elle fait l'entrée en table des symboles de 'ID' placé à l'adresse 1 de la zone variable et l'entrée en table des constantes de l'entier 120 placé aussi à l'adresse 1 de la zone constante.
PC:=PC+1 ; PV:=PV+1; GENPHRASE(10); APPLTEST(COMP))	Mise à jour des compteurs. Appels aux procédures standard MFACE: GENPHRASE à partir de la règle 10 et APPLTEST pour la rangée de procédures COMP, qui respectivement crée un ensemble de tests et exécute les procédures sur cet ensemble.
JEU D'ESSAI COMP[1]=(0,3)(3,'120',0); (6,'3517','87')	Procédure manquante 1 offrant 2 jeux d'essai : l'entier 120 à 3 caractères et l'entier 351787 à 6 caractères.

Suite du programme	Commentaires
COMP[2]=(0,2)(2,['ID','AB'])	Procédure manquante 2 : offre aussi 2 jeux d'essai, l'identificateur 'ID' puis 'AB' à 2 caractères chacun.
RESULTATS PC,PV;	Nous demandons d'écrire, à l'issue de l'interprétation de chaque "phrase de test", résultat de genphrase, les deux repères PC et PV, pointeurs à l'exécution dans les deux zones constantes et variables.

Remarque :

Il peut être intéressant de vérifier les tables ; dans l'exemple ci-dessus les procédures COMP[15]et COMP[16] utilisent les tables TABLE et TABLECONST, il ne s'agit pas de voir si l'adjonction y est faite, car ceci est fait par une procédure standard Face, mais si les indicatifs et informations placées y sont justes.

Nous créons pour cela une procédure standard de mise au point : MENTREE ayant les mêmes paramètres et la même fonction que ENTREE[3] dans Face, mais qui édite de plus la table concernée (cf. annexe 0).

III.4. LE COMPILATEUR MFACE

Il s'agit d'implémenter le langage MFACE. Pour cela nous allons évidemment utiliser le langage Face et écrire un compilateur MFACE en Face, et nous allons le concevoir comme nous l'avons vu au chapitre I. Auparavant, définissons la représentation en mémoire des informations à l'exécution d'un programme MFACE.

Un programme MFACE, comme nous l'avons décrit en début de chapitre, comprend en première partie un programme Face, puis des compléments permettant de créer les jeux d'essai. Ainsi si nous écrivons le compilateur MFACE en Face, il comprendra déjà tout le compilateur FACE en Face et des compléments concernant les deuxième et troisième parties d'un programme MFACE, c'est-à-dire une partie de grammaire supplémentaire et des procédures de rangée supplémentaires attachées aux règles de la grammaire supplémentaire.

Donc la représentation des informations Face à l'intérieur du compilateur MFACE sera la même que dans le compilateur Face : représentation des grammaires, des éléments, des repères, rangées d'éléments ou de repères, des rangées de procédures [3]. Il nous reste à définir la représentation des éléments propres à MFACE :les jeux d'essai correspondant aux procédures manquantes et la liste des résultats à écrire.

3-4-1. Représentation des informations propres à MFACE

3-4-1-1. Les procédures manquantes dans les rangées de procédures

Les jeux d'essai donnés pour chacune des procédures manquantes permettent de générer plusieurs corps de procédures pour un même numéro de règle manquante, correspondant chacun à un empilement des résultats pour un des jeux d'essai donnés: à un numéro i de règle manquante peuvent être associées k_i corps de procédures manquantes, d'où k_i étiquettes de début de corps, si l'on a k_i jeux d'essai.

Nous devons pouvoir accéder à toutes ces étiquettes lors de l'exécution de APPLTEST ou de GENAPPLTEST.

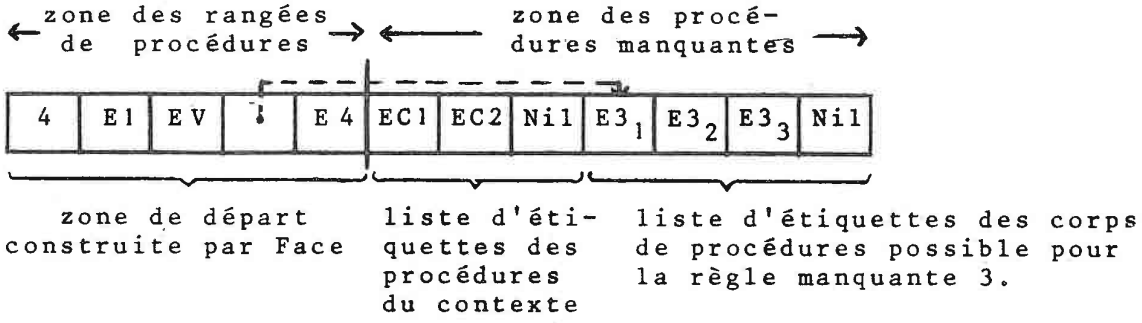
Rappelons comment sont représentées les rangées de procédures à l'intérieur du compilateur Face [3 § 2-2-3].

Chaque rangée de procédures de taille K est représentée par K + 2 mots consécutifs :

- le premier mot contient l'adresse de retour à la procédure appelante (Appl ou Anapplyse dans le cas de Face).
- le deuxième mot contient le nombre de procédures de la rangée.
- les mots suivants contiennent les adresses symboliques de début de corps des procédures de la rangée. Si a est l'adresse du deuxième mot de la zone, a+i contient l'adresse symbolique du début du corps de la procédure d'indice i si elle existe, sinon a+i contient l'adresse symbolique d'une procédure vide.
- pour un numéro de règle manquante, on ne peut pas placer à la (a+i)ème place les k_i étiquettes de corps de procédures, alors nous associerons à la (a+i)ème place la liste d'étiquettes symboliques de débuts des corps de procédures, placée dans une autre zone.
- de plus, nous avons vu que l'on pouvait avoir une pile à initialiser (résultats provenant du contexte cf. § 2-4-2 et 3-1-4) ; pour réaliser cette initialisation, nous créons une procédure, comme une procédure de rangée, qui sera appelée avant les appels des procédures de rangée guidés par la représentation postfixée de la ramification. Cette procédure, liée à la rangée puisqu'elle est liée au contexte, a aussi plusieurs jeux d'essai (cf. 3-1-4), et la liste d'étiquettes des débuts de corps de procédures, qui lui sont associées, sera placée juste au début de la nouvelle zone.

Nous allons mieux voir sur un exemple la représentation de la rangée de procédures :

```
RANGEE(4) PROC RP
     RP[1] = ...  ;
     RP[4] = ...  ;
     RP[3] = ...    Procédure manquante
```



E_i est l'adresse symbolique du début du corps de la procédure d'indice i.

EV est l'adresse symbolique d'une procédure vide.

Remarque :

Nous avons été obligés de créer une nouvelle zone pour les étiquettes des procédures manquantes, car la taille de la zone des rangées de procédures est calculée dans le compilateur Face au premier passage ; et ici lors du premier passage nous ne pouvons pas connaître la taille de la zone des étiquettes associées aux procédures manquantes.

3-3-1-2. Les numéros de procédures manquantes

Appltest et genappltest doivent savoir quels sont les numéros des procédures manquantes ; en effet pour une ramification à tester elles doivent reconnaître les appels aux procédures manquantes pour en prendre alors les différents jeux d'essai possibles (cf. II-3-3). Nous allons donc créer une zone contenant la liste des numéros des procédures manquantes.

3-3-1-3. Les résultats

Ici aussi Appltest et Genappltest doivent connaître quels sont parmi les repères ou rangées de repères ceux que

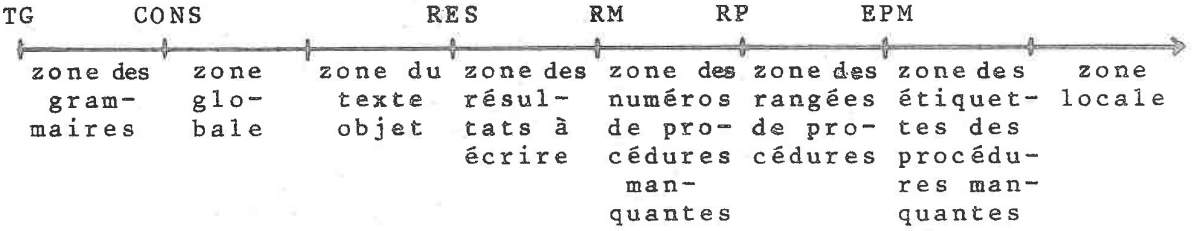
l'utilisateur a demandé d'imprimer. Pour cela nous allons créer une autre zone à l'exécution contenant les adresses,relatives à la zone globale, des repères ou rangées de repères à imprimer en fin d'interprétation de chaque phrase de test, et pour chacun d'entre eux un indicateur : 0 pour un repère et 1 pour une rangée de repères.

3-3-2. Organisation de la mémoire à l'exécution

L'organisation de la mémoire à l'exécution va être la même que celle du compilateur Face, nous allons rappeler dans l'ordre les différentes zones trouvées en mémoire [3], en y insérant les trois nouvelles décrites précédemment :

- une zone des grammaires
- une zone globale contenant les objets permanents (objets locaux)
- une zone pour le programme-objet comprenant la traduction de l'expression finale, des procédures et des procédures des rangées (y compris celle des procédures manquantes)
- une zone des résultats à écrire, contenant les adresses en zone globale
- une zone des numéros de procédures manquantes
- une zone des rangées de procédures contenant les tables représentant les rangées de procédures
- une zone desétiquettes des procédures manquantes, y compris les étiquettes de la procédure supplémentaire (concernant le contexte) (cf. § 3-3-1-1)
- une zone locale contenant les résultats intermédiaires, les objets locaux aux expressions fermées et aux procédures.

Nous obtenons un partage de la mémoire selon le schéma ci-dessous:



Exceptée la zone locale, les allocations sont toutes statiques et la taille des différentes zones sont connues à la compilation.

3-3-3. Le compilateur MFACE

Nous allons suivre ici les indications proposées dans le chapitre I à propos de l'écriture d'un compilateur : étude de la grammaire, étude du nombre et des résultats des passages, puis écriture du compilateur paragraphe par paragraphe.

3-3-3-1. La grammaire MFACE en Face

```
PROGRAMME : 'A TESTER',PGME FACE,'JEU D'ESSAI',JEUX D'ESSAI
     DONNES,'RESULTATS',LISTE DE RESULTATS|400|'A TESTER',
     PGME FACE,'JEU D'ESSAI',JEUX D'ESSAI DONNES|404|
     'A TESTER',PGME FACE,'RESULTATS',LISTE DE RESULTATS|401;
JEUX D'ESSAI DONNES : PILE DE DEPART,LISTE DE PROC.MANQUANTES||
     LISTE DE PROC.MANQUANTES|;
PILE DE DEPART : EN TETE DE PILE,LISTE DE JEUX D'ESSAI|;
EN TETE DE PILE : ID,'=(',DECIMAL,')'|402 ;
LISTE DE PROC.MANQUANTES : PROC.MANQUANTE||PROC.MANQUANTE,
     LISTE DE PROC.MANQUANTES|;
PROC.MANQUANTE : EN TETE, LISTE DE JEUX D'ESSAI|413 ;
EN TETE : ID,'[',DECIMAL,']=(',DECIMAL,',',DECIMAL,')'|403;
LISTE DE JEUX D'ESSAI : '(',JEU D'ESSAI,')'|405|'(',JEU D'ESSAI,
     ');'LISTE DE JEUXD'ESSAI|406;
JEU D'ESSAI : VALEURS D'ESSAI||VALEURS D'ESSAI,',',JEU D'ESSAI|;
VALEURS D'ESSAI : ELEM|409|'[',LISTE,']'|;
LISTE : ELEM|411|LISTE,',',ELEM|412;
ELEM : CHAINE FERMEE|415|DECIMAL|414;
LISTE DE RESULTATS : ID|416|LISTE DE RESULTATS,',',ID|417;
PGME FACE : CO GRAMMAIRE FACE EN FACE, CONTENANT ID, DECIMAL,
     ELEM,CHAINE FERMEE CO
```

3-3-3-2. Définition des passages et de leurs résultats

Comme le compilateur Face, le compilateur MFACE se décrit en deux passages, car il va contenir le compilateur Face.

Le premier passage du compilateur MFACE correspond au premier passage de Face, il réalise essentiellement une étude lexicographique du programme source (sauf celle des identificateurs locaux), et l'analyse syntaxique. Les résultats de ce passage sont les mêmes que pour le compilateur Face [3], nous allons les énumérer :

- l'entrée en TABLE de tous les identificateurs des déclarations globales
- l'allocation des objets alloués par ces identificateurs
- l'entrée en TABLE 1 des symboles terminaux et non-terminaux des grammaires
- la génération des directives SYMBOL qui créeront à l'assemblage du programme-objet les constantes et réserveront les places des variables globales.

Le deuxième passage du compilateur MFACE comprend les actions du deuxième passage du compilateur Face : il effectue les allocations des identificateurs locaux et des résultats intermédiaires et réalise la génération du texte-objet en META-SYMBOL (assembleur CII Iris 80), et c'est à ce passage qu'il va

1) réaliser l'allocation en zone des étiquettes des procédures manquantes de chacune des étiquettes générées
2) réaliser l'allocation de la zone des numéros des règles manquantes et de celle des résultats à imprimer
3) générer le texte-objet correspondant aux divers corps des procédures manquantes
4) générer la zone-objet des numéros des règles manquantes
5) générer la zone-objet des adresses des résultats à imprimer
6) générer la zone-objet des étiquettes des procédures manquantes.

3-3-3-3. Ecriture du compilateur par paragraphes

Nous rappelons ici que les rangées de procédures associées au premier et au deuxième passage s'appelaient LEX

et GEN dans le compilateur Face. La définition du premier passage est inchangée dans MFACE, aussi la première rangée de procédures LEX reste identique dans le compilateur MFACE, et s'applique aux règles de la grammaire Face.

Nous allons uniquement compléter la deuxième rangée GEN du compilateur Face, complément s'appliquant aux règles supplémentaires de MFACE données ci-dessus. Nous les définirons évidemment par "paragraphes".

Paragraphe de génération des procédures manquantes

Programme

```
JEUX D'ESSAI DONNES : PILE DE DEPART,LISTE DE PROC.MANQUANTES|;
LISTE DE PROC.MANQUANTES : PROC.MANQUANTE||PROC.MANQUANTE,
     LISTE DE PROC.MANQUANTES|;
PROC.MANQUANTE : EN TETE,LISTE DE JEUX D'ESSAI|413;
EN TETE : ID, '[' ,DECIMAL, ']=(' ,DECIMAL, ',' , DECIMAL,
     ')'|403;
LISTE DE JEUX D'ESSAI : '(' , JEU D'ESSAI, ')'|405|'(' ,
     JEU D'ESSAI, ');' ,LISTE DE JEUX D'ESSAI|406;
JEU D'ESSAI : VALEURS D'ESSAI||VALEURS D'ESSAI, ',' ,
     JEU D'ESSAI|;
VALEURS D'ESSAI : ELEM|409| '(' ,LISTE, ')'|;
LISTE : ELEM|411|LISTE, ',' ,ELEM|412;
ELEM : CHAINE FERMEE|415|DECIMAL|414;
REP NB; REP NRM, PEPM:=0;
RANGEE(10) REP FINAL, REGMAN, Z1;
```

Commentaire

Grammaire du paragraphe, nous y avons ajouté le non-terminal ELEM, équivalent du non-terminal ELEMENT de la grammaire Face, mais qui est traité lors de la génération (deuxième passage) alors qu'ELEMENT était constitué lors du premier passage.

Déclaration des identificateurs globaux

NB : indice dans FINAL

NRM : compteur d'allocation en zone des numéros des règles manquantes

PEPM : compteur d'allocation en zone des étiquettes des procédures manquantes dans Z1

FINAL : rangée de repères qui repère pour chacunedes <valeurs d'essai>, du jeu d'essai examiné, le nombre de valeurs trouvées

REGMAN : rangée contenant les numéros des règles manquantes, il sera utilisé par la suite pour la génération de la zone des numéros de règles manquantes, on ne génèrera pas directement les ordres DATA, pour ne pas mêler la zone de texte-objet et celle des numéros de règles manquantes. Cette rangée prend peu de place dans le compilateur car il existe peu de règles manquantes pour le paragraphe

Z1 : pour les mêmes raisons que REGMAN, nous gardons dans cette rangée les étiquettes de procédures manquantes, nous génèrerons la zone grâce à ce tableau Z1.
Nous utiliserons d'ailleurs la rangée de repères Z du compilateur Face qui garde les étiquettes des procédures de rangée, avec le compteur d'allocation CG.

Programme

```
GEN[403]=(15,4)((RANGEE(2)REP INDIC;INDIC[P1]:=P1;INDIC[P2]:=P2;
                RANGEE(3)REP INF=RECHERCHE(TABLE;INDIC;B;ERREUR);
                P1:=INF[2]);(RANGEE(3)REP ALPHA;
                ALPHA[1]:=P4;ALPHA[2]:=P5;ALPHA[3]:=P6;
                TRAV:=DEC(ALPHA;P7);
                ALPHA[1]:=P8;ALPHA[2]:=P9;ALPHA[3]:=P10;
                P2:=DEC(ALPHA;P11);
                ALPHA[1]:=P12;ALPHA[2]:=P13;ALPHA[3]:=P14;
                P3:=DEC(ALPHA;P15);
                P4:='NIL'; NB:=0; Z[P1+TRAV]:=PEPM; NRM:=NRM+1;
                REGMAN[NRM]:=TRAV));

GEN[414]=(4,1)(RANGEE(3)REP ENTIER;ENTIER[1]:=P1; ENTIER[2]:=P2;
                ENTIER[3]:=P3; P1:=DEC(ENTIER;P4));
GEN[415]=(3,1)(P4>4 : ERREUR(10),() );

GEN[412]=(0,0)(FINAL[NB]:=FINAL[NB]+1);

GEN[411]=(0,0)(NB:=NB+1; FINAL[NB]:=1);

GEN[409]= GEN[411]
GEN[405]=(4,RANGEE,4)(P3≠NB : ERREUR(100),();
         GENPROC(P4;Z[P1];P2); P4:='NIL');
```

Commentaires

G 403 : - On vérifie si l'identificateur est bien celui de la rangée, et on empile l'adresse de début de la table de la rangée (P1).
- On empile les nombres décimaux α et β (P2 et P3), nombre d'arguments et nombre de résultats de la procédure manquante.
- On empile le 'Nil' permettant de créer la base d'une rangée en Pile [3], rangée des valeurs des jeux d'essai.
- On place,dans la table de la rangée Z, le pointeur sur la liste d'étiquettes.
- On place dans REGMAN le numéro de la règle manquante trouvée, en incrémentant le pointeur d'allocation NRM dans cette zone.
DEC est la procédure standard Face de conversion décimale binaire.
RECHERCHE, ERREUR sont aussi des procédures standard Face, de recherche en table, et de positionnement d'un code d'erreur ; B est une procédure auxiliaire du compilateur Face de filtrage [3].

G 414 : valeur entière : conversion de l'entier décimal que l'on empile.

G 415 : valeur "chaîne de caractère" : on vérifie si la chaîne n'a pas plus de quatre caractères.

G 412 : On incrémente de 1 le compteur de valeurs pour un élément du jeu d'essai.

G 411 et G 409 : On incrémente le nombre d'éléments à monter sur la pile à 1, et on initialise à 1 le nombre de valeurs pour cet élément.

G 405 : Fin d'un jeu d'essai, toutes les valeurs de chacun des éléments à monter ont été conservées sur la pile, on appelle alors GENPROC qui va générer les procédures manquantes possibles, pour ce jeu d'essai, on garde le nombre d'éléments à enlever de la pile (P2). On remet le 'Nil' comme base de la rangée en Pile.

Programme

```
GEN[406]=( ,RANGEE,4)(P3≠NB : ERREUR(100),();
          GENPROC(P4;Z[P1];P2); NB:=0; Z1[PEPM]:='NIL';
          PEPM:=PEPM+1; P4:=0);


GEN[413]=(4,0)(P4=0:(),(ETI:=CREETI();Z1[PEPM]:=ETI;
          PEPM:=PEPM+1;ORDRE('LW';3;' ';Z[P1];0);
          ORDRE('AI';5;' ';0-P2;0);ORDRE('B';0;'*';3;0);
          Z1[PEPM]:='NIL';PEPM:=PEPM+1);
```

Paragraphe de la pile de départ

```
PILE DE DEPART : EN TETE PILE,LISTE DE JEUX D'ESSAI||VIDE|407;
EN TETE PILE : ID, '=(', DECIMAL, ')'|402;

GEN[402]=(7,4)((RANGEE(2)REP INDIC;INDIC[1]:=P1;INDIC[2]:=P2;
          RANGEE(3)REP INF=RECHERCHE(TABLE;INDIC;B;ERREUR);
          P1:=INF[2]; P2:=0); (RANGEE(3)REP ENTIER; ENTIER[1]:=P4;
          ENTIER[2]:=P5;ENTIER[3]:=P6;P3:=DEC(ENTIER;P7);
          P4:='NIL'; NB:=0);


GEN[407]=(0,0)(Z1[PEPM]:=Z[P1]; Z1[PEPM+1]:='NIL'; PEPM:=PEPM+2);
```

Commentaires

G 406 : Fin des jeux d'essai pour une procédure manquante : on vérifie si le nombre d'éléments à monter sur la pile est celui qui avait été donné dans l'entête. On génère les procédures pour le dernier jeu d'essai. On place le 'Nil' en fin de la liste d'étiquettes générées pour ce numéro de règle manquante. On reinitialise NB à 0 et P4 compteur d'éléments à 0.

G 413 : Fin d'une procédure manquante : s'il n'y a pas eu de jeu d'essai (P4 = 'Nil') on génère une procédure manquante qui supprime les arguments de la pile, et on place son étiquette en zone des étiquettes des procédures manquantes.

G 402 : En-tête de la procédure 'supplémentaire' décrivant le contexte :
- on vérifie si l'identificateur est bien celui de la rangée ;
- comme pour les procédures manquantes, on empile l'adresse de la table de la rangée ; le nombre d'arguments à enlever de la pile, ici 0, le nombre de résultats à empiler (P3) et on place le 'Nil' de base de rangée en pile [3].

G 407 : Cas où il n'y a pas de pile au départ du paragraphe : pour des raisons d'uniformité, on place comme étiquette, celle de la procédure vide, et le 'Nil' juste derrière dans Z1, table des étiquettes.

Programme

Paragraphe de la liste des résultats

```
LISTE DE RESULTATS : ID|416|LISTE DE RESULTATS, ',' , ID|417;

GEN[416]=(3,0)(ERR≠0;(),ARRT();REGMAN[1]:=NRM;
        RANGEE(2) REP INDIC;INDIC[1]:=P1; INDIC[2]:=P2;
        RANGEE(3) REP INF=RECHERCHE(TABLE;INDIC;B;ERREUR);
        ETI:='RES';ORDRE('DATA';0;' ';INF[2];0);ETI:=' ';
        DECOMP(2;INF[1])='RANGEE DE REPERES';
        ORDRE('DATA';0;' ';'1';0),ORDRE('DATA';0;' ';'0';0));



GEN[417]=(3,0)(RANGEE(2) REP INDIC;INDIC[1]:=P1; INDIC[2]:=P2;
        RANGEE(3) REP INF=RECHERCHE(TABLE;INDIC;B;ERREUR);
        ORDRE('DATA';0;' ';INF[2];0);
        DECOMP(2;INF[1])='RANGEE DE REPERES':ORDRE('DATA';0;
        ' ';'1';0),ORDRE('DATA';0;' ';'0';0));
```

Paragraphe final

```
PROGRAMME :'A TESTER',PGME FACE,'JEU D'ESSAI',JEUX D'ESSAI
        DONNES,'RESULTATS',LISTE DE RESULTATS,';'|400|
        'A TESTER',PGME FACE,'RESULTATS',LISTE DE RESULTATS,
        ';'|401|
        'A TESTER',PGME FACE,'JEU D'ESSAI',JEUX D'ESSAI DONNES|
        404;

GEN[400]=(0,0)(ORDRE('DATA';0;' ';'NIL ';0);
        ETI:=' RM';ORDRE('EQU';0;' ';0;'$';0);
        ECRIRANG(REGMAN;NRM);
        ETI:='RP';ORDRE('EQU';0;' ';'$';0);ECRIRANG(Z;CG-1);
        ETI:='EPM';ORDRE('EQU';0;' ';'$';0);ECRIRANG(Z1;PEPM-1);
        ORDRE('END';0;' ';'DEB';0));
```

Commentaires

G 416 : Début de la liste des résultats

Fermeture du fichier texte objet et on complète REGMAN avec sa taille, on génère l'étiquette de début de la zone des résultats (RES), puis l'ordre DATA avec l'adresse relative du repère ou de la rangée de repères, trouvée en Table. Si c'est un repère on génère ensuite 'DATA 0', sinon 'DATA 1'.

La procédure DECOMP(i,A) est une procédure standard Face, qui prend le ième octet à l'intérieur du mot A.

G 417 : Nouveau résultat à imprimer, on génère les deux DATA comme précédemment.

G 400 : Fin dé génération

- on génère un 'Nil' en fin de liste des résultats ;
- on recopie le tableau REGMAN, grâce à la procédure standard ECRIRANG, la zone sera étiquetée par ' RM' ;
- on recopie le tableau Z, la zone sera étiquetée par 'RP' puis le tableau Z1, la zone sera étiquetée par 'EPM' ;
- génération de la directive SYMBOL END.

Programme

```
GEN[401]=(0,0)(ORDRE('DATA';0;' ';'NIL ';0);
          CG=1(),(ETI:='RP';ORDRE('EQU';0;' ';0;'$';0);
          ECRIRANG(Z;CG-1);ORDRE('END';0;' ';'DEB';0)));


GEN[404]=(0,0)(REGMAN[1]:=NRM;ETI:='RES';ORDRE('DATA';0;' ';
         'NIL ';0);
          ETI:=' RM';ORDRE('EQU';0;' ';0;'$';0);ECRIRANG(REGMAN;NRM);
          ETI:=' RP';ORDRE('EQU';0;' ';0;'$';0);ECRIRANG(Z;CG-1);
          ETI:='EPM';ORDRE('EQU';0;' ';0;'$';0);ECRIRANG(Z1;PEPM-1);
          ORDRE('END';0;' ';'DEB';0));
```

Reprenons la règle de grammaire, de la grammaire Face concernant le programme, car la procédure de la rangée GEN qui lui est associée, sera changée.

```
PGME FACE : DEFINITION DES GRAMMAIRES, LISTE DE DECLARATIONS,
             EXPRESSION FERMEE|120;


GEN[120]=(5,0) P1 = sans résultat : (), ERREUR(19);
```

Commentaires

G 401 : - Génération du 'Nil' en fin de liste des résultats
- S' il y a eu des rangées de procédures (CG≠1) on recopie le tableau Z
- Génération de la directive Symbol End.

G 404 : - Cas où il n'y a pas de résultats à écrire, on génère cependant l'étiquette 'RES' avec Nil placé derrière
- Recopie des tableaux REGMAN, Z et Z1 comme précédemment
- Génération de la directive Symbol End.

G 120 : Cette règle était la dernière dans la grammaire Face; elle s'occupait donc de la fin de la génération, alors qu'ici, elle n'est plus qu'une règle comme les autres. On vérifie que l'expression fermée finale est sans résultat.

3-3-3-4. Les procédures auxiliaires supplémentaires

GENPROC : Procédure qui génère à partir d'un jeu d'essai des textes-objets relatifs aux procédures manquantes et remplit la zone des étiquettes des procédures manquantes (cf. § 3-4-1).
Par exemple pour le jeu d'essai:(['REEL','ENTIER'],15), GENPROC va créer 2 procédures manquantes dont les corps seraient:(P1:='REEL';P2:=15) et (P1:='ENTIER';P2:=15).

```
PROC GENPROC(RANGEE(30)REP T;REP RET,A)
        (RANGEE(10)REP ETAT;REP J:=1;INIT(ETAT;J;NB);
        ECR PROC(T;ETAT;RET;A);I:=1;POSSPROC(T;ETAT;RET;A;I);
        ML:=A>NB:A,NB;ML:=ML+1)
```

Commentaires :

T : tableau contenant les différentes valeurs des NB éléments qui doivent être empilés par la procédure manquante.
RET : adresse-retour des procédures manquantes à créer.
A : nombre d'arguments à enlever de la pile.

Pour prendre tous les jeux d'essai pour les NB éléments à empiler, nous allons créer un tableau ETAT, qui sera initialisé à 1 : ETAT[I] pointera sur la valeur envisagée pour l'élément I. Nous ferons donc varier ETAT, élément par élément, jusqu'au moment où ses éléments deviennent égaux à FINAL (cf. 3-3-3-1). Initialisation de ETAT à 1, grâce à la procédure INIT, puis écriture de la procédure ainsi choisie par ECRPROC ; recherche d'une nouvelle possibilité entre les différentes valeurs d'essai entre les éléments grâce à la procédure POSSPROC. Initialisation de ML au SUP(A,NB).

POSSPROC :Procédure qui va envisager toutes les variations des jeux d'essai pour les NB éléments, grâce à ETAT qui contient le rang de la valeur envisagée pour chacun des éléments et à FINAL qui contient le nombre de valeurs pour chacun des éléments. C'est une procédure récursive qui a pour argument le pivot I.

```
PROC POSSPROC(RANGEE(30)REP T;RANGEE(10)REP ETAT;REP RET,A,I)
        I>NB:(),ETAT(I)=FINAL(I):POSSPROC(T;ETAT;RET;A;I+1),
        (ETAT(I):=ETAT(I)+1;J:=1;INIT(ETAT;J;I-1);
        ECRPROC(T;ETAT;RET;A);POSSPROC(T;ETAT;RET;A;1))
```

Commentaires :

I étant le pivot, si pour le Ième élément nous considérons la dernière valeur (ETAT(I) = FINAL(I)), on passe à l'élément suivant I+1, sinon on prend la valeur suivante pour le Ième élément (ETAT(I):= ETAT(I)+1), on reinitialise alors les éléments précédant le Ième à leurs premières valeurs et on repart avec le pivot 1.

ECRPROC : Procédure qui génère une procédure manquante en prenant comme valeurs dans T pour les NB éléments, les valeurs repérées par le tableau ETAT.

```
PROC ECRPROC(RANGEE(30)REP T;RANGEE(10)REP ETAT;REP RET,A)
        (REP J;ETI:=CREETI();Z1[CG]:=ETI;CG:=CG+1;
        ORDRE('LW';'3';' ';0;RET;0);ETI:=' ';ORDRE('AI';
        '5';' ';0-A;0);
        R:=REGLIB();ORDRE(' ';R;' ';0;0);J:=1;
        BOUCLEL(J;ETAT;R;T);LIBERER(R);ORDRE('B';0;'*';0;3;0))
```

Commentaires :

- Création de l'étiquette de début de corps de procédure ; on la place dans Z1 dans la liste d'étiquettes (cf. § 3-3-2-1). On charge l'adresse de retour, on se positionne en dessous des éléments à éliminer dans la zone locale (zone où se fait le passage des paramètres empilés). On appelle une procédure qui fait une itération sur les NB éléments en générant la mise en zone locale de la valeur pour chacun. Retour à l'adresse indiquée.

BOUCLEL : Procédure qui place en zone locale la valeur du Jème élément repérée par ETAT[J], l'index en zone locale est incrémenté au fur et à mesure.

```
PROC BOUCLEL (REP J;RANGEE(10)REP ETAT;REP R;RANGEE(30)REP T)
        J>NB:(),(REP REG:=REGLIB();ORDRE('LW';REG;' ';
        ETAT[J];0);
        ORDRE('AW';REG;0;R;0);ORDRE('LW';REG;' ';T[REG];0);
        ORDRE('STW';REG;'*';'5';0);ORDRE('AW';R;0;FINAL[J];0);
        ORDRE('AI';'5';' ';'1';0);BOUCLEL(J+1;ETAT;R;T))
```

Commentaires :

La séquence d'ordres générés pour une valeur d'un élément est la suivante, sachant que les paramètres sont communiqués par la zone locale, dont le pointeur est le registre 5 :

LW,REG	ETAT[J]
AW,REG	R
LW,REG	T[REG]
STW,REG	*5
AW,R	FINAL[J]
AI,5	1

INIT : Procédure qui initialise à 1 le Ième élément, elle est récursive et le dernier élément qu'elle initialise est le Nème.

```
PROC INIT(RANGEE(10)REP T;REP I,N)
        I>N:(),(T(I):=1;INIT(T;I+1;N))
```

III.5. LA PROCEDURE STANDARD APPLTEST

Nous allons dans cette partie nous intéresser seulement à la procédure APPLTEST, nous étudions GENPHRASE,l'autre procédure standard,dans le chapitre suivant; quant à GENAPPLTEST, elle sera donnée dans l'annexe 1, car elle est une combinaison des deux procédures.

APPLTEST est une procédure permettant le test des procédures de la rangée, citées dans le paragraphe ; pour cela elle applique les procédures de la rangée, suivant l'ordre donné par des rangées d'entiers, appelées "phrases de tests" (cf. § II-3-3), représentant les ramifications sous forme postfixée et engendrées par GENPHRASE. Elle considèrera l'une après l'autre les rangées d'entiers fournies par GENPHRASE ; de plus elle envisagera les différentes possibilités entre les procédures manquantes citées dans la rangée d'entiers considérée (cf. § 2-3-3), et écrira la pile finale (résultats de l'exécution de la dernière procédure citée dans la rangée d'entiers) et les repères et rangées de repères demandés par l'utilisateur.

3-5-1. Etude de Appltest

APPLTEST a donc pour paramètre la rangée de procédures à tester, et a accès aux chaînes (rangées d'entiers), représentations postfixées des ramifications, résultats de GENPHRASE.

Rappelons qu'à un numéro de règle manquante peuvent être associésplusieurs procédures manquantes : elles permettent d'obtenir les jeux d'essai du paragraphe (cf. 2-3-1 et 3-1-4). Pour cela, APPLTEST va associer, à un numéro de règle manquante, la liste des étiquettes de début de corps de procédures empilant les diverses valeurs envisagées.

APPLTEST peut être décrit par :

Pour toute rangée d'entiers du fichier créé par GENPHRASE faire:
    début
      Construction du tableau TAPPLREG grâce aux numéros des règles manquantes, cités dans la rangée.
      Pour chaque combinaison des procédures manquantes entre elles faire :
        début
          Appel des procédures de la rangée dans l'ordre indiqué par la rangée d'entiers
          Ecriture des résultats demandés par l'utilisateur
        fin
    fin

Exprimons cet algorithme de façon plus précise, en considérant les zones d'exécution comme des tableaux, les identificateurs utilisés seront :

CHAINE : rangée de repères désignant une rangée d'entiers représentant une ramification.

NE : taille de CHAINE (la fin du fichier créé par GENPHRASE sera indiqué par une rangée d'entiers de taille nulle).

EPM : tableau contenant les étiquettes des corps des procédures manquantes (correspondant à une zone créée pour l'exécution cf. 3-4-3).

RM : tableau contenant les numéros des règles manquantes correspondant à la zone créée(cf. § III 4-3).

TAPPLREG:rangée de repères contenant la liste, dans l'ordre rencontré, de tous les numéros de procédures manquantes cités dans CHAINE.

ETAT : rangée de repères qui désigne pour chacun des numéros de procédures manquantes cités dans CHAINE, l'étiquette de début du corps de procédure envisagé (cette étiquette est prise dans l'ensemble des étiquettes des corps de procédures, zone créée à la compilation). Le premier élément de ETAT concerne la procédure "supplémentaire" concernant le contexte.(cf. § 2-4-2 et § 3-1-4).

```
PROC APPLTEST (RANGPROC)
     CHAINE:=LECT FICHIER();
     NE:=CHAINE[1];
     SI NE≠0 ALORS ETAT[1]:=EPM[1];
                   CONSTRUCTION DE TAPPLREG CO Soit J éléments
                       dans TAPPLREG CO
                   CO CONSTRUCTION DE ETAT:CO
                   POUR I DE 2 A J FAIRE
                       ETAT[I]:=EPM[RP[TAPPLREG[I]+2]]
                   MAPPL(CHAINE;TAPPL REG;ETAT;RANGPROC);
                   RECHERCHE POSSIBILITE(ETAT;1;J+1;RANGPROC);
                   APPLTEST(RANGPROC);
              FSI
```

CONSTRUCTION DE TAPPLREG

Module qui construit la rangée de repères TAPPLREG, contenant les appels aux procédures manquantes, à l'intérieur de CHAINE, il utilise donc CHAINE et la zone RM où se trouvent les numéros de procédures manquantes du paragraphe. L'algorithme en est simplement :

```
J:=0;
POUR I DE 1 A NE FAIRE
     DEBUT NRM:=RM[1];K:=1;
           JUSQU'A(K>NRM OU CHAINE[I]=RM[K]FAIRE K:=K+1;
           SI CHAINE[I]=RM[K] ALORS J:=J+1
                                    TAPPLREG[J]:=RM[K]
                              FSI
     FIN
```

3-5-2. Procédures utiles à APPLTEST

RECHERCHE POSSIBILITE

Procédure récursive qui considère les différentes procédures possibles pour chacun des appels aux procédures manquantes, rencontrés dans CHAINE.

Si I est le pivot, si ETAT[I] désigne la dernière étiquette,alors on passe au pivot suivant (I+1), sinon on prend l'étiquette suivante pour I et on repart de la première étiquette pour tous les K inférieurs au pivot I.

(J est toujours le nombre d'éléments de ETAT).

L'algorithme peut s'exprimer :

```
PROC RECHERCHE POSSIBILITE(ETAT;I;J;RANGPROC)
     SI I ≤ ALORS SI    EPM[ETAT[I]+1]='NIL'
                        ALORS RECHERCHE POSSIBILITE(ETAT;
                              I+1;J;RANGPROC)
                        SINON
                   DEBUT ETAT[I]:=ETAT[I]+1;
                         SI I>1 ALORS ETAT[1]:=C(EPM);
                                      POUR K DE 2 A I-1 FAIRE
                                           ETAT[K]:=EPM[RP
                                           [TAPPLREG[K]]]
                                 FSI
                         MAPPL(CHAINE;TAPPLREG;ETAT;RANGPROC);
                         RECHERCHE POSSIBILITE(ETAT;1;J;
                         RANGPROC)
                   FIN ;
```

LA PROCEDURE MAPPL

MAPPL est une procédure semblable à la procédure APPL de Face. En effet, elle considère une rangée d'entiers et applique les procédures de la rangée suivant l'ordre donné par la rangée d'entiers. Seulement, si un entier I de la chaîne (ou rangée d'entiers) est un numéro de procédure non manquante on prend l'étiquette du corps de procédure dans la zone RP à l'emplacement I [10] , mais si I est un numéro de procédure manquante on considère l'étiquette, du corps de procédure, repérée par ETAT.

C'est une procédure sans résultat dont les paramètres sont :

1) Une rangée d'entiers qui contient les numéros de procédures : CHAINE

2) Une rangée d'entiers qui cite les numéros des procédures manquantes dans la précédente rangée : TAPPLREG

3) Une rangée de repères qui pointent sur les étiquettes des procédures envisagées pour chaque appel aux procédures manquantes : ETAT

4) Une rangée de procédures : RANGPROC.

Et elle a accès à la zone de renseignements de la rangée de procédures.

Sa fonction regroupe 3 actions :

1) Charger la valeur de l'élément suivant dans la rangée de repères représentant la chaîne
2) Chercher l'adresse du corps de la procédure
3) Branchement à cette adresse.

Nous obtenons l'algorithme suivant :

```
J:=0
POUR K DE 1 A NE FAIRE
      SI CHAINE[K]≠TAPPLREG[J]ALORS J:=J+1
                                    BRANCHEMENT A L'ETIQUETTE
                                       RP[ETAT[J]]
                                 SINON BRANCHEMENT A
                                       RP[CHAINE[K]+2]
```

(RP est le tableau contenant les renseignements sur la rangée de procédures)

La programmation en Métasymbol en sera donnée dans l'annexe 1.

III.6. LE COMPILATEUR MFACE CONVERSATIONNEL

3-6-1. Avantages de MFACE

MFACE tel qu'il a été décrit dans ce chapitre permet la mise au point des paragraphes, mais il permet aussi le test de passages complets d'un compilateur écrits en Face.

Supposons que l'on veuille tester le premier passage du compilateur Algol 60, en mettant un petit programme de test. Il suffirait pour cela d'écrire le programme MFACE suivant : (l'exemple avait déjà été choisi au § 1-3-2)

Programme	Commentaires
A TESTER	
PROGRAMME:BLOC|99|ETIQUETAGE, BLOC|99;	Grammaire nécessaire au premier passage d'Algol 60.
RANGEE(500) REP CHAINE; REP ML,NO,CETI; RANGEE(15) REP ALLOC;	Déclarations globales : CHAINE : qui sera le résultat d'analyse. ML : pointeur en zone locale. NO : numéro du bloc en cours. CETI : compteur d'étiquettes. ALLOC : repère l'adresse en zone locale des blocs.
PROC ELT CREETI=() (CETI:=CETI+1;OCTET(1;CETI;'E'))	Procédure auxiliaire CREETI qui crée des étiquettes grâce au compteur CETI, et la lettre E en première place. OCTET est une procédure standard Face.
RANGEE(120) PROC LEX; LEX[1]=	Déclaration de la rangée de procédures concernant le premier passage : LEX.
(ANALYSE(1;CHAINE);ECRFICHIER (CHAINE); APPLTEST(LEX))	Expression termée qui appelle l'analyseur. On écrit la chaîne résultat sur le fichier grâce à ECRFICHIER pour appeler APPLTEST.

RESULTAT ML,NO,CETI,ALLOC,... ;	Il n'y a pas de jeux d'essai dans ce cas-là, et on demande d'écrire tous les repères et rangées de repères intéressants à l'étude lexicographique.

Nous pouvons ainsi faire des tests de compilateur Face, passage par passage. Nous pouvons aussi faire le test d'une procédure seule d'une rangée.

Par exemple, prenons la règle sur le primaire d'un compilateur Algol 60, en ne considérant qu'un identificateur. Voici le programme de test obtenu :

Programme A TESTER	Commentaires
PRIMAIRE : PRIMAIRE INDICE||IDENT| 74|APPEL PROCEDURE|; IDENT : 'A'|1;	Grammaire ayant la règle 74 dont la procédure est à tester. La règle 1 est une règle manquante.
RANGEE[2] REP CHAINE; REP ML,NO,CETI;	Déclarations globales : CHAINE contiendra la rangée d'entiers permettant d'appliquer les procédures. ML,NO,CETI : repères globaux comme précédemment.
RANGEE[74] PROC GEN; GEN[74]=(1,3) (RANGEE(1) REP INDIC; INDIC[1]:=P1 ; ⋮ P1:=INFO[1];P2:=INFO[2];P3:=0);	G74 : procédure à tester, qui recherche l'identificateur en table et place ses arguments : son type, son adresse, marque de résultat intermédiaire.

```
(RANGEE(1) REP INDIC;
INDIC[1]:='A';
RANGEE(2) REP INFO;
INFO[1]:='REEL';INFO[2]:=5;
ENTREE(TABLE;INDIC;INFO;FILTRE);
      :
CHAINE[1]:=1;CHAINE[2]:=74;
ECRFICHIER(CHAINE);
APPLTEST(GEN))
```

Expression fermée : contenant l'initialisation du contexte ; placer les 3 identificateurs A,B,C en table. Initialisation de la chaîne avec son écriture en Fichier.
Appel de APPLTEST

JEU D'ESSAI

```
GEN[1]=(0,1)(['A','B','C'])
```

Une procédure manquante de numéro 1, qui monte un résultat sur la pile : les caractères de l'identificateur : soit A, soit B, soit C.

Nous pouvons aussi faire écrire la chaîne, résultat de l'analyseur, dans une étape de vérification de la grammaire (cf. § 1-3-1-2).

3-6-2. Le langage MFACE doit être conversationnel

Nous venons de voir que la mise au point d'un paragraphe est entièrement automatique : l'exécution se fait sur l'ensemble des ramifications engendrées par GENPHRASE et en prenant tous les jeux d'essai proposés pour les procédures manquantes. Bien que le critère de choix pour la génération des ramifications soit restrictif (cf. chapitre IV) , le test de certaines ramifications engendrées n'apporte pas de renseignements supplémentaires.

Par exemple, dans l'exemple § 4-1-2, la chaîne 10 8 6 9 7 2 5 3 1 n'apporte rien de plus que la chaîne 10 8 6 8 6 2 5 3 1.

De plus, pour une ramification donnée, certains jeux d'essai des procédures manquantes sont superflus (cf. la remarque à la fin du § 2-3-3). Et pour un paragraphe important la mise en oeuvre risque d'être lourde.

Et, en fait, l'utilisateur a envie, à certains moments, d'intervenir pour choisir la "phrase de test" à interpréter. Il faut que le langage MFACE soit conversationnel pour que la mise au point puisse être guidée par l'utilisateur.

La procédure GENPHRASE permet la vérification de la grammaire et fournit ainsi un ensemble de tests possibles, engendrés par la grammaire, à l'utilisateur.

A partir de là, la mise au point deviendrait conversationnelle : l'utilisateur choisissant dans l'ensemble de tests ceux qui lui semblent intéressants, et au vu de la phrase de test choisie il donnerait au fur et à mesure ou globalement les jeux d'essai pour les procédures manquantes, intervenant dans la ramification.

CHAPITRE 4

LA PROCEDURE GENPHRASE

IV.1. INTRODUCTION

Ce chapitre a été évoqué à plusieurs reprises au cours des chapitres précédents : nous avons cité une procédure GENPHRASE qui génère

- un ensemble de mots engendrés par une grammaire donnée ; ceci dans le but de vérifier la grammaire (cf. § 1-3-1 et 3-1-2)
- un ensemble de représentations postfixées des ramifications correspondant aux mots du langage précédent, ceci pour permettre le test des procédures de rangée (cf. § 1-3-3, 2-3-3 et 3-1-2).

4-1-1. Aide à la vérification de la grammaire

Il est utile pour l'utilisateur de voir si le langage L, engendré par la grammaire G donnée, est inclus dans le langage LS que l'on veut décrire par la grammaire ($L \subset LS$, cf. 1-3-1-3). Pour cela il suffit de fournir à l'utilisateur une procédure standard qui engendre des mots de L; mais nous ne pouvons engendrer qu'un nombre fini de mots appartenant à L et l'utilisateur veut surtout obtenir une idée plus précise des mots de L.

Pour cela, nous allons lui proposer une autre procédure standard TRANSGRAM, cette procédure va transformer la grammaire en supprimant tous les non-terminaux de la grammaire qui engendrent un langage fini, il restera alors un ensemble de non-terminaux, que nous appellerons par la suite 'non-terminaux récursifs' ; à chacun de ces non-terminaux récursifs correspondront des règles de grammaire dans lesquelles nous aurons substitué aux non-terminaux qui engendrent un langage fini, leur langage engendré.

Par exemple, si nous prenons la grammaire correspondant au paragraphe de l'instruction conditionnelle d'Algol 60:

<inst>::=<instr.conditionnelle>^{1}|<autres instr.>|^2

<instr.conditionnelle>::=si<expr.bool.>alors<autres instr.>
<2ème membre>|[3]

<2ème membre>::=<vide>[4]|sinon<instr.>|[5]

<autres instr.>::=<instr.affectation>[6]|<instr.pour>|[7]

<instr.affectation>::=AFFECTATION|[8]
<instr.pour>::=INSTRUCTION POUR|[9]
} règles manquantes

La grammaire va être modifiée, il restera 2 non-terminaux récursifs : <instr.> et <instr.conditionnelle> :

<instr.>::=<instr.conditionnelle>|AFFECTATION|INSTRUCTION POUR
<instr.conditionnelle> ::=SI CONDITION ALORS AFFECTATION|
SI CONDITION ALORS INSTRUCTION POUR|
SI CONDITION ALORS AFFECTATION
SINON <instr.>|
SI CONDITION ALORS INSTRUCTION POUR
SINON <instr.>|

Les règles alors obtenues donnent à l'utilisateur une forme plus accessible de la structure des mots du langage engendré, la récursivité des règles y est toujours apparente.

Ainsi TRANSGRAM va simplifier la grammaire de départ et fournir une nouvelle grammaire, il ne sera pas nécessaire de spécifier des arguments, elle prendra pour donnée la dernière grammaire citée et fournira une nouvelle grammaire.

Nous étudierons plus en détails cette procédure TRANSGRAM dans le paragraphe 4-2.

4-1-2. Génération d'un ensemble de phrases pour le test d'un paragraphe : GENPHRASE

Il est utile de générer automatiquement des mots engendrés par une grammaire donnée, pour que l'utilisateur la vérifie (cf. § 1-3-1), il faut aussi fournir un ensemble de ramifications engendrées par cette grammaire pour tester les

procédures des rangées (cf. § 2-5-2) associées aux règles de la grammaire donnée.

Les procédures des rangées sont appelées dans l'ordre indiqué par les numéros des règles étiquetant des ramifications engendrées par la grammaire (cf. § 1-1-1). Pour mettre au point ces procédures, il faut extraire de l'ensemble des situations possibles d'appels, un ensemble de tests intéressant , c'est-à-dire tester ces procédures sur diverses ramifications engendrées par la grammaire. Il faut donc obtenir parmi les ramifications engendrées, des ramifications intéressantes, et pour ceci avoir un bon critère de choix.

Nous avons décidé de ne jamais choisir deux fois la même règle sur une même branche de ramification engendrée ; ce qui permet aisément de borner l'algorithme de génération des ramifications engendrées par G et d'obtenir "presque toutes" les situations d'appels des procédures d'une rangée à tester ; nous discutons de cet algorithme de choix dans le § IV-4. Nous fournissons ainsi à l'utilisateur un ensemble de phrases d'essai intéressantes pour tester une rangée de procédures (cf. chapitre III-1-2).

Si nous reprenons l'exemple précédent, nous obtiendrons les ramifications dont les mots des feuilles sont :

SI CONDITION ALORS AFFECTATION
SI CONDITION ALORS INSTRUCTION POUR
SI CONDITION ALORS AFFECTATION SINON AFFECTATION
SI CONDITION ALORS AFFECTATION SINON INSTRUCTION POUR
SI CONDITION ALORS INSTRUCTION POUR SINON AFFECTATION
SI CONDITION ALORS INSTRUCTION POUR SINON INSTRUCTION POUR AFFECTATION
INSTRUCTION POUR

ce qui correspond aux ramifications dont les suites de numéros de règles postfixées sont :

10 8 6 4 3 1
10 9 7 4 3 1
10 8 6 8 6 2 5 3 1
10 8 6 9 7 2 5 3 1

10 9 7 8 6 2 5 3 1
10 9 7 9 7 2 5 3 1
8 6 2
9 7 2

Dans certains cas, il y aura ainsi trop de phrases d'essai générées (cf. § III-6-2). Pour générer ces ensembles, nous ne partirons pas de la grammaire initiale mais de la grammaire simplifiée : la simplification a éliminé dans les règles tous les non-terminaux qui engendrent un langage fini ; il suffit, en partant d'un axiome donné, de dériver les non-terminaux récursifs rencontrés en ne réutilisant jamais deux fois la même règle lors d'une dérivation d'un non-terminal. GENPHRASE agira donc sur la grammaire simplifiée, mais l'utilisateur n'a pas connaissance de cette simplification préalable. GENPHRASE aura comme argument un numéro de règle d'un axiome de la grammaire.

Prenons un exemple d'une grammaire qui a deux axiomes X1 et X2, le premier numéro de règle associée à X1 est 15, le premier numéro de règle associée à X2 est 37 ; pour engendrer les mots du langage engendrés par X1 et X2, il suffit d'écrire l'expression fermée :

(GENPHRASE(15) ; GENPHRASE(37)).

Le premier appel de GENPHRASE fera lui-même l'appel du module de TRANSGRAM, de la simplification de la grammaire sans écriture de la grammaire simplifiée, la génération de mots se faisant à partir de la grammaire simplifiée.

Si l'utilisateur veut en plus l'écriture de cette grammaire simplifiée il doit alors faire appel à TRANSGRAM et il écrit alors l'expression fermée :

(TRANSGRAM() ; GENPHRASE(15) ; GENPHRASE(37)).

Dans ce cas, la procédure TRANSGRAM simplifie la grammaire et l'écrit, et les deux appels de GENPHRASE génèrent les mots et ramifications du langage en utilisant directement la grammaire simplifiée.

Ainsi l'utilisateur, appelant la procédure standard GENPHRASE à l'intérieur de l'expression fermée du test du paragraphe, obtient les ramifications engendrées par la grammaire du paragraphe, nécessaires au test des procédures de rangées du paragraphe.

Il obtient aussi l'impression des mots des feuilles, ce qui lui permet de vérifier partiellement sa grammaire ; il peut aussi appeler TRANSGRAM qui lui permet de vérifier plus complètement sa grammaire.

Nous allons ainsi dans le paragraphe suivant définir cette simplification de la grammaire, utilisée par les deux procédures TRANSGRAM et GENPHRASE.

IV-2. SIMPLIFICATION DE LA GRAMMAIRE

Comme nous venons de le voir, nous allons simplifier la grammaire en y remplaçant les non-terminaux qui engendrent un langage fini, par leur langage engendré, ce qui permet d'obtenir une grammaire qui engendrera les mêmes ramifications que la grammaire initiale.

Ainsi la simplification de la grammaire va se découper en plusieurs étapes :

1) Déterminer l'ensemble N1 des non-terminaux qui engendrent un langage fini, d'où N2 l'ensemble des non-terminaux récursifs.
2) Pour chacun des non-terminaux A de N1, il faut générer les ramifications engendrées par le non-terminal A.

3) Pour chacun des non-terminaux A de N2, nous transformerons les règles qui lui sont associées.

Auparavant nous rappelons la représentation des grammaires, telle que Face l'a adoptée [3] [15].

4-2-1. Représentation d'une grammaire

Soit une grammaire G = (T,N,::=,X), à chaque non-terminal, élément de N, doit correspondre la liste des règles dont il est premier membre.

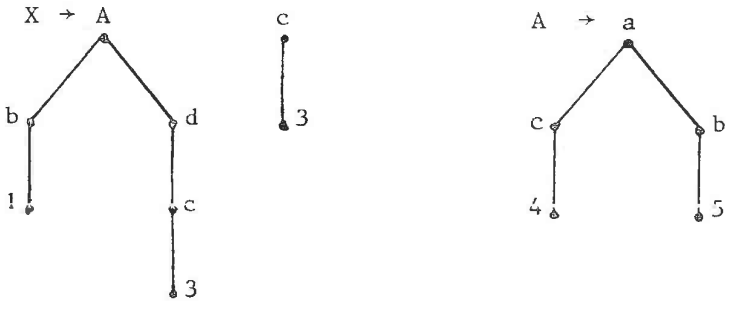
Ces règles $::=\lambda_1 |^1\lambda_2 |^2 \ldots$ sont représentées par une ramification sur (N∪T), telles que les mots λ_1, λ_2 ... de (N∪T) étiquettent les chemins joignant une racine à une feuille [14]. Les numéros des règles 1, 2, ... seront placés aux feuilles correspondantes. Dans la suite, nous noterons R l'ensemble des entiers, numéros des règles de la grammaire G donnée.

Exemple :

Soit la grammaire $X ::= Ab |^1 Adc |^2 c |^3$

$A ::= ac |^4 ab |^5$

Elle sera représentée par les ramifications :



Chaque non-terminal conduit à une ramification représentant les règles dont il est le premier membre. Les numéros des règles sont placés "aux feuilles".

En Face, la grammaire G sera représentée en mémoire par une table des non-terminaux permettant l'accès à la ramification des règles dont ils sont premiers membres. De plus, la chaîne de caractères des non-terminaux est conservée dans un tableau dit, table des non-terminaux ; si le non-terminal est situé à l'indice i dans cette table, il sera repéré par cet indice dans les ramifications représentant les règles de grammaire.

De même les terminaux sont représentés par une table des terminaux, l'indice du terminal dans cette table permet de la repérer dans les ramifications.

Les ramifications des règles sont représentées par lien vertical et horizontal [3] [15].

4-2-2. Recherche des non-terminaux qui engendrent un langage fini

Considérons le graphe (N,r) où N est l'ensemble des non-terminaux de la grammaire, et r la relation "est dans la règle de" définie par

$(\forall A, B \in N, A\ r\ B \Leftrightarrow$ B a une occurrence dans une des règles de grammaire de premier membre A)

Les non-terminaux récursifs (ceux qui n'engendrent pas de langage fini) sont des éléments A de N tels que $A\ r^+ A$ où r^+ désigne la fermeture transitive de r.

Il suffira donc de chercher la fermeture transitive de la relation r pour obtenir les ensembles N1 et N2, ensemble des non-terminaux qui engendrent un langage fini, et ensemble des non-terminaux récursifs, tous deux complémentaires dans N.

Etant donnée la grammaire, nous en déduirons le graphe de r que nous représenterons par sa matrice associée ; puis nous utiliserons un algorithme de recherche de la ferme-

ture transitive r^+, de la relation r, par un algorithme connu (par exemple celui de Warshall [5]).

Représentation :

Les ensembles N1 et N2 seront représentés par un tableau booléen NT : NT(I) = 0 si le non-terminal I appartient à N2
1 si le non-terminal I appartient à N1.

4-2-3. Génération des langages finis

GENPHRASE génère non seulement les mots mais aussi les ramifications engendrées par la grammaire, aussi nous générons, pour chacun des non-terminaux à éliminer (ceux de N1, qui engendrent un langage fini), l'ensemble de leurs ramifications, ensemble fini lui aussi, qui fournit les mots du langage en prenant les mots des feuilles.

Génération de l'ensemble des ramifications engendrées par un non-terminal de N1, sous sa forme postfixée :

Si GENRAMIF(A) est une procédure récursive, qui génère les ramifications engendrées à partir d'un non-terminal A, elle peut s'écrire :

GENRAMIF(A) : si A ∈ T alors le résultat est A
sinon pour chacune des règles i de premier membre A : $A ::= B_1 B_2 \ldots B_{qi} \mid^i$ faire
début
GENRAMIF(B_1) : Soit r_1 un des k_1 résultats
⋮
GENRAMIF(B_{qi}) : Soit r_{qi} un des k_{qi} résultats ;
d'où K = K1×K2×...×Kqi résultats s'écrivant sous la forme postfixée : $r_1 \cdot r_2 \ldots r_{qi} \cdot i$
fin

où $r_1 \cdot r_2$ indique la concaténation entre les deux chaînes représentant les ramifications sous forme postfixée. Il faudra appliquer GENRAMIF à tous les éléments de N1.

On peut décrire aussi cette procédure en tenant compte de la représentation de la grammaire comme nous l'avons décrite au paragraphe 4-2-1 sous forme d'une fonction récursive de ramification [14].

Soit donc la fonction F, qui a pour argument la ramification issue de A dans la représentation de la grammaire G, et pour résultat, l'ensemble des ramifications engendrées par A, toujours sous forme postfixée :

si ramif est la fonction qui, à tout non-terminal $B \in N$, associe la ramification qui le représente, et si nous notons $(A \times s + t)$ la ramification représentant les règles issues de B, la fonction F se décrit alors :

$$
\begin{aligned}
F(A \times s + t) = \text{si } s = \Lambda \text{ et } t = \Lambda \text{ et } A \in (T \cup R) &: A \\
\text{et } A \in N &: F(\text{ramif}(A)) \\
\text{si } s = \Lambda \text{ et } t \neq \Lambda \text{ et } A \in T &: A \cup F(t) \\
\text{et } A \in N &: F(\text{ramif}(A)) \cup F(t) \\
\text{si } s \neq \Lambda \text{ et } t = \Lambda \text{ et } A \in T &: F(s) \cdot A \\
\text{et } A \in N &: F(s) \cdot F(\text{ramif}(A)) \\
\text{si } s \neq \Lambda \text{ et } t \neq \Lambda \text{ et } A \in T &: F(s) \cdot A \cup F(t) \\
\text{et } A \in N &: F(s) \cdot F(\text{ramif}(A)) \cup F(t)
\end{aligned}
$$

où . est toujours la concaténation entre deux suites, ou deux ensembles de suites (correspondant à l'enracinement de ramifications). U est la réunion d'ensembles entre ensemble-résultats issus du même non-terminal (correspondant à la concaténation de deux ramifications).

Si nous voulons uniquement les mots du langage, il suffit de se restreindre aux terminaux, dans les représentations postfixées des ramifications.

4-2-4. Représentations internes des représentations postfixées engendrées

D'après la description de la fonction GENRAMIF (cf §4-2-3), nous voyons que nous devons représenter en mémoire toutes les "chaînes" r_i , qui sont des parties de la représentation postfixée d'une des ramifications résultats, et que nous appellerons par la suite "morceaux". Pour obtenir une représentation correcte des k résultats, il suffit de considérer la relation successeur $\underline{s}$ entre ces morceaux r_i, telle que $r_i \ \underline{s} \ r_j$, si et seulement si r_i appartient à l'ensemble des résultats de GENRAMIF (B_i) et r_j à l'ensemble des résultats de GENRAMIF(B_{i+1}), de la règle $A::=B_1 \ldots B_i \ B_{i+1} \ldots B_{qk}|^k$ pendant l'appel de GENRAMIF(A).

Ainsi nous obtenons un graphe, pour cette relation $\underline{s}$ entre les morceaux de représentations postfixées, représentant le résultat de GENRAMIF.

Exemple :

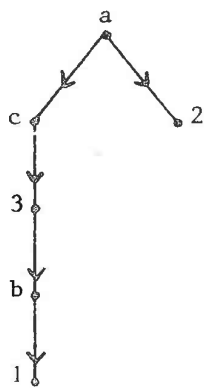
1) Prenons tout d'abord une grammaire simple, dont tous les non-terminaux engendrent un langage fini :

$A::=aBb|^1 a|^2$

$B::=c|^3$

Nous obtenons les morceaux de représentation postfixée :

pour A :



pour B :

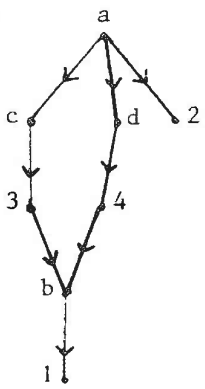
2) Prenons une autre grammaire :

A::=aBb|[1] a|[2]

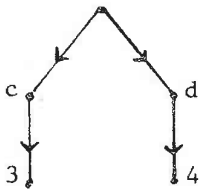
B::=c|[3] d|[4]

Nous obtenons pour chacun des non-terminaux A et B, éléments de NI, les graphes suivants :

pour A :



pour B :



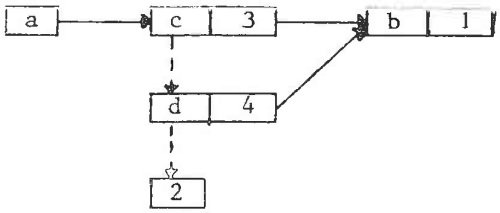
Nous représenterons les graphes obtenus, d'une façon habituelle, par des liens successeur et alternative de chacun des éléments, et en considérant comme élément un morceau de représentation postfixée.

Exemple : reprenons l'exemple précédent, dont la grammaire est A::=aBb|[1] a|[2]

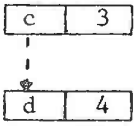
B::=c|[3] d|[4]

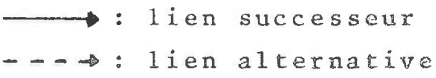
Voici les deux graphes obtenus, tels qu'ils seront représentés :

pour A :



pour B :





Nous modifions alors le tableau NT (cf § 4-2-2) décrivant les ensembles N1 et N2 : pour I $\in$ N1, NT(I) pointe sur le graphe décrivant le langage engendré par le non-terminal I. Nous allons maintenant pouvoir définir l'algorithme de simplification de la grammaire.

4-2-5. Transformation de la grammaire

Nous voulons simplifier la grammaire, en y supprimant tous les non-terminaux qui engendrent un langage fini (éléments de N1).

Soit G=(X,N,T,::=) la grammaire de départ, nous obtenons une grammaire G'=(X,N2,T',::=')

où
- X est toujours l'axiome de la grammaire, il appartient toujours à N2, si N2 n'est pas vide ; c'est-à-dire qu'il n'engendre pas de langage fini
- N2 est l'ensemble des non-terminaux récursifs (N=N1 $\oplus$ N2)
- T' = T $\cup$ R, car apparaissent dans les règles, des numéros de règles, dus aux substitutions des éléments de N1 de manière à conserver les représentations postfixées
- ::=' est la restriction de ::= aux éléments de N2.

Si nous représentons cette simplification par une fonction TRANSF, TRANSF peut s'écrire :

```
TRANSF(A) : si A ∈ N2 alors pour chaque règle i de premier
                            membre A parmi les règles
                            A::=B1 B2 ... Bqi |i faire :
                            pour les qi éléments Bj faire
                             si Bj ∈ N1 alors substituer Bj
                                        par son lan-
                                        gage fini fsi
            fsi
```

Ici aussi, nous pouvons décrire cette transformation de la grammaire sur les ramifications décrivant la grammaire, si SIMPL est la fonction qui à partir des ramifications représentant la grammaire, donne les ramifications représentant la grammaire simplifiée mais sous forme postfixée, SIMPL s'écrit :

SIMPL(A×s+t) : si s = Λ et t = Λ et A ∉ Nl : A
et A ∈ Nl : le résultat est F(ramif(A))
si s = Λ et t ≠ Λ et A ∉ Nl : A R SIMPL(t)
et A ∈ Nl : résultat de F(ramif(A)) R SIMPL(t)
si s ≠ Λ et t = Λ et A ∉ Nl : SIMPL(s) . A
et A ∈ Nl : SIMPL(s) . résultat de F(ramif(A))
si s ≠ Λ et t ≠ Λ et A ∉ Nl : SIMPL(s) . A R SIMPL(t)
et A ∈ Nl : SIMPL(s) . résultat de F(ramif(A)) R SIMPL(t)

où . et R sont les relations décrites précédemment au § 4-2-3, et F et ramif les fonctions décrites aussi au § 4-2-3.

4-2-6. Représentation de la grammaire simplifiée

La grammaire-argument est représentée par une ramification (cf. § 4-2-1), à l'intérieur de cette ramification, nous substituons chaque non-terminal qui engendre un langage fini, par l'ensemble des ramifications sous forme postfixée qu'il engendre, et les règles sont transformées sous forme de morceaux de représentations postfixées.

Reprenons un exemple, voici la grammaire G donnée :

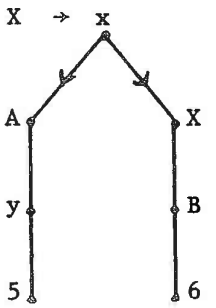
X ::= x A y | [5] x X B | [6]

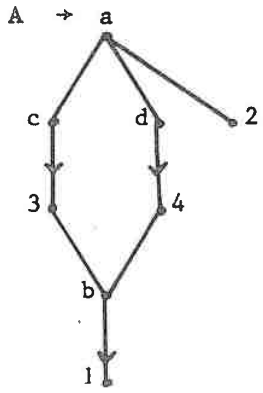
A ::= a B b | [1] a | [2]

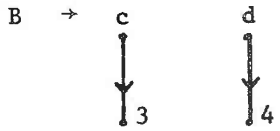
B ::= c | [3] d | [4]

où A et B sont des non-terminaux qui engendrent un langage fini.

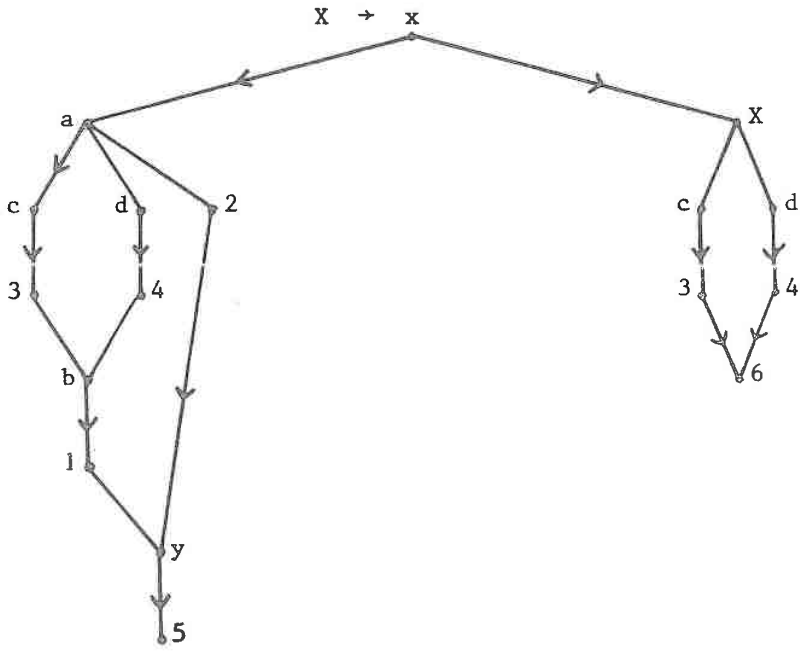
Voici la ramification concernant X, et les graphes relatifs à A et B avant la simplification de la grammaire :







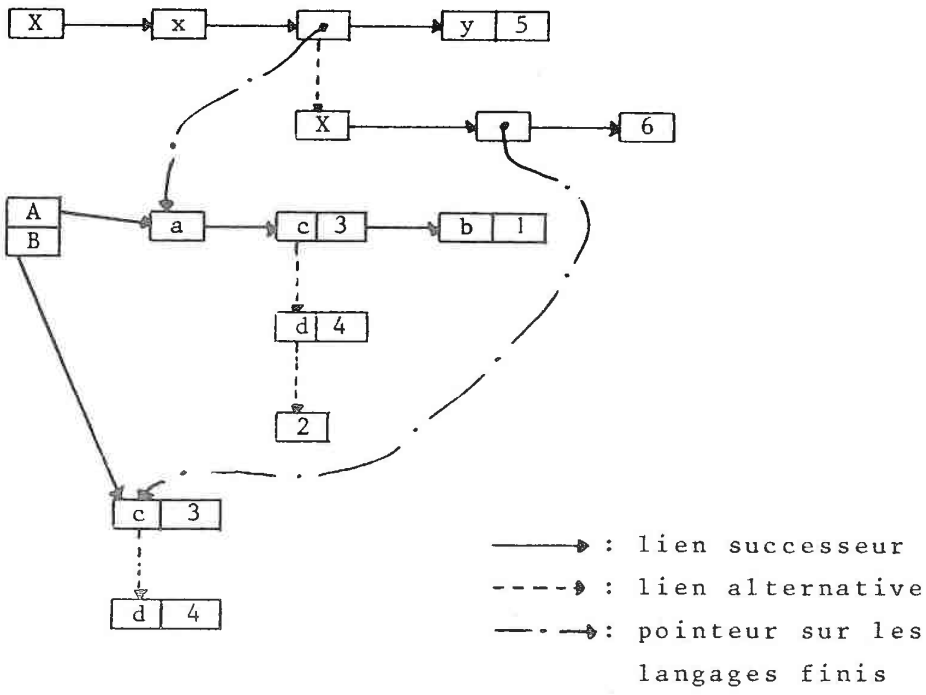
ce qui conduirait au graphe suivant, après la simplification :



Pour éviter une perte de place trop importante, nous ne faisons pas les recopies effectives des langages finis à l'intérieur de la grammaire, car cela demanderait trop de place en mémoire, mais alors cela nécessite un parcours, pour TRANSGRAM et GENPHRASE, dans le graphe de la grammaire et dans le graphe des langages finis.

Ici aussi nous modifions le tableau NT, des non-terminaux ; pour chacun des non-terminaux I de N2, NT(I) pointe sur le graphe représentant l'ensemble de ses règles de grammaire ; et le graphe est représenté par les liens successeur et alternative pour chacun des morceaux des représentations postfixées.

D'où dans l'exemple précédent, pour le seul non-terminal récursif X , le graphe :



4-2-4. La procédure TRANSGRAM

TRANSGRAM est donc la procédure standard qui effectue la simplification de la grammaire, telle que nous venons de la

décrire, après avoir recherché les ensembles N1 et N2, et engendré les ramifications issues des éléments de N1 ; mais elle doit aussi écrire cette grammaire simplifiée, c'est-à-dire pour chacun des non-terminaux A récursifs de N2, écrire la liste des règles de premier membre A.

Pour cela, pour chacun de ces non-terminaux récursifs A, il suffit de parcourir les chemins de son graphe associé, chacun des chemins trouvés décrit une règle de premier membre A.

Il suffit d'utiliser un algorithme de recherche de tous les chemins dans un graphe, et pour chacun des points de ce graphe, qui est un morceau de représentation postfixée des ramifications, nous en imprimons les éléments terminaux.
(Des résultats relatifs à cette procédure seront donnés dans l'annexe 2.)

IV-3. GENERATION D'UN ENSEMBLE DE TESTS : GENPHRASE

GENPHRASE est une procédure qui génère un ensemble de mots engendrés par la grammaire donnée et un ensemble de ramifications, correspondant aux mots de l'ensemble précédent ; en fait nous allons générer l'ensemble des ramifications engendrées par la grammaire, d'où nous tirerons les mots du langage en prenant les éléments de T, et les phrases de tests pour le paragraphe en prenant les éléments de R. Le critère choisi pour borner l'algorithme est de ne jamais prendre deux fois la même règle sur une même branche de ramification.

Nous allons décrire l'algorithme de génération de ces ramifications.

4-3-1. Algorithme de génération des ramifications

Si GENTEST(A) est la procédure qui génère les ramifications sous forme postfixée engendrées par un non-terminal A de la grammaire simplifiée G, GENTEST s'écrit :

GENTEST(A) : si A $\notin$ N2 alors le résultat est A
sinon début
- choix d'une règle K de premier membre A dans G
  A ::= $B_1 B_2 \ldots B_q \mid^k$
- interdiction de reprendre la règle K
- GENTEST(B_1) : soit $ramif_1$ l'un des résultats
  ⋮
- GENTEST(B_q) : soit $ramif_q$ l'un des résultats
- un des résultats est : $ramif_1$ . ... $ramif_q$. k
- restauration de la règle K

fin

fsi

Pour générer l'ensemble des ramifications voulues, il faudra prendre successivement tous les choix possibles, selon le critère de choix ci-dessus, en partant de l'axiome voulu.

Pour exprimer le choix de la règle, puis l'interdiction de la prendre, et sa restauration, nous allons utiliser une pile : PILE-CHOIX, dans laquelle nous empilons les règles choisies ; pour choisir une nouvelle règle, il suffira qu'elle ne soit pas en PILE-CHOIX.

Ainsi, nous allons transformer l'algorithme en y introduisant cette pile, et en y éliminant la récursivité, ce qui nécessite une nouvelle pile PILE-RETOUR, conservant les non-terminaux récursifs rencontrés, pour reprendre la règle où on en était resté lors de la dérivation de ce non-terminal.

Voici l'algorithme :

```
A:=X;
BOU:si A ∉ N2 alors le résultat est A
                sinon début.pour K de 1 à nombre de règles
                                de A faire :
                                choix de K tel que K non
                                en PILE-CHOIX
                            .EMPILER PILE-CHOIX(A,K)
                            .pour tous les Bi de la k ième règle
                                A::=B1.B2...Bq|k faire début
                                si Bi ∈ T alors résultat
                                                est Bi
                                          sinon EMPILER
                                                PILE-RETOUR
                                                (Bi,K)
                                                A:=Bi
                                                allera BOU
                                RETOUR:
                                                        fin
                            .résultat est K
                            .DEPILER PILE-CHOIX(A,K)
                    fin
    si PILE-RETOUR non vide alors restaurer(Bi,K)
                                allera RETOUR
                        fsi
```

Cet algorithme génère comme GENRAMIF et GENTEST un ensemble de ramifications engendrées par le non-terminal ; c'est-à-dire, comme nous l'avons étudié pour GENRAMIF, nous obtiendrons un graphe, dont tous les chemins sont des ramifications engendrées par le non-terminal.

Plutôt que de construire ce graphe, représentant l'ensemble des ramifications, ce qui oblige à le parcourir ensuite pour générer les ramifications, nous allons retransformer l'algorithme pour générer directement une à une les ramifications.

De plus, si nous construisons le graphe des ramifications, cela nécessite plus de place en mémoire, en effet la grammaire simplifiée et le graphe devraient être présents lors de la construction de ce dernier.

4-3-2. Algorithme de génération des ramifications une à une

Si nous générons les ramifications une à une, nous devons obliger l'algorithme à n'en construire qu'une en faisant le choix d'une règle de premier membre A ; et après en avoir construit une, en construire une autre le plus simplement possible, pour cela nous regardons si la dernière règle utilisée n'avait pas une alternative, ou sinon l'avant dernière, etc... Ce qui va nous conduire à une pile conservant tous les B_i par lesquels nous passons : PILE-CHEMIN, et une dernière permettant de reformer le résultat à partir du résultat précédent : PILE-RES.

PILE-CHEMIN doit permettre aussi de reconstituer PILE-CHOIX et PILE-RETOUR en fonction de l'alternative choisie. L'algorithme devient alors :

```
K1 := 1;
A := X;
BOU: si A ∉ N2 alors EMPILER PILE-RES(A)
                     EMPILER PILE-CHEMIN(A,K1)
                sinon début
                     . EMPILER PILE-CHEMIN(A,K1)
                     . choix d'une règle K à partir de
                       K1, non en PILE-CHOIX, et de
                       premier membre A
                     . EMPILER PILE-CHOIX(A,K)
                     . pour tous les Bi de la kième règle
                       faire
                          début si Bi ∈ T
                                alors EMPILER
                                      PILE-RES(Bi)
                                sinon EMPILER
                                      PILE-RETOUR
                                      (Ri,K);
                                      A := Bi; K1 := 1;
                                      allera BOU
```

```
                                    RETOUR:
                                                fsi
                                            fin
                                . EMPILER PILE-RES(K)
                                . DEPILER PILE-CHOIX(A,K)
                                fin
                    si PILE-RETOUR non vide
                       alors restaurer(B_i,K_i)
                             dépiler(B_i,K_i)
                             allera Retour
                       sinon début
                             co imprimer le résultat à partir de
                                PILE-RES co
                             RECH:si PILE-CHEMIN non vide alors
                                    si co sommet PILE-CHEMIN a une
                                          alternative satisfaisant
                                          le critère de choix co
                                       alors restaurer(A,K)
                                             restaurer PILE-CHOIX
                                             et PILE-RETOUR
                                             allera BOU
                                       sinon dépiler PILE-CHEMIN
                                             allera RECH
                                       fsi
                                    fsi
                             fin
                       fsi
```

Comme nous l'avions précisé dans le chapitre III, GENPHRASE écrit, à partir de chaque ramification trouvée, le mot du langage correspondant sur l'imprimante, et la phrase de test pour le paragraphe sur un fichier-disque. (Nous trouverons des résultats concernant cette procédure dans l'annexe 2.)

4-3-3. Conclusion

Nous avons choisi pour la génération de phrases à partir de la grammaire de ne jamais réutiliser deux fois la même règle lors d'une dérivation d'un non-terminal (cf. § 4-1-2); ce critère permet d'engendrer un nombre restreint de phrases. Nous pourrions prendre un critère de choix moins restrictif, tout en bornant l'algorithme, par exemple de passer au maximum deux fois dans une même règle lors de la dérivation d'un non-terminal.

Reprenons la grammaire citée en exemple au § 4-1-1, dont les huit phrases engendrées en respectant le premier critère de choix ont été énumérées au § 4-1-2; en prenant maintenant le critère de choix ci-dessus, nous obtenons évidemment les phrases précédentes, mais aussi d'autres, en particulier :

SI CONDITION ALORS AFFECTATION
SINON SI CONDITION ALORS AFFECTATION
SINON AFFECTATION

dont la ramification sous forme postfixée est :

10 8 6 10 8 6 8 6 2 5 3 1 5 3 1

On y trouve deux occurrences des règles 3 et 1, ce qui produit un enchaînement d'appels de procédures qui n'est pas testé par les jeux d'essai précédents.

Seulement avec ce nouveau critère, 18 phrases seront générées, dont la plupart n'offre pas d'intérêt supplémentaire pour le paragraphe.

Ce critère offre un ensemble de tests plus complet, mais avec encore plus de tests redondants qu'avec le premier critère de choix.

A l'échelon du test d'un paragraphe de grammaire réduite il pourra être intéressant de le prendre pour Genphrase. Dans les autres cas, et en particulier pour la grammaire du compilateur entier, on ne peut envisager de le prendre que dans une version conversationnelle (cf. § 3-6-2), où l'utilisateur désigne lui-même les phrases de l'ensemble à éliminer pour le test. On pourrait sans doute envisager une réduction automatique de l'ensemble de phrases de test, si l'on trouve de bons critères d'élimination de certaines des phrases.

CONCLUSION

Dans ce travail nous avons défini et implémenté des outils d'aide à la mise au point d'un compilateur écrit en Face ; pour cela nous avons mis à la disposition de l'utilisateur un langage de mise au point Mface et une bibliothèque de sous-programmes de mise au point (cf. annexe 0). Le langage Mface permet de décrire les données nécessaires pour engendrer les jeux d'essai (Genphrase) et interpréter ces jeux d'essai à l'échelon du paragraphe (Appltest), ou même d'engendrer des jeux d'essai au niveau du compilateur entier.

Nous avons vu qu'il faut tout d'abord rendre le langage de mise au point conversationnel (cf. § III.6-2), ce qui permettra de prendre pour Genphrase un critère de choix meilleur (cf. § 4-3-3) afin de rendre plus complet l'ensemble de tests.

Dans une étape ultérieure, il faudrait que la partie définitive du paragraphe (partie appartenant au compilateur entier cf. § 2-5-1) soit conservée, sous forme de modules-objets. L'ensemble des modules-objets, de chaque procédure de rangée, de chaque procédure auxiliaire, complété au moins par l'expression fermée finale du compilateur entier, formera le compilateur sous sa forme définitive. Pour réaliser ceci il faut que les compilateurs Face et Mface génèrent directement des modules-objets. Cela représente un gain de temps et d'efficacité considérable pour la mise au point du compilateur ; en effet, ceci éviterait une compilation totale du compilateur et surtout, si l'on décèle des erreurs lors du test du compilateur entier, il suffirait de recompiler le ou les modules concernés.

ANNEXES

A N N E X E O

Bibliothèque de mise au point :

ECRIREP
ECRIRANGREP
MENTREE
APPLTEST
GENPHRASE
TRANSGRAM
GENAPPLTEST

La bibliothèque de mise au point comprend des sous-programmes permettant la mise au point des paragraphes ou du compilateur entier. Ces sous-programmes permettent d'écrire les repères et les rangées de repères, soit explicitement en cours d'exécution par ECRIREP et ECRIRANGREP, soit implicitement pendant l'interprétation du paragraphe, par APPLTEST et GENAPPLTEST; un autre sous-programme permet de vérifier les tables MENTREE (cf. § 3-3-2), qui édite lors de son appel la table concernée.

Nous pourrions compléter la bibliothèque en y ajoutant une procédure MISAPPL, ayant pour paramètre une rangée de repères représentant la ramification et une rangée de procédures, qui applique les procédures de la rangée dans l'ordre indiqué par la rangée d'entiers comme APPL et APPLTEST, mais permet de plus un suivi de l'exécution : elle écrit les résultats, ceux cités dans la troisième partie du programme Mface, et les résultats empilés, après chaque appel aux procédures de la rangée.

Dans l'exemple décrivant un programme Face, cité au § 1-1-2, nous aurions pu ainsi faire un suivi de l'exécution, en y plaçant l'expression fermée suivante, suivie de la partie résultats :

Fin du programme	Commentaires
(ANALYSE(1;CHAINE); MISAPPL(CHAINE;LEX); MISAPPL(CHAINE;GEN))	Appel de l'analyseur, le résultat est placé dans CHAINE. Appel de MISAPPL : appel des procédures citées dans CHAINE, d'abord pour la rangée LEX, puis GEN, avec l'écriture de la pile et des résultats entre chaque appel.
RESULTATS NB;ML;NO;AD;	Les résultats à écrire successivement sont : NB : nombre de blocs trouvés ML : pointeur vers la zone locale NO : numéro du bloc en cours AD : rangée de repères pointant sur les adresses en zone locale de chacun des blocs

Voici dans l'ordre,les sous-programmes que nous trouverons dans la bibliothèque :

ECRIREP, ECRIRANGREP, MENTREE, APPLTEST, GENPHRASE, TRANSGRAM, GENAPPLTEST, MISAPPL.

NOM	PARAMETRES	ROLE
ECRIREP(A)	Repère : A	Procédure qui imprime la valeur repérée par A.
ECRIRANGREP(RA)	Rangée de repères : RA	Procédure qui imprime les valeurs repérées par la rangée RA.
MENTREE(TABLE;INDIC; INFO;FILTRE)	Procédure : TABLE qui a pour résultat la rangée de repères dans la table, l'information correspondant à l'indicatif donné. Rangée de repères : INDIC. Rangée de repères : INFO. Procédure : FILTRE qui a pour résultat un booléen et pour argument la rangée de repères INDIC.	Adjoint à une table l'information donnée INFO, pour l'indicatif INDIC, grâce à la procédure TABLE [3]. L'entrée est sélectionnée par la procédure FILTRE. La procédure édite aussi la table dans son entier (cf. § III-3-2).

APPLTEST(RPROC)	Rangée de procédures:RPROC	Considère un ensemble de ramifications, représentées sous forme postfixées par des rangées d'entiers, et applique les procédures de la rangée RPROC suivant l'ordre donné pour chacune d'entre elles, en communiquant à l'utilisateur, à la fin de cette application, les résultats sur la pile et ceux demandés par l'utilisateur, tout en considérant les différents jeux d'essai (cf. § 3-1-2)
GENPHRASE(NX)	Entier : NX, numéro de la règle de l'axiome.	Génère un ensemble de ramifications, représentées sous leur forme postfixée dans des rangées d'entiers, engendrées à partir de l'axiome qui a pour règle, celle dont le numéro est NX (cf. § 3-1-2 et 4-1-2).
TRANSGRAM(NX)	Entier : NX, numéro de la règle de l'axiome.	Ecrit la grammaire simplifiée : pour cela elle part de la grammaire donnée, dont une des règles de l'axiome porte le numéro NX, et appelle le module de simplification de la grammaire (cf. § 4-1-1).

GENAPPLTEST(NX;RPROC)	Entier : NX, numéro d'une règle de l'axiome. Rangée de procédures : RPROC.	Combine les effets de GENPHRASE et APPLTEST : génère le même ensemble de ramifications que GENPHRASE mais,chaque fois qu'une nouvelle ramification est trouvée, applique comme APPLTEST les procédures de la rangée RPROC dans l'ordre cité par la ramification, en prenant les différents jeux d'essai possibles, et en écrivant les résultats de la pile et ceux demandés (cf. § 3-1-2).
MISAPPL(CHAINE; RPROC)	Rangée d'entiers : CHAINE représentant la ramification sous forme postfixée. Rangée de procédures:RPROC.	Applique les procédures de la rangée, dans l'ordre indiqué par la rangée d'entiers, et imprime les résultats demandés et les résultats empilés entre chacun des appels.

A N N E X E 1

Programmation de MAPPL

La procédure GENAPPLTEST

La Procédure MAPPL (CHAINE; TAPPLREG; ETAT; RPROC)

La procédure MAPPL est appelée par APPLTEST, elle a été décrite au § 3-5-2 ; rappelons cependant qu'elle applique les procédures d'une rangée de procédures RPROC, dans l'ordre donné par une rangée d'entiers CHAINE, décrivant une ramification. Une autre rangée d'entiers, TAPPLREG, indique les entiers trouvés dans la rangée précédente, correspondant à des numéros de procédures manquantes ; une dernière rangée, ETAT, repère l'étiquette de la procédure envisagée pour chacun des appels à des procédures manquantes.

Cette procédure a été programmée en Métasymbol, son analyse et programmation est voisine de celle de APPL [10].

```
MAPPL   LW,R1   -5,POP    R1 contient l'adresse de la chaîne
        LW,R2   -4,POP    R2 contient l'adresse de TAPPLREG
        LW,R3   -3,POP    R3 contient l'adresse de ETAT
        LW,R5   -2,POP    R5 contient l'adresse de la rangée
        AI,R1   -1                          de procédures
        LW,R4   R1
        AW,R4   *R1       } calcul de l'adresse du dernier
        STW,R4  ADERNELT                          élément
        LW,R4   *R3
        AI,R3   1
        B       0,R4      Appel à la procédure qui concerne le
@RPL    AI,R1   1                             contexte
        CW,R1   ADERNELT
        BG      FINAPPEL  Test de fin sur le dernier élément
        LW,R4   *R1       Chargement du n° de la procédure
        CW,R4   *R3       Comparaison avec les n° de procédures
        BNE     PLOIN     manquantes
        LW,R4   *R3       Cas d'une procédure manquante
        AI,R2   1         On a l'adresse de l'étiquette dans
        AI,R3   1         le tableau ETAT
        B       0,R4
```

```
PLOIN     LW,R4   *R5,R4      Cas d'une procédure non manquante
          B       0,R4       Branchement à l'adresse trouvée dans
          LW,R3   POP        la table de la rangée
M:SNAP    DCBLO,'SORTIE PILE',(*R3,*R3+F)  Ecriture de la pile
          LW,R1   *RES       RES est l'étiquette de la zone où
          CI,R1   0          se trouvent les résultats à imprimer
          BE      FINPROC    Branchement s'il n'y a pas de résultat
          CI,R2   1          à  imprimer
          LW,R3   1
          CW,R1   R2         Test de fin sur la liste des résultats
          BE      FINPROC    à imprimer
BOU       LW,R4   RES,R3
          AI,R3   1
          LW,R5   RES,R3
          CI,R5   1          Test pour savoir si le résultat à im-
          BE      B1         primer est un repère ou une rangée de
                             repères
M:SNAP    DCBLO,'VALEUR',(*R4,*R4+1)
          GOTO    B2
B1
M:SNAP    DCBLO,'RANGEE',(*R4,*R4+F)        Impression de 16 élé-
B2        AI,R2   1                         ments pour la rangée
          AI,R3   1
          GOTO    BOU
FINPROC   LW,R4   0,POP
          B       0,R4
```

La Procédure GENAPPLTEST(NX,RPROC)

Cette procédure combine les actions de GENPHRASE et APPLTEST (cf. § 3-1-2) : après la génération d'une ramification, à partir de l'axiome dont une des règles est de numéro NX, elle applique les procédures de la rangée RPROC, dans l'ordre cité par la ramification, en considérant les différents jeux d'essai pour les procédures manquantes et en imprimant les résultats, ceux pris sur la pile et ceux demandés par l'utilisateur.

Nous reprenons l'algorithme de génération des ramifications une à une, (cf. § 4-3-2) dans lequel nous insérons, lors de la création d'une ramification, l'algorithme principal de APPLTEST, qui choisit les jeux d'essai et applique chacune des procédures citées dans l'ordre de la ramification (cf. § 3-5-1).

L'algorithme devient :

```
      K1:=1 ;
      A:=X;
BOU : si A ∉ N2 alors empiler PILE-RES(A)
                      empiler PILE-CHEMIN(A,K1)
                sinon début
                      empiler PILE-CHEMIN(A,K1)
                      choix d'une règle K à partir de K1, non
                      en PILE-CHOIX et de premier membre A
                      empiler PILE-CHOIX(A,K)
                      pour tous les B_i de la Kième règle faire
                        début si B_i ∈ T
                                alors empiler PILE-RES(B_i)
                                sinon empiler PILE-RETOUR(B_i,K)
                                      A:=B_i ; K1:=1 ;
                                      allera BOU ;
                                fsi
                             RETOUR :
                        fin
                      empiler PILE-RES(K)
                      dépiler PILE-CHOIX(A,K)
                      fin
      si PILE-RETOUR non vide
         alors restaurer(B_i,K1)
               dépiler PILE-RETOUR(B_i,K1)
               allera RETOUR
         sinon début
               co imprimer le résultat à partir de PILE-RES
                  et le mettre dans CHAINE co
               APPLTEST1(CHAINE;RPROC)
```

```
RECH : si PILE-CHEMIN non vide
          alors si co SOMMET PILE-CHEMIN a une
                      alternative satisfaisant
                      le critère de choix co
                alors restaurer(A,K0)
                      restaurer PILE-CHOIX
                            et PILE-RETOUR
                      allera BOU
                sinon dépiler PILE-CHEMIN
                      allera RECH
                fsi
          fsi
     fin
fsi
```

APPLTEST1 (CHAINE; RPROC)

```
ETAT[1]:= EPM[1] ;
CONSTRUCTION DE TAPPLREG co soit J éléments dans TAPPLREG co
co CONSTRUCTION DE ETAT co
POUR I DE 2 à J FAIRE ETAT[I]:= EPM[TAPPLREG[I]+2]
MAPPL(CHAINE;TAPPLREG;ETAT;RPROC);
RECHERCHE POSSIBILITE(ETAT;1;J+1;RPROC);
```

Le module CONSTRUCTION de TAPPLREG se trouve au § 3-5-1, et les procédures MAPPL et RECHERCHE POSSIBILITE au § 3-5-2.

A N N E X E 2

RESULTATS DE GENPHRASE ET TRANSGRAM

GRAMMAIRE DONNEE

<INSTRUCT>::=<INS.COND>|60<AUTRES INS.>|61

<INS.COND>::=SI<EXP.BOOL.>ALORS<AUTRES INS.><2.MEMBRE>|62

<2.MEMBRE>::=<VIDE>|63SINON<INSTRUCT.>|64

<AUTRES INS.>::=<INST.AFF>|65<INST.POUR>|66

<INST.AFF>::=AFFECT|67

<INST.POUR>::=INS.POUR|68

<EXP.BOOL>::=EXP.BOOL|69

<VIDE>::= |

} règles manquantes

GRAMMAIRE SIMPLIFIEE

il reste 3 non-terminaux récursifs :

INSTRUCT::=INS.COND / 60
AFFECT / 61
INS.POUR / 61

INS.COND::= SI EXP.BOOL ALORS AFFECT 2.MEMBRE / 62
SI EXP.BOOL ALORS INS.POUR 2.MEMBRE / 62

2.MEMBRE::= / 63
SINON INSTRUCT / 64

STOP 0

PHRASES ET RAMIFICATIONS GENEREES:

```
PHRASE
   SI       EXP.BOOL     ALORS     AFFECT
RAMIFICATION:   69    67    65    63    62    60

PHRASE
   SI       EXP.BOOL     ALORS     AFFECT      SINON     AFFECT
RAMIFICATION:   69    67    65    67    65    61    64    62    60

PHRASE
   SI       EXP.BOOL     ALORS     AFFECT      SINON    INS.POUR
RAMIFICATION:   69    67    65    68    66    61    64    62    60

PHRASE
   SI       EXP.BOOL     ALORS    INS.POUR
RAMIFICATION:   69    68    66    63    62    60

PHRASE
   SI       EXP.BOOL     ALORS    INS.POUR     SINON     AFFECT
RAMIFICATION:   69    68    66    67    65    61    64    62    60

PHRASE
   SI       EXP.BOOL     ALORS    INS.POUR     SINON    INS.POUR
RAMIFICATION:   69    68    66    68    66    61    64    62    60

PHRASE
 AFFECT
RAMIFICATION:   67    65    61

PHRASE
INS.POUR
RAMIFICATION:   68    66    61
```

GRAMMAIRE DONNEE

<EXPRES.>::=<TERME>|<EXPRES.><OPA><TERME>|[8]

<TERME>::=<FACTEUR>|<TERME>*<FACTEUR>|[6]

<FACTEUR>::=<PRIMAIRE>|<FACTEUR>**<PRIMAIRE>|[5]

<PRIMAIRE>::=PRIMAIRE|[1]

<OPA>::= +|[2] -|[3]

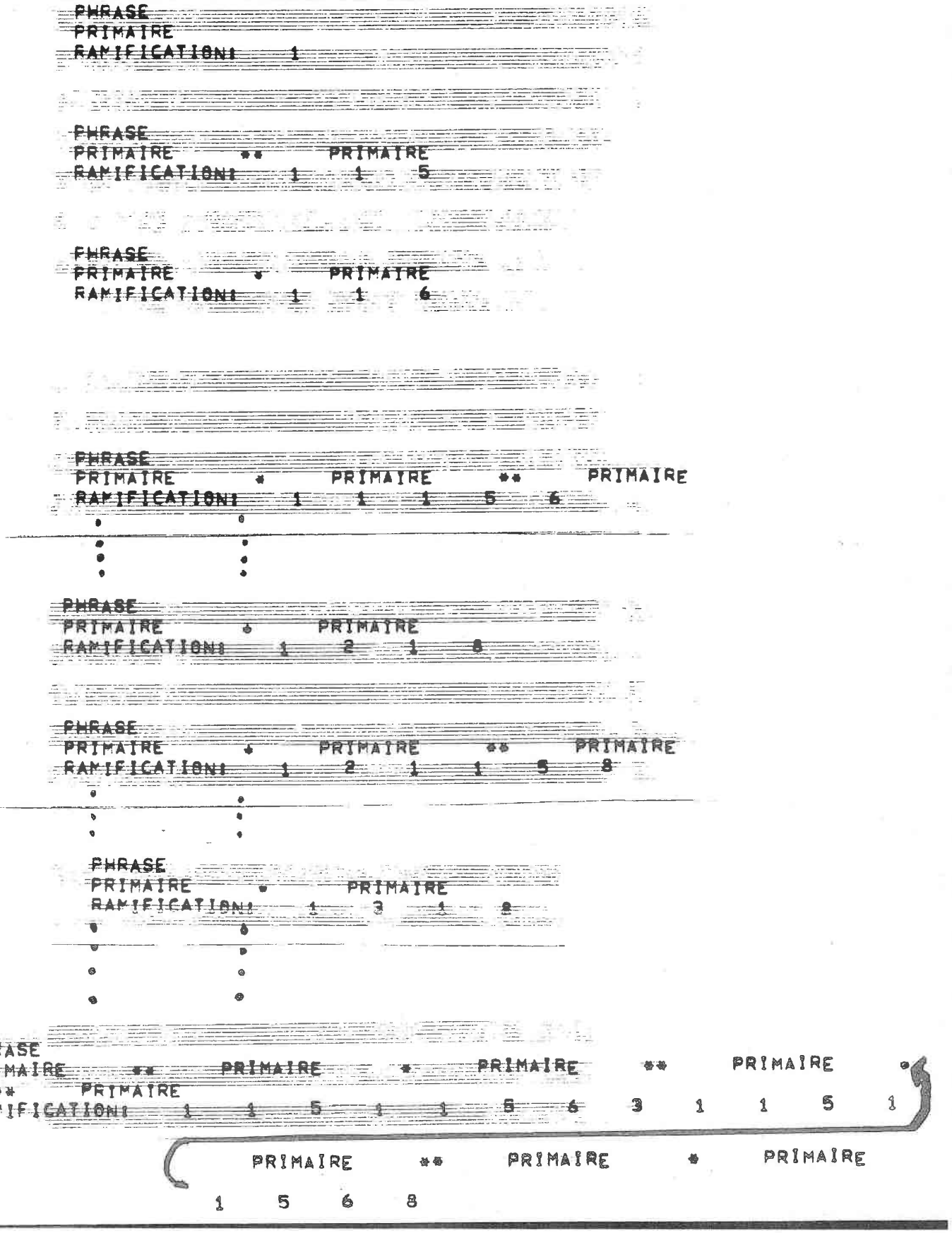
} règles manquantes

GRAMMAIRE SIMPLIFIEE

```
FACTEUR ::=PRIMAIRE /
          FACTEUR      **    PRIMAIRE /  5
TERME   ::=FACTEUR /
          TERME        *     FACTEUR  /  6
EXPRES.::= TERME /
          EXPRES.      +     TERME    /  8
          EXPRES.      -     TERME    /  8
*STEP* 0
JOB STEP 03 TERMINATED AT  18*35*49* AFTER 0000.05 MIN
```

temps pour TRANSGRAM et GENPHRASE

PHRASES ET RAMIFICATIONS GENEREES



BIBLIOGRAPHIE

[1] AHO A.V., ULLMAN J.D.
The theory of parsing, translation and compiling.
Englewood Cliffs (N.J.), Prentice Hall (1972).

[2] BAUER F.L. (ed)
Compiler construction. Lecture Notes in Computer Science 21.
Springer Verlag (1974).

[3] BELLEGARDE F.
Face: Langage d'écriture de compilateurs. Définition et implémentation.
Thèse de spécialité, Nancy (1972).

[4] CONWAY R.W.
Design of a separable transition. Diagram compiler.
Comm. ACM 6, p. 396-408 (1963).

[5] DERNIAME J.C., PAIR C.
Problèmes de cheminement dans les graphes.
Monographies d'Informatique, Dunod (1971).

[6] FELDMAN J., GRIES D.
Translator writing systems.
Comm. ACM Vol 11, Num 2, p. 77-113.

[7] GINSBURG S.
The mathematical theory of contexte free languages.
McGraw-Hill, New-York (1966).

[8] GRIFFITHS M.
Introduction to compiler-compilers.
Compiler construction (cf. [2]).

[18] GRIFFITHS M.
Analyse syntaxique pour la production de compilateurs. (1973).

[19] HOUSSAIS B.
Production systématique de tests pour une implémentation d'Algol 68.
Congrès AFCET 1976 (à paraître).

[9] HOPCROFT J.E., ULLMAN J.D.
Formal languages and their relation to automata.
Reading (Mass.), Addison Wesley (1969).

[10] MAROLDT J.
Définition de Face, Langage pour l'écriture de compilateurs. Implémentation d'un sous-ensemble.
Thèse de Docteur-Ingénieur, Nancy (1972).

[11] PAIR C., QUERE A.
Définition et étude des bilangages réguliers.
Information and control (1968).

[12] PAIR C.
Sur des notions algébriques liées à l'analyse syntaxique.
Fontainebleau (1969).

[13] PAIR C.
Cours de compilation.
Ecole d'Eté, Neufchatel (1972).

[14] PAIR C.
Langages et automates.
Cours Ecole des Mines (1973).

[15] PISTRE H.
L'analyse syntaxique dans Face, Langage pour l'écriture de compilateurs.
Thèse de Spécialité (1975).

[16] RUSSEL E.C.
Meta 5 manual. Version for XDS SIGMA 7.
Underbatch processing monitor, IRIA (1970).

[17] VAN WIJNGAARDEN A.
Report on the algorithmic language Algol 68.
Mathematisch Centrum, Amsterdam (1968).

NOM DE L'ETUDIANT : MADAME BOUCHET ép. RASSER Anne-Marie

NATURE DE LA THESE : DOCTORAT DE SPECIALITE en MATHEMATIQUES APPLIQUEES

VU, APPROUVE

& PERMIS D'IMPRIMER

NANCY, le 10/9/1976

LE PRESIDENT DE L'UNIVERSITE DE NANCY I

UNIVERSITÉ DE NANCY I
Le Président

M. BOULANGE