Institutional Repository - Research Portal
Dépôt Institutionnel - Portail de la Recherche

researchportal.unamur.be

University of Namur

THESIS / THÈSE

MASTER EN SCIENCES INFORMATIQUES

Création d'un environnement de spécification pour le langage SPES

Buyse, Michel; Vanhemelryck, Paul

Award date:
1983

Awarding institution:
Universite de Namur

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

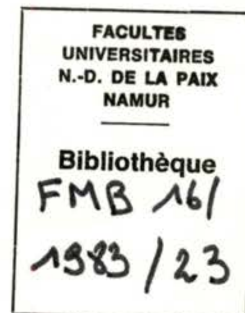
- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal ?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 05. May. 2024

FACULTE UNIVERSITAIRE
N.D. DE LA PAIX - NAMUR
Institut d'Informatique

Année académique 1982-1983


FACULTES UNIVERSITAIRES N.-D. DE LA PAIX NAMUR
Bibliothèque
FMB 16/
1983/23

CREATION D'UN ENVIRONNEMENT DE SPECIFICATION POUR LE LANGAGE SPES

Michel Buyse
Paul Vanhemelryck

Mémoire présenté en vue de l'obtention du grade de licencié et maître en informatique.

Avant d'aborder le corps de ce travail, nous tenons à remercier toutes les personnes qui, à titre divers, lui ont permis de voir le jour.

Nos remerciements vont tout d'abord à M. A. van LAMSWEERDE professeur à l'Université de Namur et promoteur de ce travail. Sa compétence scientifique, son attention soutenue et ses encouragements nous furent d'un très grand recours, tout au long de l'élaboration de ce travail.

Nous tenons également à remercier les membre du Centre de Recherche en Informatique de Nancy pour l'accueil et les conseils prodigués. Parmi eux, nous remercierons plus particulièrement :

- M. J.P. FINANCE, professeur à l'Université de Nancy, pour l'attention qu'il a portée à ce travail, ainsi que pour le bien-fondé de ses suggestions.

- M. A. Queré, notre principal intermédiaire durant le stage, dont l'aide fût précieuse tout au long de ce travail, que sa disponibilité, à tout moment, soit ici remerciée.

- M. J.P. JACQUOT, qui nous fût d'une aide précieuse (et patiente) pour les problèmes techniques rencontrés.

Tous nos remerciements vont également à Mmes D. De Decker et C. Buelens, sans qui la dactylographie de ce mémoire n'aurait pu être réalisée.

Enfin, ces remerciements n'auraient pas été complets si nous n'avions pas évoqué ici, toutes les personnes qui , d'une façon quelconque, ont contribué à ce mémoire.

TABLE DES MATIERES

Pages

INTRODUCTION 1

PREMIERE PARTIE : PRESENTATION DU CADRE EXISTANT 6

1. L'usage d'un éditeur syntaxique 7

1.1 Introduction 7

1.2 Les éditeurs orientés ligne 8

1.3 Les éditeurs syntaxiques 9

1.3.1 Caractéristiques d'un éditeur syntaxique 9

1.3.1.1 Fonctions d'un éditeur syntaxique 9

1.3.1.2 Représentation interne des objets 10

1.3.1.3 Visualisation des objets 12

1.3.2 Avantages et inconvénients des éditeurs syntaxiques 12

1.3.3 Présentation de quelques réalisations dans le domaine des éditeurs syntaxiques 13

1.3.3.1 Le synthétiseur de Cornell 13

1.3.3.2 S.D.S. 25

1.3.3.3 Conclusion 28

2. Présentation du système MENTOR 31

2.1 Identification des objets manipulés 32

2.2 Phase d'édition 35

2.3 Visualisation 36

2.4 MENTOL 37

2.5 Génération d'un éditeur sous MENTOR 39

2.5.1 METAL 40

2.5.1.1 Introduction 40

2.5.1.2 Programme METAL 40

2.5.2 Présentation du système SYNTAX 47

./...

3. Présentation du langage SPES 51
3.0 Préliminaires 51
3.1 Présentation générale du langage 51
3.2 Exemple 52
3.3 Les énoncés 55
3.3.1 Structure générale des énoncés 55
3.3.1.1 Présentation informelle 55
3.3.1.2 Structure générale : formalisation 57
3.3.2 Les définitions informelles 57
3.3.2.1 Présentation informelle 57
3.3.2.2 Les définitions informelles : formalisation 58
3.3.3 Les définitions de type 58
3.3.3.1 Présentation informelle 58
3.3.3.2 Les définitions de type : formalisation 60
3.3.4 Les définitions formelles 60
3.3.4.1 Présentation informelle 60
3.3.4.2 Les définitions formelles implicites 60
3.3.4.3 Les définitions formelles explicites 61
3.3.4.4 Définition d'une liste d'objets 65
3.3.4.5 Formalisation de la syntaxe des définitions formelles 66
3.4 Les descriptions de type 68
3.4.1 Les descriptions d'expressions 68
3.4.1.1 Présentation informelle 68
3.4.1.2 Formalisation 69
3.4.2 Les descriptions strictes 69
3.4.2.1 Présentation informelle 69
3.4.2.2 Formalisation 72
3.4.3 Les descriptions d'adjonctions 72
3.4.3.1 Présentation informelle 72
3.4.3.2 Formalisation 73
3.5 Portée des identificateurs 73
3.6 Mécanisme de référence d'énoncé 73

./...

DEUXIEME PARTIE : LES OUTILS DEVELOPPES 75

1. Présentation générale de l'environnement de spécification 76

1.1 Scénario d'utilisation 76

1.2 Philosophie de développement 78

1.3 Architecture globale de l'environnement de spécification 79

2. Les outils de base 81

2.1 Génération de l'éditeur syntaxique pour le langage SPES 82

2.2 Le décompilateur 87

2.2.1 Présentation 87

2.2.2 Spécification 88

2.2.3 Conception 90

2.2.3.1 Les outils disponibles 90

2.2.3.2 Construction du décompilateur 91

3. Outils de génération de texte SPES 94

3.1 La gestion de pseudo-commentaires 95

3.1.1 Présentation 95

3.1.2 Spécification 97

3.1.3 Concepts 97

3.1.4 Choix de conception 98

3.1.4.1 L'insertion d'un énoncé dans la liste d'énoncés archivés 101

3.1.4.2 Extraction d'un énoncé de la liste des énoncés archivés 108

3.2 Création assistée de parties d'énoncés déjà référencés 110

3.2.1 Opportunité 111

3.2.2 Spécification de la procédure 111

3.2.2.1 Structure d'un énoncé SPES 111

3.2.2.2 Structure de l'en-tête et du corps du nouvel énoncé 115

3.2.2.3 Illustration 115

3.2.2.4 Contrôles à effectuer 116

./...

3.2.3 Conception de l'algorithme 117

3.2.3.1 Structure générale de l'algorithme 120

3.2.3.2 Détails de l'algorithme 124

3.2.4 Quelques détails d'implémentation 135

3.3 Génération de définitions de type abstraits pour les constructeurs de type 137

3.3.1 Introduction 137

3.3.2 Spécification 138

3.3.3 Conception 139

3.3.4 Recherche de méta-types 141

3.3.4.1 Recherche du méta-type associé au constructeur "pile" 142

3.3.4.2 Recherche du méta-type associé au constructeur "file" 143

3.3.4.3 Recherche du méta-type associé au constructeur "ensemble" 144

3.3.4.4 Recherche du méta-type associé au constructeur "suite" 146

3.3.4.5 Recherche du méta-type associé au constructeur "struct" 147

3.3.4.6 Recherche du méta-type associé au constructeur "structfonct" 148

3.3.5 Realisation 149

3.3.5.1 La normalisation des descriptions d'expression 149

3.3.5.2 La génération des descriptions d'expression 151

3.4 Génération de descriptions de type à partir d'énoncés 152

3.4.1 Introduction 152

3.4.2 Spécification 153

3.4.3 Conception 154

3.4.3.1 Les constructeurs unaires 155

3.4.3.2 Les constructeurs n-aires 158

3.4.3.3 Exemple 159

3.4.4 Réalisation du processus de génération de description de type *def-type(dossier) 160

./...

4. Outils d'analyse d'une spécification 163

4.1 Vérifications effectuées au sein d'un énoncé 163

4.1.1 Existence et unicité des définitions d'objets et détection des objets non utilisés 164

4.1.1.1 Exemple 165

4.1.1.2 Spécification 166

4.1.1.3 Concepts 167

4.1.1.4 Construction 170

4.1.2 Recherche des cycles de définitions d'objets 173

4.1.2.1 Spécification 174

4.1.2.2 Concepts 174

4.1.2.3 Procédé de construction 176

4.1.3 Vérification des concordances de type au sein d'un énoncé 177

4.1.3.1 L'évaluateur de type 179

4.1.3.2 Processus de vérification de type 184

4.2 Vérifications de cohérence effectuées au sein du dossier 188

4.2.1 Présence d'un et un seul énoncé racine 189

4.2.1.1 Spécification 189

4.2.1.2 Procédé de construction 189

4.2.1.3 Construction 190

4.2.2 Détection des cycles de références 191

4.2.2.1 Réalisation 191

4.2.2.2 Procédé de construction 192

4.2.2.3 Conception 193

4.3 Vérifications effectuées au sein de la bibliothèque 196

4.3.1 Vérification de l'existence d'une description pour tout type 196

4.3.1.1 Spécification 196

4.3.1.2 Procédé de construction 197

4.3.1.3 Construction 198

4.3.2 Détection de cycles de descriptions de type 198

4.3.2.1 Spécification 198

4.3.2.2 Procédé de construction 199

./...

4.4 Contrôles effectués dans la spécification 199

4.4.1 Introduction et concepts 199

4.4.2 Procédé de construction 201

4.4.3 Création de l'arborescence et des prédicats-arcs 201

4.4.3.1 Pré-condition 201

4.4.3.2 Spécification 201

4.4.3.3 Rappels 202

4.4.3.4 Procédé de construction 203

4.4.3.5 Construction 208

4.4.4 Construction de la table de vérité associée aux prédicats-arcs 209

4.4.5 Détection des redondances et des incompatibilités 212

4.4.5.1 Cohérence d'une ligne de prédicat-ars 216

4.4.5.2 Interactions entre une ligne de vérité et une ligne du prédicat-chemin 217

4.4.5.3 Interactions entre plusieurs conditions 227

5. <u>Production de récapitulatifs</u> 230

5.1 Production de récapitulatif de profils d'énoncés 231

5.1.1 Opportunité 231

5.1.2 Spécification de la procédure 232

5.1.3 Conception de l'algorithme 234

5.1.3.1 Structure de l'algorithme 234

5.1.3.2 Détail de l'algorithme 235

5.1.4 Quelques détails sur l'implémentation de procédure 237

5.2 Impressions regroupées 239

5.2.1 Opportunité 239

5.2.2 Spécification de la procédure 239

5.2.3 Conception de l'algorithme 248

5.2.3.1 Structure générale de l'algorithme 248

5.2.3.2 Détails de conception de l'algorithme 249

./....

5.2.4 Quelques détails d'mplémentations 252

CONCLUSIONS 255

A. Evaluation critique de MENTOR 256

B. Evaluation critique de SPES 258

C. Appréciation des outils développés 259

D. Extensions possibles à ce travail 260

ANNEXE A: MENTOL 262

1. Repère 262

2. Les commandes de déplacement dans l'arbre 262

3. Assignation d'un repère à un noeud 263

4. Manipulation d'arborescences 264

4.1 Copie 264

4.2 Destruction 264

4.3 Remplacement 264

5. Recherche d'une occurence d'un sous-arbre particulier 264

6. Affichage 265

7. Les structures de contrôle 265

7.1 La séquence de commandes 265

7.2 La boucle 266

7.3 La sortie de boucle 266

7.4 La commande de traitement d'exception 266

7.5 La commande de liaison de commandes 267

ANNEXE B : ALGORITHME D'EVALUATION DU TYPE DU RESULTAT D'EXPRESSION 268

ANNEXE C : ALGORITHME DU VERIFICATEUR DU TYPE 271

ANNEXE D : PROCEDE DE CONSTRUCTION DU CONSTRUCTEUR DE TABLE DE VERITE 274

REFERENCES BIBLIOGRAPHIQUES 284

INTRODUCTION

Le coût élevé du developpement et de la maintenance de logiciels fiables constituent un problème qui a retenu une attention croissante au cours de ces dernières années. /‾BOE,83_7, /‾LAM,82_7.

On a ainsi estimé que le coût de maintenance peut atteindre 70 % du coût total d'un logiciel, alors que la spécification et la conception de celui-ci ne représente que 10% .

Une des causes principales de ce phénomène réside dans le fait que les spécifications des différents besoins à satisfaire et problèmes à résoudre par le logiciel sont généralement médiocres /‾GER,80_7.

Bien spécifier un problème est une tâche difficile car :

- les spécifications sont à la base du contrat liant la personne qui pose le problème(le demandeur) et l'informaticien qui le résoud. En ce sens elles doivent être à la fois précises et compréhensibles dans les deux parties;
- ces spécifications constituent en même temps le point de départ du travail de l'informaticien, qui ne connaît généralement pas le référentiel de base du demandeur: elles doivent par conséquent faciliter les tâches ultérieures de conception, de validation et de maintenance.

Les recherches actuelles dans le domaine de la spécification s'étendent sur plusieurs axes : les méthodes de spécification les langages de spécification et les aides automatiques à la spécification /‾LAM,82_7.

La description du problème à résoudre peut,bien entendu, se faire en langue naturelle, ce qui peut présenter certains inconvénients.

Nous mentionnerons plus particulièrement le caractère ambigu de la langue naturelle (un mot ou une phrase peuvent avoir plusieurs significations), les risques de verbosité et de "bruits" pour la spécification ainsi que les risques d'erreurs d'interprétation. On considère de plus en plus que l'utilisation d'un langage formel de spécification peut remédier à de tels problèmes, et facilier la transition vers les étapes ultérieures de développement. Un tel langage offrira des concepts interprétables d'une et une seule façon, de par le fait qu'ils doivent respecter des règles syntaxiques et sémantiques précises.

De plus, le caractère formel d'une spécification permet à celle-ci d'être traitée par machine. De tels traitements peuvent être classifiés en deux groupes :

- Il y a des traitements de base permettant, par exemple, l'édition, la vérification syntaxique et sémantique de spécifications formelles.
- Il y a des traitements spécialisés destinés à fournir certaines aides à l'utilisateur dans son travail de description de problèmes (par exemple, génération automatique de parties de texte d'une spécification formelle en fonction d'informations déjà contenues dans cette spécification).

Le projet SPES, developpé au Centre de Recherche et Informatique de Nancy,s'inscrit dans ce genre de préoccupations.

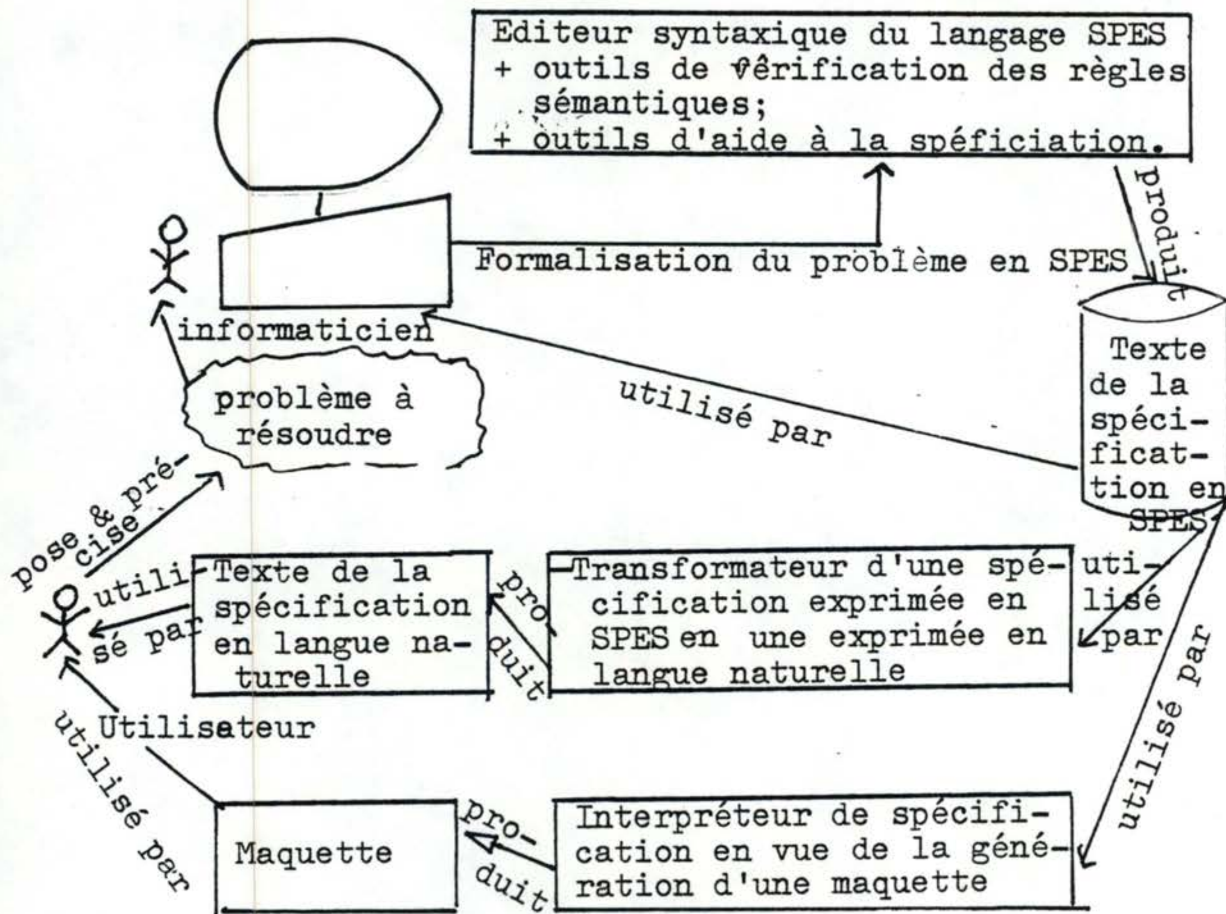
Son noyau est constitué d'une langage formel de spécification appelé SPES, destiné à être utilisé en relation avec une démarche déductive [FAI, 79] [DUB,82] , [DUB,81]

./...

L'objectif de ce projet est de développer un environnement de spécification autour du langage SPES. Cet environnement devra comprendre :

- des outils de base permettant l'édition et le contrôle syntaxique et sémantique du texte SPES;
- des outils spécialisés permettant :
  - l'interprétation de texte SPES en vue de générer une maquette du logiciel;
  - la transformation de texte SPES en textes en langue naturelle destinés au demandeur;
  - un certain nombre d'outils d'aides à la conception de textes SPES.

Le schéma suivant reprend les différentes préoccupations relatives au projet SPES.



L'objectif de notre travail était de développer des outils de base pour le langage SPES, ainsi qu'un certain nombre d'outils d'aide à la conception de textes SPES.

Les grandes lignes de notre travail furent les suivantes :

- Défintion d'une syntaxe et d'un début de sémantique pour le langage SPES, au départ d'une grammaire incomplète et mal définie.

- Génération d'un éditeur syntaxique pour le langage SPES, réalisé en utilisant l'environnement MENTOR [MEL,82] [MEL 2,82] de l'I.N.R.I.A.

- Développement d'outils spécialisés effectuant principalement la vérification de règles sémantiques ainsi que la génération automatique de parties de textes SPES.

Le présent mémoire comprend deux parties :

La première présentera le cadre existant .

Dans un premier chapitre, nous introduirons les concepts liés à l'usage d'un éditeur syntaxique. Nous étudierons brièvement les éditeurs orientés lignes, en insistant sur leurs désavantages, pour aborder ensuite l'étude des éditeurs syntaxiques, en soulignant leur caractéristiques, avantages et inconvénients. Nous terminerons ce chapitre par la présentation de quelques réalisations dans ce domaine.

Le second chapitre consiste en une présentation du système MENTOR, système générique utilisé comme éditeur syntaxique pour le langage SPES. Nous présenterons les concepts de base du système MENTOR, à savoir la caractérisation des objets manipulés,

./...

la phase d'édition consistant à effectuer le passage entre la forme externe et la forme interne, et la visualisation effectuant l'opération inverse. La génération d'un éditeur sous MENTOR est réalisée par intermédiaire d'un programme écrit en METAL, fournissant la description du langage paramètre. Nous étudierons les concepts qui y sont utilisés.

Le dernier chapitre consiste en l'étude du langage SPES. Après une présentation générale du langage, nous fournirons un exemple de spécification en SPES. Nous en détaillerons ensuite les différentes constructions.

La seconde partie de ce mémoire présentera l'environnement de spécification que nous avons réalisé pour le langage SPES.

Dans le premier chapitre sera présenté l'architecture des outils créés, ainsi qu'un scénario de leur utilisation par le spécifieur. Nous présenterons également la philosophie de leur développement, en y justifiant les caractéristiques des outils créés ainsi que le choix opéré parmis les outils réalisables.

Le second chapitre introduit les outils de base, nécessaires à l'obtention d'un éditeur syntaxique pour le langage SPES par l'utilisation du système MENTOR. Nous présenterons, dans un premier temps, les problèmes pouvant y survenir lors de la génération de l'éditeur syntaxique. Nous aborderons ensuite la manière de réaliser le passage de la forme interne à la forme externe via le décompilateur.

Le troisième chapitre présentera les outils réalisant la génération du texte SPES. Le premier de ces outils se rapporte à la gestion de messages, les pseudo-commentaires, reprenant sous

./...

forme textuelle les informations relatives aux références entre énoncés. Le second outil permet la création assistée de nouveaux énoncés, consistant à la génération de la sémantique informelle et du type des objets résultats et arguments de l'énoncé. Le troisième outil consiste en la génération de définitions de types suivant le concept des types abstraits, caractérisant un type par les opérations y associées; opérations possédant elles-mêmes une sémantique formelle. Le dernier outil présenté dans ce chapitre concerne la génération de descriptions de type à partir d'énoncés. Cet outil permet d'inférer la structure d'un objet en se basant sur les manipulations effectuées sur celui-ci.

Le quatrième chapitre présente les outils d'analyse d'une spécification. Ces outils assurent la vérification de règles sémantiques d'une spécification SPES. Nous classerons ces vérifications en 4 groupes, selon qu'elles sont effectuées au niveau de l'énoncé, d'une liste d'énoncés, de la bibliothèquem des types, ou en fin de la spécification. Ces vérifications portent sur des règles écrites en langue naturelle dans la description du langage. Parmi celles-ci, nous mentionnerons :

- la vérification des concordances de type au sein d'un énoncé;
- le contrôle de l'existence d'un et un seul énoncé racine dans une liste d'énoncés;
- la vérification de l'apparition d'une description pour tout type.

Le cinquième et dernier chapitre de cette partie est consacrée aux outils producteurs de récapitulatifs. Nous y envisagerons successivement le récapitulatif des profils d'énoncés précisant pour chaque énoncé le type de ses arguments et résultats, et le récapitulatif de la spécification, comprenant la liste des énoncés, le récapitulatif des profils d'énoncés et la bibliothèque de type.

Notre conclusion comportera quatre points : l'évaluation du système MENTOR, l'appréciation du langage SPES, la critique des outils développés et enfin, les extensions possibles de ce travail, dans lequel nous insisterons sur la nécessité d'adjonction d'un pilote, aidant le spécifieur et imposant une démarche de spécification.

./....

Notre expérience en tant que créateur et utilisateur des outils développés nous a, en effet, fait sentir de façon cruciale la nécessité d'un interface entre l'utilisateur et le système MENTOR. L'interface à créer (le pilote) imposerait l'utilisation d'une démarche de spécification, et effectuerait automatiquement des contrôles sémantiques sur des textes syntaxiquement corrects.

PREMIERE PARTIE : PRESENTATION DU CADRE EXISTANT

La première partie de ce mémoire introduit les concepts globaux que nous retrouverons tout au long de ce travail.

Le premier chapitre introduit la notion d'édition, et présente quelques caractéristiques communes aux éditeurs syntaxiques, ainsi que certaines réalisations dans ce domaine.

Le second chapitre présente le système MENTOR, mis au point à l'INRIA à Paris. C'est par l'utilisation de ce système que sera généré l'éditeur syntaxique pour le langage SPES.

Le troisième chapitre présente les notions de base du langage de spécification SPES, langage pour lequel l'environnement de spécification sera créé.

1. L'USAGE D'UN EDITEUR SYNTAXIQUE

1.1 Introduction

La création et la modification de texte représentent des tâches courantes dans un environnement informatique. Ces tâches sont assurées via l'utilisation d'éditeurs.

L'éditeur est un programme qui permet à l'utilisateur de créer et de réviser un objet "document". Nous utilisons le terme "document", afin d'inclure des objets tels que programmes, textes, équations, tables,diagrammes, graphiques. A ce propos, nous nous limiterons aux éditeurs de texte; le mot texte étant pris dans son sens général (suite de caractères, en général texte de programme, texte de spécification...).

Parmi ces éditeurs, nous distinguons deux familles : les éditeurs orientés ligne et les éditeurs syntaxiques.

- Les éditeurs orientés ligne sont conçus pour la manipulation de textes quelconques (textes libres, textes de programmes...). Les caractères, les chaînes de caractères, les lignes en sont les unités de traitement.
- Les éditeurs syntaxiques prennent en compte la structure syntaxique d'un langage pour en manipuler les textes. Les unités syntaxiques du langage en sont les unités de traitement.

Nous évoquerons brièvement les désavantages des éditeurs orientés ligne, pour aborder ensuite l'étude des éditeurs syntaxiques où nous mettrons en évidence un certain nombre de caractéristiques, d'avantages et d'inconvénients. Nous citerons un certain nombre de réalisations dans ce domaine. Parmi ces réalisations, nous étudierons plus spécialement le système Mentor, qui fait partie de notre environnement de travail.

./...

1.2 Les éditeurs orientés ligne

Les inconvénients des éditeurs orientés ligne proviennent essentiellement de la structuration des objets qu'ils manipulent. D'une manière générale, ils conçoivent le contenu d'un texte comme une suite de lignes; chaque ligne étant constituée d'une suite de caractères. A ce niveau, ils ne peuvent donc pas faire la différence entre un texte libre, un texte de programme, un texte de spécification, ... puisque chacun de ceux-ci est constitué de suites de lignes de caractères.

De même, ils sont limités dans l'aide qu'ils peuvent fournir pour construire des textes structurés (textes de programme ou de spécification obéissant à une syntaxe déterminée) de par le manque d'informations qu'ils possèdent sur les objets qu'ils manipulent. En effet, rien ne leur permet de différencier une chaîne de caractères d'une autre : dans le cas d'une chaîne PROCEDURE XYZ, rien ne leur permet de dire que PROCEDURE est un mot-réservé du langage et que la chaîne XYZ est un identificateur de PROCEDURE. Ils ignorent également que le mot PROCEDURE introduit une structure de bloc qui doit se terminer par le mot-réservé END.

Cette incapacité de prise en compte de la structure syntaxique d'un texte empêche toute vérification de la validité syntaxique de celui-ci lors de l'édition, et exclut toute possibilité de construction automatique de parties de texte en fonction des structures utilisées.

En conséquence, l'analyse syntaxique n'est possible que plus tard, lors de l'interprétation ou de la compilation. Un certain nombre d'aller-retour entre édition et compilation seront donc en général nécessaires pour obtenir un texte syntaxiquement correct, ce qui pourra influencer défavorablement le coût du développement

du programme et la productivité du programmeur

Ces inconvénients ont conduit les recherches vers de nouveaux types d'éditeurs, prenant plus en compte l'aspect syntaxique des textes.

1.3 Les éditeurs syntaxiques

Nous commencerons par évoquer certaines caractéristiques communes à ce type d'éditeur, ainsi que leurs avantages et leurs inconvénients. Ensuite, nous présenterons quelques réalisations dans ce domaine : le synthétiseur de Cornell, SDS. Ce choix a été guidé par les idées que chacun de ces systèmes apporte au niveau du concept d'édition syntaxique.

En conclusion, nous comparerons ces différents systèmes de manière à en souligner les différences.

1.3.1 Caractéristiques d'un éditeur syntaxique

1.3.1.1 Fonctions d'un éditeur syntaxique

La plupart des éditeurs sont interactifs et orientés vers des introductions et des visualisations faites à partir de vidéos.

Les fonctions classiques assurées par les éditeurs de texte, à savoir la création, la modification, la suppression et la visualisation de textes ou parties de texte , tiennent compte de ces caractéristiques.

Dans le cas des éditeurs syntaxiques, ces opérations se font en termes de construction du langage plutôt qu'en termes d'unité textuelle (caractère, chaîne de caractères, ligne...).

Aux fonctions de création et de modification s'ajoute la fonction de contrôle syntaxique qui permet de s'assurer que la partie de texte créée ou modifiée est bien conforme à la syntaxe du langage.

Ces fonctions produisent et/ou utilisent une même représentation interne du texte.

La fonction de visualisation transforme cette représentation en une représentation textuelle. Nous y reviendrons après l'étude de la représentation interne.

1.3.1.2 Représentation interne des objets

La plupart des éditeurs syntaxiques reposent sur une représentation interne du texte à éditer sous forme d'arbre appelé arbre abstrait. Cet arbre met en évidence la structure syntaxique du texte pour une grammaire donnée.

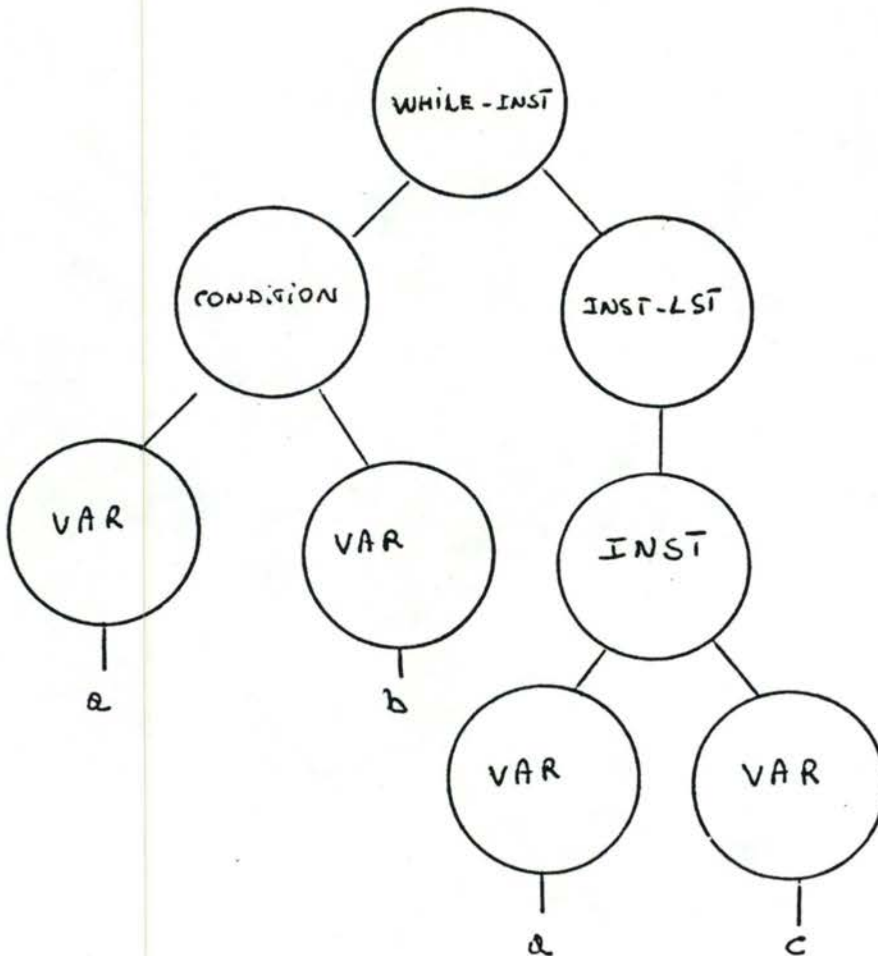
Les noeuds de cet arbre sont des objets structurés qui représentent des constructions du langage, constitués d'un nom et d'un ensemble de champs. Le nom identifie une construction du langage et les champs font référence au(x) fil(s) d'un noeud qui peuvent être des objets structurés ou des unités lexicales différenciées des mots-réservés.

Dans l'arbre, les noeuds seront les non-terminaux du langage et les terminaux faisant référence aux unités lexicales (différenciés des mots-réservés); les feuilles seront ces unités lexicales et les arcs des pointeurs reliant un noeud père à ses noeuds fils.

Prenons par exemple les règles suivantes :

<while-inst> ::= WHILE <condition> DO <inst-lst> END
<inst-lst> ::= <inst>
<inst-lst> ::= <inst> ; <inst-lst>
<inst> ::= <var> := <var>
<condition> ::= <var> = <var>
où les mots soulignés sont des mots-réservés.

La représentation arborescente de l'instruction :
WHILE a = b DO a:=c END
serait :



L'ensemble des règles définies plus haut constitue ce que l'on appelle généralement la syntaxe concrète du langage. Dans la représentation arborescente, la structure du langage apparaît plus simple parce qu'on y fait abstraction de tous les mots-réservés. Cette structure simplifiée du langage sera généralement appelée syntaxe abstraite du langage.

1.3.1.3 Visualisation des objets

La fonction de visualisation des éditeurs syntaxiques est généralement réalisée par l'intermédiaire d'une routine particulière appelée décompilateur, qui transforme un arbre abstrait en un texte formaté. Elle utilise des schémas de décompilation qui décrivent la correspondance existante entre un noeud et sa représentation textuelle.

Ces schémas de décompilation contiennent des mots-réservés, des ponctuations ainsi que des instructions destinées au gestionnaire d'écran qui permettent de donner une représentation agréable du texte (indentations, sauts de lignes, ...).

La décompilation s'effectue par parcours de l'arbre abstrait en générant le texte correspondant au noeud rencontré, suivant son schéma de décompilation.

1.3.2 Avantages et inconvénients des éditeurs syntaxiques

Les éditeurs syntaxiques accroissent considérablement la productivité des programmeurs pour plusieurs raisons :

- Ils déchargent le programmeur de détails syntaxiques propres à chaque langage (productions, mots-réservés, format des instructions).

./...

- Ils réduisent considérablement le temps de mise au point d'un programme en effectuant un contrôle syntaxique immédiat sur les parties de texte introduites, en signalant les erreurs, permettant ainsi à l'utilisateur une correction immédiate. D'autres avantages sont liés à la structure d'arbres qu'ils utilisent. On peut, par exemple, supprimer tout un arbre sans devoir supprimer successivement tous les sous-arbres qui le constituent.
- Cette structure interne arborescente peut être exploitée par d'autres outils. Elle pourrait permettre, par exemple, une automatisation de certaines phases du développement de programme, tels que des contrôles sémantiques sur des parties de texte, l'établissement de graphes d'appel, ...

Pour terminer ce paragraphe, il faut signaler deux désavantages des éditeurs syntaxiques :

- Ils sont généralement gourmands en place mémoire, principalement à cause du stockage de l'arbre abstrait en mémoire et des routines interactives. Cet inconvénient devient peu-à-peu mineur à mesure que le coût des mémoires diminue.
- Ils sont généralement lents en temps d'exécution par la gestion de l'arbre abstrait et le contrôle syntaxique. Cependant cette lenteur est largement compensée par la rapidité avec laquelle les programmes sont constitués.

1.3.3 Présentation de quelques réalisations dans le domaine des éditeurs syntaxiques.

1.3.3.1 Le synthétiseur de Cornell (TEIT, 81a) (TEIT, 81b) (MEY, 82)

Le synthétiseur de programmes, créé par une équipe de l'Université Cornell est un environnement de programmation interactif qui inclut un système d'édition guidé par la syntaxe, une

aide à la mise au point et une aide à l'exécution de programmes.

Son développement a débuté en 1978.

La première réalisation a été effectuée pour le langage PL-1, ce qui a donné le PL/CS. A l'heure actuelle, d'autres réalisations sont en cours, notamment pour le langage Pascal. Parallèlement à ces nouvelles réalisations, l'équipe du Cornell essaye de définir un système permettant de générer un synthétiseur de programme pour un langage donné à partir de la définition de sa grammaire (cf Mentor).

Identification des objets manipulés

Deux types d'objets sont manipulés dans le synthétiseur de programmes : les schémas (templates en Anglais) et les phrases.

Un schéma est une représentation pré-établie d'un non-terminal du langage. Il est constitué de détenteurs de place (placeholders en Anglais), de mots-réservés du langage, de ponctuation et de formats d'indentation.

Les schémas constituent la charpente de construction des programmes. Ils ont un caractère immuable. Les détenteurs de place identifient les endroits où peuvent se faire les insertions d'autres schémas et de phrases. De plus, les détenteurs de place désignent des classes syntaxiques auxquelles appartiennent les différents objets qui peuvent se placer à cet endroit.

Par exemple, le schéma d'une instruction conditionnelle pourrait être le suivant :

```
IF (condition)
    THEN statement
    ELSE statement;
```

où "condition" et "statement" sont des détenteurs de place.

Une phrase est une séquence arbitraire de symboles. Elle est généralement utilisée pour introduire les instructions d'affectation, les expressions et les listes de variables considérées comme des objets non structurés.

Phase d'édition

L'utilisateur ne manipule que des schémas et des phrases. Au niveau interne, les différentes parties du programme seront représentées par des arbres dont les noeuds sont générés à partir des schémas et des phrases manipulés par l'utilisateur.

Vue externe de l'édition

Les programmes sont créés par raffinements successifs en insérant de nouveaux schémas ou de nouvelles phrases dans le squelette formé à partir des schémas précédemment introduits.

L'insertion se fait en remplaçant un détenteur de place par un schéma ou une phrase. Chaque insertion provoque un contrôle syntaxique qui permet de s'assurer que l'objet de remplacement appartient bien à la classe syntaxique désignée par le détenteur de place. Le remplacement se fait toujours pour un détenteur de place désigné par le curseur sur l'écran.

Vue interne de l'édition

Chaque insertion provoque la génération d'un noeud à partir du schéma ou de la phrase insérée par l'utilisateur.

./...

Ce noeud est représenté en mémoire par un enregistrement de longueur variable qui contient des caractères, s'il s'agit d'un noeud généré à partir d'une phrase; des zones réservées à des pointeurs, s'il s'agit d'un noeud généré à partir d'un schéma. Les zones pointeurs correspondent aux détenteurs de place du schéma. Le remplacement d'un détenteur de place par un schéma ou une phrase va se traduire au niveau interne par la création d'un nouveau noeud ou d'un pointeur de la zone correspondante, du noeud père vers le nouveau noeud. Les noeuds ainsi reliés formeront une arborescence.

Commandes

Il existe deux types de commandes. Les commandes de déplacement du curseur et les commandes d'éditions.

Commandes de déplacement du curseur

En général, le curseur se déplace sur l'écran par incréments logiques, c'est-à-dire d'un schéma à l'autre, d'un schéma à ses constituants, d'un constituant à un autre dans un même schéma, d'une phrase à ses caractères, et d'un caractère à l'autre dans une même phrase.
Ces déplacements vont se traduire au niveau interne par des déplacements dans l'arbre suivant l'incrément utilisé :

- up, down : déplacent le curseur d'un élément du programme à l'autre, s'arrêtant sur un schéma, un détenteur de place ou une phrase. La commande "up" déplace le curseur en remontant dans le texte, la commande "down" en suivant le texte.
- left, right : déplacent le curseur à l'intérieur d'une phrase, c'est-à-dire d'un caractère à l'autre. la commande "left"déplace le curseur en remontant dans le texte, la commande "right" en suivant le texte.

./...

- long up, long down : déplacent le curseur d'un schéma à un autre schéma situé à un même niveau dans la structure du programme. La commande "long up" déplace le curseur en remontant dans le texte, la commande "long down" en suivant le texte.
- diagonal : déplace le curseur vers l'élément englobant d'un niveau; c'est-à-dire d'un caractère à une phrase, d'une phrase à un détenteur de place, d'un détenteur de place à un schéma, d'un schéma à un autre schéma.
- long diagonal : remonte le curseur au niveau de l'élément englobant tous les autres, ou autrement dit, à la racine de l'arbre.

Commandes d'édition

Création

Chaque schéma du langage est identifié par un mnémonique. Il suffit donc de taper la commande ".X" pour insérer le schéma identifié par X à l'endroit désigné par le curseur.

Au niveau interne, cette commande va provoquer un contrôle syntaxique suivit éventuellement de la création d'un nouveau noeud ou d'un pointeur du noeud père vers le nouveau noeud, comme nous l'avons vu précédemment.

Un programme est créé à partir de suite d'insertions et de déplacements dans la structure arborescente formée à partir des schémas insérés. Nous avons vu comment se font les insertions et les déplacements dans la structure. Afin d'illustrer une phase de création d'une partie de programme, nous allons prendre l'exemple suivant:

./...

Soit le texte :

```
□IF (k<o)
   THEN "instruction"
   ELSE  PUT  SKIP  LIST ('non-négatif');
```

où □ représente la position courante du curseur.
Nous voulons remplacer le détenteur de place "instruction" par le schéma "PUT SKIP LIST (Liste-d'-expressions)".
Nous taperons successivement les commandes "down", "down" pour déplacer le curseur sur le détenteur de place "instruction", puisqu'il est le deuxième détenteur de place dans le schéma IF-THEN-ELSE.
On tapera la commande .p (où .p est mnémonique du schéma "PUT SKIP LIST (Liste-d'-expressions)". L'insertion est directement affichée à l'écran et le curseur automatiquement déplacé sur le détenteur de place "liste-d'-expressions". On aura donc affiché à l'écran :

```
IF  (k< o)
    THEN  PUT  SKIP  LIST (□iste-d'-expressions);
    ELSE  PUT  SKIP  LIST ('non-négatif');
```

Pour affiner le détenteur de place "liste-d'-expressions", l'utilisateur introduira explicitement cette liste, l'introduction du premier caractère provoquant le remplacement immédiat du détenteur de place. Cette dernière manipulation illustre la manière dont se fait l'introduction d'une phrase.

Suppression et modification

Les modifications d'un programme se font par l'intermédiaire de suppressions et d'insertions de schémas et de phrases. La suppression, comme l'insertion, se fait à partir de la position courante du curseur. La commande de suppression est "delete". Elle supprime l'élément auquel le curseur fait référence ainsi que tous les éléments qui lui sont inclus.

Remarques

Le synthétiseur de programmes offre en plus du contrôle syntaxique, un contrôle sémantique. Les erreurs telles que duplication de déclarations, non-existence de déclarations, erreur de type, sont signalées au même titre que les erreurs syntaxiques.

Dans certains cas, les erreurs sémantiques sont temporairement autorisées. Ainsi, on peut utiliser dans le programme des variables non déclarées. Dans ce cas, le fait est signalé et toutes les occurences présentes ou à venir de la variable seront soulignées de manière à ce que l'utilisateur sache que la déclaration n'a pas été faite. Ce dernier pourra la faire à n'importe quel moment.

Les outils d'aide à l'exécution et à la mise au point de programmes

Phase d'exécution après l'édition

Une particularité du synthétiseur de programmes est de pouvoir exécuter un programme directement après son édition sans ajouter de délai dû à la compilation. En fait, un code interprétable est généré pendant la phase d'édition du programme, ce qui permet d'exécuter un programme à n'importe quel niveau de son développement. Lorsque l'exécution du programme est suspendue, le contrôle passe à l'édition; un message donne la raison pour laquelle il y a eu suspension et le curseur est positionné dans le programme source au point de suspension. Ceci permet, lorsque la suspension a eu lieu par la rencontre d'un détenteur de place, d'éditer le texte qui doit venir le remplacer et de relancer l'exécution à cet endroit.

./...

Cette possibilité permet à l'utilisateur de mettre au point un programme de manière incrémentale. Il peut en effet ajouter un code et ensuite voir si ce code exécute bien ce qu'il en attendait. Cette façon de procéder est critiquable dans la mesure où la mise-au-point du programme se fait par essais/ erreurs plutôt que par raisonnement sérieux sur papier.

Le suivi de l'exécution

Cet outil permet à l'utilisateur de voir le programme source s'exécuter, instruction par instruction, et d'afficher le résultat de certaines actions. Pour ce faire, l'écran est divisé en deux, la première partie affiche le texte du programme tandis que les résultats sont affichés dans la deuxième partie. Lorsqu'une instruction est exécutée, le curseur se déplace dans le texte du programme indiquant ainsi l'instruction exécutée.

L'exécution à la carte d'un programme

Cet outil est très semblable au précédent, à la différence près que c'est l'utilisateur qui détient le contrôle de l'exécution. Il peut faire exécuter le programme à n'importe quelle vitesse et sélectionner des parties de programmes à visualiser.

Le contrôle des variables

L'utilisateur peut spécifier le ou les variables qu'il désire contrôler au moment de l'exécution du programme. Ces variables et leurs valeurs sont affichées sur une partie de l'écran lorsque leurs valeurs sont modifiées. (Ex. : instruction d'affectation).

Décompilation de la forme interne

Un texte de programme est représenté par un

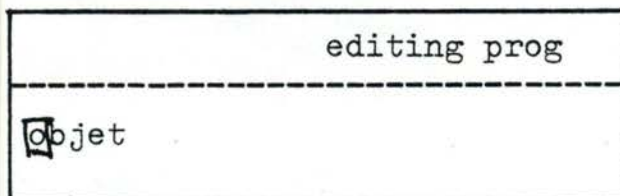
./...

arbre où les schémas et les phrases en sont les noeuds. C'est à partir de cette représentation que doit se faire la représentation textuelle.

La routine d'impression va donc parcourir cet arbre en remplaçant successivement les noeuds rencontrés par le texte qu'ils représentent. Lorsqu'elle rencontre un noeud généré à partir d'un schéma, la routine d'impression va rechercher dans une table les mots-réservés, la ponctuation et les formats d'indentation correspondant au schéma, et imprimer le texte. Lorsqu'elle rencontre un noeud généré à partir d'une phrase, la routine d'impression imprime le texte de la phrase tel quel. Les formats d'indentation sont des caractères interprétables par le gestionnaire de l'écran.

Illustration d'une phase d'édition

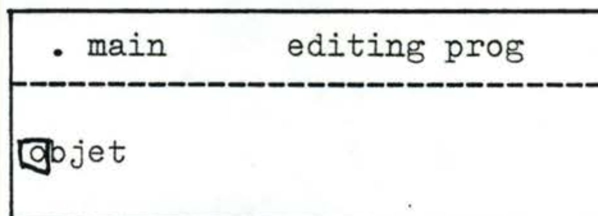
Un utilisateur désire écrire un programme Pl-1. Il a fait appel au système et signalé que le programme sera écrit dans un fichier de nom "prog".



)Zone système (pour l'intro-
)duction des commandes et
)l'affichage des messages)

)Zone texte (pour afficher
)le texte du programme édité)

L'objet "objet"est un détenteur de place. L'utilisateur désire le remplacer par le schéma d'une procédure principale Pl-1. Il tape la commande ".main" (main = mnémonique de ce schéma). La situation à l'écran sera successivement :



frappe de la commande.

./...

```
                        editing prog
----------------------------------------
/*comment*/
prog : PROCEDURE OPTIONS (MAIN);
        {déclaration}

        {statement}
END prog;
```

résultat de l'exécution de la commande

Le curseur désigne à présent le détenteur de place "comment". Les accolades indiquent la présence possible de listes aux endroits où elles sont placées. Ainsi {déclaration} indique la possibilité d'écrire une liste de déclarations.

Supposons à présent que l'utilisateur désire remplacer le détenteur de place "comment"par la chaîne de caractères "demonstration". Cette chaîne est une phrase et donc introduite explicitement à l'endroit désigné par le curseur.

La situation à l'écran sera après introduction :

```
                        editing prog
----------------------------------------
/*demonstration*/
prog : PROCEDURE OPTIONS (MAIN);
        {déclaration}
        {statement}
END prog;
```

Supposons à présent qu'il désire remplacer le détenteur de place "déclaration" par le schéma correspondant à la déclaration de variables de type entier. Il tape la commande .fx (fx =mnémonique de ce schéma). La situation à l'écran après exécution de la commande sera :

```
                        editing prog
-----------------------------------------------
/*demonstration*/
prog : PROCEDURE OPTIONS (MAIN);
       DECLARE (list-of-variables)FIXED;
      {déclaration}
      {statement }
END prog;
```

La situation à l'écran a été obtenue après un certain nombre de manipulations.

```
                        editing prog
-----------------------------------------------
 /*demonstration*/
 prog : PROCEDURE OPTION (MAIN);
        DECLARE (a,b,c ) FIXED;
        DECLARE (e,f,g ) FLOAT;
        GET LIST (a);
        PUT LIST (list-of-variables);
END prog;
```

./...

Supposons qu'il désire à présent supprimer la déclaration des variables de type entier. Il déplace le curseur jusqu'au schéma correspondant à cette déclaration. Il tape successive les commandes "diagonal" (curseur sur le P de PUT),"long-up" (curseur sur le G de GET), "long-up" (curseur sur le D du deuxième DECLARE), "long-up" (curseur sur le D du premier DECLARE).

Il tape la commande delete pour supprimer la déclaration et obtient à l'écran

```
error : undeclared variable   editing prog
------------------------------------------
/*demonstration*/
prog : PROCEDURE OPTIONS (MAIN)
       DECLARE (e,f,g ) FLOAT;
       GET LIST (a);
       PUT LIST (list-of-variables);
END prog
```

La variable "a" n'a plus de déclaration. Elle sera donc soulignée.

Travaux similaires

Nous citons ici le système EMILY (MEY,82) développé par HANSEN en 1971.

Ce système est très semblable au synthétiseur de Cornell, excepté :
- La non-existence dans EMILY d'outils d'aides à la mise-au-point et l'exécution de programmes. (La notion d'environnement de programmation n'existe pas).

./...

- Les phrases dans EMILY sont utilisées pour introduire les terminaux du langage tels que identificateurs, constantes, les schémas pour introduire les non-terminaux du langage, y compris les listes de variables, les expressions et les instructions d'affectation.

- Dans EMILY, le programme est représenté par un arbre dont les noeuds représentent les non-terminaux du langage et les feuilles les terminaux du langage. La génération d'un noeud se fait par sélection dans un menu d'une proposition de remplacement pour le non-terminal désigné par le curseur; la génération d'une feuille par introduction explicite du terminal.

1.3.3.2 S.D.S. (MEY, 82) (FRA,81)

Le système S.D.S., développé par FRASER, est encore expérimental. Bien que nous ayons peu d'informations à son sujet, il nous est apparu intéressant de le présenter. Son principal intérêt réside dans la possibilité qu'il a de générer un éditeur syntaxique pour une structure hiérarchique de données à partir d'une description formelle de celle-ci. Il est, à ce point de vue, représentatif des efforts entrepris à l'heure actuelle dans le domaine des éditeurs syntaxiques.

S.D.S. a été utilisé pour générer un éditeur d'arbres binaires, un éditeur graphique, un éditeur de documents et un éditeur pour un sous-ensemble du langage C.

Comme il est très semblable au synthétiseur de Cornell et au système EMILY, nous mettrons surtout en évidence ce qui lui est spécifique.

Présentation du système

Les objets manipulés par S.D.S. sont des

./...

articles et des textes. Les définitions des articles et des textes de S.D.S. sont celles des schémas et des phrases d'EMILY, excepté qu'aux articles s'ajoute un appel de procédure.

La procédure permet de décrire des actions à réaliser lorsque l'utilisateur manipule un article qui la référence.

Les actions à réaliser peuvent être, une visualisation des objets, un contrôle sémantique, des traitements particuliers propres à l'utilisateur, ...

La structure des données est représentée par un arbre dont les noeuds et les feuilles ont été générés respectivement à partir des articles et textes manipulés par l'utilisateur, ceci à la manière du synthétiseur de Cornell.

La procédure à laquelle on fait référence dans un article est exécutée lors de la création du noeud à partir de cet article.

<u>Définition de la grammaire paramètre et des procédures</u>

La définition de la grammaire se fait par l'intermédiaire d'une description formelle. Celle-ci met en évidence les non-terminaux de la structure de données considérée et les productions qui leur sont associées, au moyen d'une structure comprenant un identificateur, des champs et un appel de procédure.

L'identificateur fait référence à un non terminal, les champs, en décrivent la production, par référence à des terminaux ou non terminaux de la structure de données, l'appel de procédure fait référence aux actions à réaliser.

Prenons un exemple de définition formelle d'un éditeur d'arbres binaires. La grammaire qui décrit des arbres binaires a seulement une production :

arbre	=	valeur arbre arbre	: construire(valeur,arbre,arbre)
non-terminal de la structure		production associée à ce non-terminal qui fait référence à un terminal (valeur) et aux non-terminaux (arbre)	appel à la procédure "constuire" avec les paramètres valeur, arbre, arbre.

Ces structures sont utilisées pour générer des articles.

La définition des procédures est indépendante de celle de la grammaire. Elle se fait en donnant le code (en SNOBOL4) correspondant aux actions à réaliser.

Dans l'exemple précédent, la procédure "construire" servira à afficher à l'écran l'arbre et ses sous-arbres. Le code associé à cette action sera :

```
px = 33; py = 0; px = 20
DEFINE ('dosub (x,bpx,bpy,px,py,dx)')
DEFINE ('construire (v, l, r)') : (ends)
dosub  dosub = DIFFER (x)line (bpx,bpy + 1, px, py-1)PUT(x)
            :(RETURN)
construire  construire = curpos (px - SIZE(v)/2,py)v
            construire = construire dosub (l,px,py,px-dx,py+4,dx/2)
            construire = construire dosub (r,px, py,px+dx,py+4,dx/2)
            : (RETURN)
ends.
```

La première ligne utilise les variables qui détiennent les coordonnées de la position à l'écran où doit s'afficher la racine de l'arbre(px,py) et le déplacement horizontal entre la racine et ses descendants (dx). Les lignes définissent la procédure "construire" et la sous-routine"dosub". La procédure "construire" permet de centrer un valeur (valeur de la racine d'un arbre) à la position indiquée par les variables px et py et fait appel à la sous-routine "dosub" pour construire les deux sous-arbres. La sous-routine imprime un sous-arbre et l'arc reliant ce sous-arbre à la racine.

La définition de la grammaire paramètre et les procédures sont compilées et chargées avec les routines du système pour former un éditeur complet.

1.3.3.3 Conclusion

Nous n'avons envisagé dans ce chapitre que quelques réalisations dans le domaine des éditeurs syntaxiques. Signalons au lecteur qu'il en existe d'autres dans les systèmes GANDALF (MED,82), Interlisp (SAND,78), (TEITE, 81), CAPS, (WILC,75), IDEAL(RIN,82). Ces systèmes ne diffèrent pas énormément des systèmes précédents, c'est pourquoi nous avons cru bon de ne pas nous y attarder.

Nous allons donc souligner quelques caractéristiques des systèmes présentés pour permettre de les comparer.

EMILY fut l'une des premières réalisations dans le domaine des éditeurs syntaxiques. Il a donc son importance puisqu'il introduisait une nouvelle conception sur les éditeurs. EMILY est un système dérivationnel, en ce sens que la construction

de l'arbre est guidée par des choix opérés sur des menus imbriqués jusqu'au plus bas niveau, comme par exemple un identificateur. Cette façon de faire peut être très contraignante pour l'utilisateur, parce qu'il doit passer par de longues dérivations intermédiaires pour introduire un simple identificateur ou une expression. Pour pallier à ce problème, le synthétiseur de Cronell conçoit deux types d'objets : les schémas et les phrases. Les schémas prennent en compte des structures du langage tandis que les phrases sont des suites de symboles quelconques. Ils sont utilisés, par exemple, pour introduire les expressions. Ceci nous permet de dire que le synthétiseur de Cornell est un système hybride, ou semi-dérivationnel : il est en ce sens moins contraignant qu'EMILY, pour cette raison. Cependant, les phrases ne sont vérifiées syntaxiquement que plus tard en fin de celles-ci. Pour une expression, par exemple, la vérification syntaxique dans le synthétiseur se fera après son introduction, tandis que dans EMILY, elle se fera au fur-et-à-mesure de sa construction.

Dans le synthétiseur de programme apparaît également la notion d'environnement de programmation : il y a intégration d'outils d'aides à la mise au point et à l'exécution des programmes. Ceci était inexistant dans EMILY.

S.D.S. permet de générer un éditeur syntaxique pour une structure hiérarchique de données et permet à l'utilisateur de définir des actions à réaliser à la création des noeuds dans l'arborescence. Sur le plan de l'édition, il n'est pas très différent du synthétiseur de Cornell.

La tendance actuelle est de permettre à l'utilisateur de définir lui-même des environnements de programmation de plus en plus complexes et de concevoir des méta-systèmes permettant la génération d'éditeur syntaxiques à partir de descriptions formelles

./...

de langages paramètres. L'évolution du système Mentor, que nous avons utilisé dans le cadre de notre travail et que nous détaillons ci-après est représentative de cette tendance.

2. PRESENTATION DU SYSTEME MENTOR (MEL,82), (GUY,80)

Le système Mentor, mis au point à l'INRIA par V.Douzeau-Gauge, G. Huet, R. Kahn et B. Lang est un système permettant l'édition syntaxique de textes.

De plus, Mentor est également un manipulateur de textes. En effet, à partir de la connaissance de la structure syntaxique des textes, il est possible à l'utilisateur de décrire des traitement élaborés de ceux-ci en tenant compte de leur structure. (Ex. :remplacer dans la liste des paramètres de l'appel d'énoncé identifié par la constante caractère A, la variable "b" par la variable "a"). Un langage nommé Mentol permet de décrire algorithmiquement ces traitements.

Une première version du système fut créée pour le langage Pascal. Autour de l'éditeur syntaxique créé pour le langage Pascal, s'est développé un ensemble d'outils d'aides à la mises au point de programmes Pascal (Ex. : élaboration du graphe des appels de procédure, recensement des variables non-utilisées par le programme...), réalisé par l'intermédiaire de procédures Mentol, formant ainsi un environnement de programmation pour le langage Pascal.

Le système Mentor est, à l'heure actuelle, un système générique au sens qu'il permet de générer un éditeur syntaxique pour n'importe quel formalisme descriptible par une syntaxe de type BNF. Le langage désiré sera alors intégré à Mentor par l'intermédiaire d'un programme formulé dans le méta-langage Metal, dans lequel l'utilisateur doit donner une description formelle du langage qu'il désire.

2.1 Identification des objets manipulés

Mentor travaille sur base de la syntaxe abstraite du langage. Rappelons que la syntaxe abstraite d'un langage définit l'ensemble des constructions disponibles dans ce langage, abstraction faite des mots-réservés. Ces constructions sont aussi appelées opérateurs.

Mentor connaît trois types d'opérateurs : les opérateurs d'arité fixe, d'arité variable et les opérateurs atomiques. L'arité d'un opérateur désigne le nombre d'opérandes associées à celui-ci.

- Les opérateurs d'arité fixe représentent les non-terminaux du langage dont les productions sont constituées d'un nombre limité de terminaux ou de non-terminaux du langage. Par exemple, le non-terminal <conditionnelle> dont la production est <u>if</u> <condition> then <instruction> else <instruction> sera représenté par un opérateur d'arité fixe vu que la production ne contient qu'un nombre limité de non-terminaux du langage (à savoir <condition> et 2 fois <instruction>). Cet opérateur sera d'arité trois parce que ses opérandes sont une condition et deux instructions. Mentor distingue des opérateurs d'arité, nul, un, deux, trois, suivant le nombre d'opérandes (0,1,2 et 3).

- Les opérateurs d'arité variable représentent les non-terminaux du langage dont les productions sont constituées d'un nombre quelconque de terminaux de même type ou de non-terminaux de même type. Par exemple, le non-terminal <liste-instruction> dont la production est $\{$<instruction>$\}^*$ (signifiant un nombre quelconque d'instructions) sera représenté par un opérateur d'arité variable, vu que la production contient un nombre

quelconque de non-terminaux de même type (à savoir <instruction>). L'opérateur représentant le non-terminal <liste-instruction> aura un nombre quelconque d'opérandes "instruction".

- Les opérateurs atomiques représentent les non-terminaux du langage dont les productions sont les terminaux génériques du langage. Nous entendons par terminaux génériques du langage, tout terminal du langage pouvant être exprimé à l'aide d'un expression régulière. (Dans cette définition, nous excluons les terminaux du langage qui sont des mots-réservés). Par exemple, les constantes caractères dans un programme seront représentées par un opérateur atomique.

Mentor manipule donc des textes ou des parties de textes sous forme d'arborescences. Les noeuds de ces arborescences sont les opérateurs d'arité fixe non nulle et les opérateurs d'arité variable, les noeuds terminaux en sont les opérateurs d'arité nulle et les opérateurs atomiques.

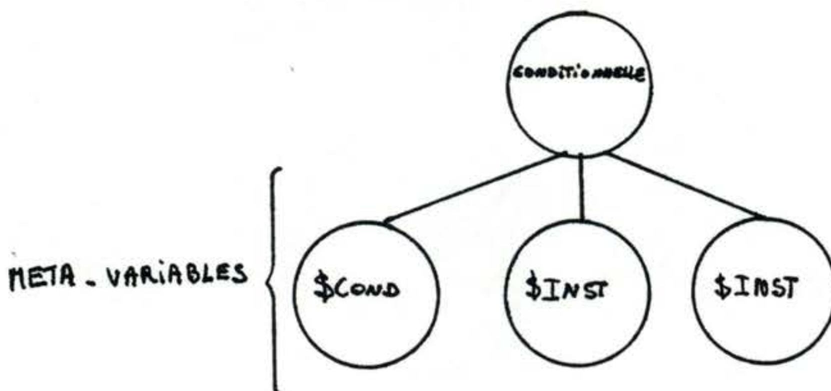
Ces opérateurs sont regroupés en classes, non nécessairement disjointes, appelées PHYLUM. Chaque noeud d'un arbre abstrait est en quelque sorte typé, au sens où un phylum lui est associé, qui indique quels sont les opérateurs qui ont le droit de s'y trouver. Le phylum associé à un noeud donné ne dépend que du père de cet emplacement (autrement dit, d'un opérateur). Chaque opérande d'un opérateur est, par conséquent, typé par son appartenance à un phylum. Dans le cas de la conditionnelle vue précédemment, les opérandes "condition" "instruction" "instruction" seront, par exemple, typées respectivement par les phylums COND, INST, INST exprimant ainsi que les opérandes qui viendront se placer à ces endroits devront

appartenir au phylum COND pour la condition, INST pour la 1ère et 2ème instruction. Le phylum COND regroupe, p.e., les opérateurs <u>pluspetit</u>, <u>égal</u>, <u>différent</u>, <u>plusgrand</u> et le phylum INST, les opérateurs <u>while</u>, <u>case</u>, <u>for</u>, <u>call</u>, <u>report</u>.

La notion de plylum est importante :

- Elle introduit des restrictions sur l'ensemble des arbres qu'il est possible de construire à partir d'un ensemble d'opérateurs donnés.
- Elle permet à Mentor de maintenir des arbres corrects, c.à.d. des arbres dont les noeuds sont des opérateurs qui appartiennent aux phylums associés à chacun des noeuds dans l'arborescence. Ainsi, chaque fois qu'une opération modifiant un arbre est exécutée (Ex. : remplacement d'un sous-arbre par un autre), Mentor vérifie la légitimité de cette opération et ne l'exécute que si elle est effectivement correcte, c.à.d. si l'arbre qui en résulte est un arbre correct. Ces opérations seront plus particulièrement étudiées dans le chapitre relatif à Mentol.

Il existe également en Mentor des arbres particuliers appelés schémas-prédéfinis. Ils sont construits en mémoire lors du chargement de Mentor pour un formalisme particulier. Il en existe un par opérateur distingué dans ce formalisme. La racine de ces arbres sont les opérateurs du formalisme et les fils, des noeuds particuliers appelés méta-variables. Une méta-variable est un opérateur terminal qui permet de considérer un sous-arbre dont on ne précise pas la structure. Le schéma prédéfini correspondant à l'opérateur "conditionnelle" est :



./...

Les schémas prédéfinis peuvent être utilisés dans certaines commandes Mentol (Ex. :remplacer un sous-arbre par le schéma prédéfini correspondant à l'opérateur "conditionnelle").

2.2 Phase d'édition

Une optique poursuivie par Mentor est de pouvoir introduire un texte comme on le ferait avec un éditeur classique. Une fois le texte tapé, il faut encore en générer l'arbre abstrait. Pour ce faire, Mentor utilise un analyseur-constructeur :

- L'analyseur comprend un analyseur lexical qui permet de reconnaître les lexèmes du texte et un analyseur syntaxique qui permet de regrouper ces lexèmes de manière à obtenir des phrases correctes du langage.

- Le constructeur regroupe les fonctions de construction d'arbres. Elles sont le lien logique entre la forme textuelle d'une phrase et sa forme arborescente.

A chaque fois qu'une réduction est effectuée, au cours de l'analyse syntaxique, la fonction de construction d'arbre, associée à la phrase reconnue (production) est exécutée pour en donner l'arbre correspondant.

Si au cours de ce processus, l'analyseur syntaxique ne peut opérer une réduction en fonction des lexèmes en présence, la phase d'édition est interrompue et un message en signale la raison.

Cependant, l'analyseur tentera de récupérer l'erreur en essayant de trouver une production pouvant contenir ces lexèmes. S'il y arrive, l'arbre est construit comme précédemment, un message

indique la correction effectuée et la phase d'édition se poursuit. Cette correction n'est pas toujours heureuse, car lorsque plusieurs solutions se présentent, il en choisit arbitrairement une.

Avant d'introduire un texte, Mentor demande à l'utilisateur de préciser un phylum. Ce phylum doit être de ceux définis dans les points d'entrées.

Les points d'entrées permettent d'indiquer au système les textes ou parties de texte que l'utilisateur est autorisé à introduire. (Ex. : si nous définissons un point d'entrée pour les variables, l'utilisateur peut introduire des variables au terminal, par contre, si nous n'en définissons pas, l'utilisateur ne sera pas autorisé à en introduire.

Remarque : Signalons que l'utilisateur peut créer un texte ou une partie de texte en utilisant les schémas pré-définis. Il lui suffit simplement de remplacer les méta-variables de ceux-ci par d'autres schémas pré-définis ou par des arbres créés à partir de textes, comme précédemment. Le remplacement d'une méta-variable par un arbre provoque un contrôle permettant de s'assurer que l'arbre créé peut effectivement remplacer la méta-variable considérée. Mentor utilisera la notion de phylum pour effecteur celui-ci.

2.3 Visualisation

La visualisation du texte correspondant à un arbre ou à un sous-arbre en mémoire est réalisé par l'intermédiaire d'un décompilateur. Il comprend pour chaque opérateur de la syntaxe abstraite, un schéma de décompilation dans lequel se trouvent les mots-réservés y associés et des détails de formatage, tels que espaces et sauts de lignes. L'impression du texte se fera en parcourant l'arbre et en imprimant successivement les schémas de

./...

décompilation correspondant aux noeuds rencontrés. Signalons que l'utilisateur doit écrire lui-même le décompilateur dans le langage Pascal.

2.4 Mentol (GUY, 80)

Mentol est le langage de commande pour Mentor. Sa particularité réside dans le fait qu'il est un langage offrant des primitives de manipulation d'arbres. Il permet à l'utilisateur d'introduire un texte qui sera représenté en mémoire par un arbre, de se déplacer dans celui-ci, de remplacer, supprimer ou copier des arbres ou des sous-arbres, d'assigner un repère à un noeud de l'arbre (le repère étant une variable contenant l'adresse du noeud en mémoire), de rechercher dans l'arbre une occurence d'un sous-arbre particulier, d'afficher un arbre ou un sous-arbre. A côté de ces primitives, il existe des structures de contrôle qui sont la séquence de commandes, la boucle, la sortie de boucle, la traitement d'exception et la liaison de commandes.

Ces commandes peuvent alors être combinées pour créer de véritables "programmes" appelés "Procédure". Nous donnons ci-après quelques explications sur certaines de ces commandes sans entrer dans leurs détails de syntaxe. (cf Annexe A)

Les commandes de déplacement dans l'arbre permettent d'atteindre un sous-arbre par rapport à un repère utilisé comme base. Elles permettent d'atteindre un noeud ascendant, un noeud situé soit à gauche, soit à droite sur un même niveau dans l'arbre ou un noeud descendant du 1er ordre du noeud repéré.

Les commandes de manipulation d'arbre (c'est-à-dire remplacer, copier, supprimer des arbres ou des sous-arbres) sont toutes suivies d'un contrôle syntaxique (cf. notion de phylum).

./...

La commande de recherche d'occurence de sous-arbres est assez particulière de Mentor. Elle permet de rechercher dans l'arbre une occurence d'un sous-arbre dont la structure est celle d'un sous-arbre passé en paramètre. Le sous-arbre passé en paramètre peut être un schéma pré-défini, ou un arbre créé à partir d'un texte introduit. (Ex. : rechercher dans l'arbre une occurence d'un sous-arbre dont la racine est l'opérateur "conditionnelle". Le sous-arbre passé en paramètre sera, par exemple, le schéma prédéfini correspondant à l'opérateur "conditionnelle").

La commande d'affichage permet de visualiser le texte correspondant à un arbre ou à un sous-arbre. Elle fait appel au décompilateur. L'utilisateur doit spécifier un repère et indiquer au système jusqu'à quelle profondeur de l'arbre, à partir du noeud repéré, doit se faire l'affichage.

La commande de traitement d'exception permet de tester la réussite ou l'échec de la commande précédente. Si la commande précédente a réussi, elle exécute la première séquence de commande, sinon la seconde. Ces deux séquences sont précisées dans la commande de traitement d'exception.

La commande de liaison de commandes ne permet d'exécuter une séquence de commandes que si la commande qui la précède a réussi.

La commande de sortie de boucles permet la sortie d'un certain nombre de niveaux d'imbrication dans les boucles.

Les procédures Mentol sont des textes créés à partir de plusieurs commandes Mentol. Ce dernier aspect donne à Mentor une grande puissance car l'utilisateur n'est pas devant un système d'édition syntaxique figé avec des commandes figées, mais il a

./...

cette fois-ci la possibilité de définir lui-même des procédures plus élaborées et plus adéquates à l'aide de commandes Mentol.

Dans le système Mentor, certaines procédures Mentol ont été pré-définies. Ces procédures sont des procédures d'utilité générales et non-spécifiques à un langage.

Nous citons ici quelques exemples :

. Save : permet de sauvegarder un programme conservé sous forme d'arbre.

. Restore : permet de lire un programme conservé sous forme d'arbre et de le reconstruire en mémoire.

. Parse : permet de construire l'arbre correspondant à un programme à partir de sa forme textuelle.

. Def : <<identificateur>, <liste de paramètres>,(<textes>)>
permet à l'utilisateur de définir une procédure Mentol. Les commandes qui constituent la procédure seront regroupées dans la zone texte. La procédure sera identifiée par un nom qui pourra avoir des paramètres locaux, ou passés par référence.

2.5 Génération d'un éditeur sous Mentor

Nous abordons ici un aspect intéressant de Mentor, à savoir qu'il est possible de générer un éditeur syntaxique pour n'importe quel langage descriptible par une syntaxe du type BNF.

Les différentes étapes de la génération sont :

- La description formelle du langage par l'intermédiaire d'un programme formulé dans le méta-langage Metal.

- La construction de l'analyseur lexical et la génération de l'analyseur lexical et syntaxique par l'intermédiaire du système SYNTAX.

Par la suite, nous allons aborder successivement ces deux étapes.

2.5.1 METAL (Mel, 82), (MEL 2, 82)(MEL 3,82)

2.5.1.1 Introduction

Le langage Metal est un méta-langage pour le système Mentor, qui permet de définir un formalisme de façon suffisamment complète pour qu'il soit possible de l'intégrer automatiquement dans le système Mentor. Un programme Metal P, est donc la définition d'un certain formalisme F. Nous appellerons Mentor-F, l'instance de Mentor, adaptée au formalisme F. Le système Mentor-F est composé du système Mentor lui-même, auquel sont associées les données décrivant le formalisme F. Du point de vue de l'utilisateur, la création des données qui décriront le formalisme F et que nous appellerons les tables de F, est faite par compilation du programme Metal P.

2.5.1.2 Programme Metal

Un programme Metal P qui définit un formalisme F, contient les éléments suivants :

1.- La syntaxe concrète de F, c'est-à-dire un ensemble de règles permettant de décider si une phrase donnée est syntaxiquement correcte dans le formalisme F.

Ces règles sont écrites dans une forme proche de la notation BNF. L'analyseur syntaxique de F sera dérivé à partir de ces règles.

2.- La syntaxe abstraite de F, c'est-à-dire la syntaxe des arbres qui représenteront les formules légales dans le formalisme F.

3.- Les fonctions de construction des arbres de syntaxe abstraite. Ces fonctions sont associées à chaque production de la syntaxe concrète. Elles sont le lien logique entre la forme textuelle, ou concrète, d'une phrase de F et sa forme arborescente, ou abstraite. A partir de ces fonctions sera dérivé le constructeur d'arbres pour le formalisme F. L'analyseur syntaxique de F sera ensuite relié au constructeur d'arbres pour former l'analyseur-constructeur associé à F.

4.- La définition des points d'entrée qui permettent d'indiquer au système les textes ou parties de textes que l'utilisateur est autorisé à introduire lors de la phase d'édition.

Le texte d'un programme Metal est structuré en chapitres et en sections. Le nom du langage est donné dans l'en-tête du programme. La syntaxe abstraite du langage est donnée dans les sections "abstract-syntax".On y trouvera plus particulièrement la définition en extension des phylums, c'est-à-dire l'énumération des opérateurs qui les composent, et les profils d'opérateurs, c'est-à-dire la définition des phylums auxquels doivent appartenir chacune de ces opérandes.

La syntaxe concrète du langage est donnée dans les sections "rules". On y trouvera plus particulièrement les productions de la syntaxe concrète, ainsi que leur fonction de construction d'arbres associée. Le couple production-fonction constitue ce que l'on appelle en Metal, une règle.

./...

La syntaxe concrète et les fonctions de construction d'arbre

Pour que le lecteur se familiarise avec l'écriture ou la lecture d'un programme Metal, nous utiliserons un exemple simplifié qui nous servira tout au long de la description des éléments d'un programme Metal.

Soit la définition suivante :

<appel-procédure> ::= <ident-proc> [(<liste-param-eff>)]
<ident-proc> :: <lettre> {<lettre>| <chiffre>}
<lettre> ::= A/B/C/D/E/F/G/H/I/J/K/L/M/N/O/P/Q/R/S/T/U/V/W/X/Y/Z
<chiffre> ::= 1/2/3/4/5/6/7/8/9/0
<liste-param-eff> ::= <param-eff> | <param-eff> <liste-param-eff>
<param-eff> ::= <cst-ent>
<cst-ent> ::= <chiffre> {<chiffre>}

où les noms entre "<" et ">" sont des non-terminaux, les symboles "[" et "]" introduisent des parties facultatives, les symboles "{" et "}" sont utilisés pour indiquer qu'un élément peut apparaître de 0 à n fois, le symbole "|" a pour signification "ou".

Nous allons à présent passer en revue les différents éléments qui constituent les zones de type "rules" en Metal

La syntaxe concrète

Pour cet exemple, la syntaxe concrète sera constituée des productions :

<appel-procédure> ::= <ident-proc> ;
<appel-procédure> ::= <ident-proc># (<liste-param-eff>#);
<ident-proc> ::=%LETCHIF;
<chiffre> :=%CHIF;
<Liste-param-eff> := <param-eff> ;
<Liste-param-eff> ::=<param-eff>#, <liste-param-eff> ;
<param-eff> ::= <cst-ent> ;
<cst-ent> ::= % ENTIER

./...

- Les noms %LETCHIF, %CHIF, %ENTIER sont des terminaux particuliers du langage, appelés terminaux génériques. Ils correspondent aux unités lexicales décrites explicitement par l'utilisateur dans l'analyseur lexical à l'aide d'expressions régulières. Ils se différencient donc des terminaux qui sont des mots-réservés du langage.
- Le symbole # qui précède certains symboles indique que ceux-ci doivent être compris comme tel et non comme des mots-réservés du méta-langage Metal.

Les fonctions de construction d'arbre

La fonction de construction d'arbre est le lien logique entre la forme concrète d'une phrase et sa forme abstraite (ou arborescence). Elle indique, pour une construction ou un opérateur du langage, quel est l'arbre associé.

Une fonction de construction d'arbre comprend :
- Le nom de l'opérateur auquel elle est associée;
- Un suffixe qui indique le type de l'opérateur (opérateur d'arité fixe, d'arité variable ou opérateur atomique);
- La définition des opérandes de l'opérateur par référence aux non-terminaux ou aux terminaux génériques de la syntaxe concrète.

Elle s'écrira sous la formule générale :
nom-opérateur¯suffixe (<opérande 1>,...,<opérande 3>)
Les suffixes indiquant le type des opérateurs sont
- "node" pour les opérateurs d'arité fixe.
  Signalons que la forme sans suffixe est équivalente à celle-ci.
- "list", "pré", et "post" pour les opérateurs d'arité variable;
  Les suffixes "pré" et"post" indiquent les endroits où doivent se faire l'ajout d'une opérande à la liste des opérandes (respectivement en début et en fin de liste).
- "atom" pour les opérateurs atomiques.

./...

Dans notre exemple, les fonctions de construction d'arbre sont :

ap_proc (< ident_proc >)
ap_proc_par (< ident_proc> , < liste_param_eff >)
ident_proc-atom (%LETCHIF)
chiffre-atom (%CHIF)
Liste_param-list ((< param_eff >))
Liste_param-pré (< param_eff >, < liste-param_eff >)
cst_ent-atom (%ENTIER)
< cst-ent >

ap_proc et ap_proc_par sont des opérateurs d'arité fixe, respectivement d'arité un et deux; liste_param est un opérateur d'arité variable dont l'ajout d'une opérande se fait en début de la liste des opérandes; ident_proc, chiffre et cst-ent sont des opérateurs atomiques.

REMARQUES

a) Chacune de ces fonctions de contruction d'arbre doit figurer dans le programme Metal, juste après la production à laquelle elle se rapporte.

b) Pour la production param-act::= < cst-ent > , nous n'avons pas de fonction de construction d'arbre. En fait, en indiquant à la deuxième ligne < cst-ent > , nous signalons que le fonction de construction d'arbre associée à la production < param-act >::= < cst-ent > est celle de < cst-ent > .

La syntaxe abstraite

La syntaxe abstraite est constituée de la définition en extension des phylums et du profil des opérateurs.

Ecrivons rapidement ce que serait la syntaxe abstraite de notre exemple avant d'aborder l'étude des différents éléments qui la constituent.

Abstract syntax

profil des opérateurs	ap-proc → IDENT-PROC; ap-proc-par → IDENT-PROC PARAM-LST; ident-proc → implemented as IDENTIFIER; chiffre → implemented as INTEGER; liste-param → CST-ENT +ₒₒₒ; cst-ent → implemented as INTEGER
définition des PHYLUMS	IDENT-PROC ::= ident-proc; (Phylums ne contenant qu'un opérateur) PARAM-LST ::= liste-param; CST-ENT ::= cst-ent;

où les noms en minuscules désignent les opérateurs et les noms en majuscules sont les PHYLUMS. Expliquons brièvement les différentes composantes de cette définition de syntaxe abstraite.

a) Profil des opérateurs

Le profil d'un opérateur fixe ou variable permet à l'utilisateur de définir le type de chacune des opérandes de l'opérateur en précisant le phylum auquel doit appartenir chacune de ses opérandes.

Dans le cas d'opérateurs d'arité variable, les opérandes doivent être de même type, donc de même phylum, mais l'utilisateur doit, en outre, indiquer si l'opérateur peut ou non ne pas avoir d'opérandes. Il l'indiquera au moyen des symboles "+" et "*", pour indiquer respectivement qu'il doit y avoir au moins une opérande et qu'il peut ne pas y en avoir.

Pour les opérateurs atomiques, l'utilisateur précise à la place du profil, le modèle d'implémentation qui doit être suivi par Mentor.

./...

- ident-proc et chiffre sont des opérateurs atomiques dont le modèle d'implémentation sera respectivement celui des IDENTIFIER et celui des INTEGER;
- ap-proc et ap-proc-par sont respectivement des opérateurs unaires dont l'opérande doit appartenir au phylum IDENT-PROC, et binaires dont la première opérande doit appartenir au phylum IDENT-PROC et la deuxième opérande au phylum PARAM-LST;
- liste-param est un opérateur d'arité variable dont les opérandes doivent appartenir au phylum CST-ENT. De plus, l'opérateur doit avoir au moins une opérande.

b) <u>Définition en extension des phylums</u>
Une définition d'un phylum revient à énumérer les opérateurs qui le constituent.

<u>Définition des points d'entrée</u>

La dernière étape dans la construction d'un programme Metal est la définition des points d'entrée dans la grammaire.

La forme d'un point d'entrée est la suivante :

<axiome> ::= [NOM-DE-PHYLUM] <non-terminal>

1) L'axiome de la grammaire est le non-terminal, à partir duquel il est possible d'atteindre tous les non-terminaux et terminaux du langage.

2) Le terminal [NOM-DE-PHYLUM] est appelé marque. Seuls les phylums dont le nom apparaît dans une marque pourront être utilisés comme points d'entrée.

2.5.2 Présentation du système SYNTAX (BOU,82)

Nous ne présenterons pas le système SYNTAX dans tous les détails, vu que l'utilisateur ne doit pas en connaître beaucoup pour pouvoir l'utiliser. La seule partie qui l'intéresse particulièrement est la définition et la construction de l'analyseur lexical pour le formalisme considéré et la génération de l'analyseur lexical et syntaxique.

Le système SYNTAX regroupe un ensemble d'outils permettant la construction et la génération d'analyseurs. Il comprend essentiellement les modules suivants :

- INTRO : qui numérote les définitions syntaxiques du langage décrites dans le programme Metal;
- Les constructeurs syntaxiques SLR1, SLRK, LR1, LALR1;
- OPTIM : qui optimise l'analyseur syntaxique produit;
- Le constructeur lexical NEW-10cl;
- Les outils d'analyse d'un texte source PARSER, SCANNER.

L'utilisation du module INTRO est la première étape par laquelle l'utilisateur doit passer pour construire un analyseur syntaxique. Dans le cadre du système Mentor, le fichier d'entrée au module INTRO a été généré automatiquement lors de la compilation du programme Metal. Le fichier d'entrée contient les définitions syntaxiques du langage dans le formalisme attendu par INTRO.

- Le module INTRO lit la grammaire, la met sous forme interne (numérotation des définitions), et effectue des tests de cohérence.

Ces tests sont les suivants :

- un non terminal doit être productif (c'est-à-dire conduire à une chaîne terminale);

./...

- un non-terminal doit être accessible depuis l'axiome;
- aucun non-terminal ne doit dériver de lui-même;
- L'utilisateur devra ensuite choisir un des constructeurs syntaxiques parmi ceux proposés dans le système SYNTAX.

Ceux-ci sont tous basés sur la technique LR, c'est-à-dire qu'ils construisent des analyseurs syntaxiques ascendants. Ils reçoivent en entrée la forme interne produite par INTRO de la grammaire et construisent un analyseur syntaxique du langage sous forme d'un automate à pile. Si au cours de la construction, ils détectent une non-conformité dans la définition de la grammaire, ils la signalent par un message d'erreur. Ces non-conformités peuvent être de deux types :

- soit il existe plusieurs possibilités de réduction de règles et il n'est pas possible de faire un choix;
- soit il y a conflit entre la lecture du symbole suivant et une réduction de règle.

L'automate produit par un des constructeurs sera ensuite optimisé par le module OPTIM. Ce dernier permettra de créer un automate qui sera utilisé par le PARSER lors de l'analyse syntaxique.

La dernière étape est la définition et la génération de l'analyseur lexical. L'utilisateur utilisera pour ce faire le constructeur lexical NEW-1Ocl.

Ce système accepte en entrée la description des unités lexicales du langage (cf. opérateurs atomiques), sous forme d'expressions régulières et la liste des terminaux délivrée par le module INTRO et produit un automate d'états finis.

La définition lexicale du langage est divisée en cinq parties :

- les classes simples;

./...

- les classes composées;
- les abréviations;
- les tokens.

a) Les classes simples

Les classes simples regroupent les caractères jouant un même rôle du point de vue lexical. Une définition de classe simple est, soit une chaîne de caractères entre guillemets, soit un code octal précédé du caractère #. Un caractère donné ne peut appartenir qu'à une classe simple. De plus; il est créé automatiquement une classe simple pour chaque caractère utilisé dans les terminaux de la grammaire et non redéfini au niveau lexical.

Exemple de classe simple :

LETTRE ::= "aAbB"   CHIFFRE ::= "1,2,3,4"
POINT ::= "."

b) Les classes composées

L'utilisateur peut définir des classes composées de caractères à l'aide de classes simples et de classes composées précédemment définies. Ces définitions se font à l'aide de deux opérateurs qui sont l'union de classes (symbole +) et la différence de classes (symbole -). Il existe aussi une classe composée qui est l'union de toutes les classes, c'est la classe ANY.

Exemple de classe composée

LETCHIF ::= LETTRE + CHIFFRE

c) Les abréviations

Il est possible de définir des abréviations permettant de simplifier l'écriture des expressions régulières.

Exemple d'abréviations

NOMBRE ::= CHIFFRE {CHIFFRE} Les accolades signifiant un nombre quelconque de fois.

d) Les tokens

La définition lexicale se termine par la définition des expressions régulières qui permettent de reconnaître les terminaux génériques du langage.

Exemple de tokens

REEL ::= NOMBRE POINT NOMBRE.

3. PRESENTATION DU LANGAGE SPES

Le langage pour lequel l'éditeur syntaxique et l'environnement d'utilisation devaient être implémentés est un langage de spécification défini au Centre de Recherche en Informatique de Nancy : SPES.

L'objet de ce chapitre est la présentation des concepts absolument nécessaires à la compréhension de la suite de ce travail. Une définition complète du langage SPES, à laquelle nous avons contribué, est présente dans [QUE, 83].

3.0 Préliminaires

Le langage SPES utilise des mots et symboles réservés qui seront soulignés dans le texte.

L'utilisation de minuscules ou de majuscules n'est pas libre. C'est uniquement dans les textes libres, sorte de commentaires, que l'utilisateur peut indifféremment utiliser ces deux classes de caractères.

La syntaxe du langage sera définie dans une forme proche de la BNF. Les conventions de définition sont :
- les symboles "<" et ">" encadrent les non-terminaux du langage
- les caractères " ::= " séparent les membres gauche et droit d'une règle
- le symbole " | " sépare différentes productions de même membre gauche.

Les caractères "[" et "]" sont des terminaux du langage défini, et n'introduisent pas la présence optionnelle.

3.1 Présentation générale du langage

Une spécification SPES comporte deux éléments :

./...

le dossier qui est formé d'une liste d'énoncés et la bibliothèque contenant une liste de descriptions de types.

Un énoncé se décompose lui-même en deux parties : un en-tête d'énoncé contenant un identificateur d'énoncé, une liste de résultats et éventuellement des arguments, et un corps contenant les définitions des objets manipulés dans l'énoncé. Conformément à l'esprit Nancéen ([FIN, 79], [DUB, 81]), chaque objet possède trois définitions permettant de connaître son type, sa sémantique informelle et une sémantique formelle.

Une description de type peut prendre trois formes :

i) la définition d'un type par utilisation conjointe de types génériques (ou constructeurs de type : pile de, suite de, ...) et d'identificateurs de types, à comparer avec la définition TYPE en Pascal.

ii) le langage utilisant les types abstraits de données ([FIN, 79], [LIS, 74]), on peut décrire un type par les opérations définissant les propriétés intrinsèques d'un objet de ce type. Les symboles représentant les opérations seront appelés opérateurs s'ils sont infixés à leurs deux arguments ou fonctions s'ils préfixent leur(s) argument(s).

iii) un type peut être défini par rapport à un autre type, à la définition duquel on ajoute un ou plusieurs opérateurs et fonctions.

3.2 Exemple

L'exemple choisi est une spécification dont le dossier contient un seul énoncé (CALC - CHIFF - AFF - PROD) et dont la bibliothèque contient deux descriptions du type ∅PRODUIT

(1) c - aff - prod CALC-CHIFF-AFF-PROD (no-produit)

(2) c - aff - prod ? le total des ventes du produit de numéro no-produit

(3) c - aff - prod : $N

(4) c - aff - prod = prix-vente (prod) * qte-vendue (prod)

(5) prod ? le produit de numéro no-produit

(6) prod : $PRODUIT

(7) prod tq nu-prod (prod) = no-produit

(8) no-produit ? numéro du produit dont il faut calculer le total des ventes

(9) no-produit : $N

(10) SORTE $PRODUIT == struct (nu-prod : $N , prix-vente : $N , qte-vendue : $N)

(11) SORTE $PRODUIT

(12) OP nu-prod : $PRODUIT ⟶ $N

(13) prix-vente : $PRODUIT ⟶ $N

(14) qte-vendue : $PRODUIT ⟶ $N

(15) groupe : $N , $N , $N ⟶ $PRODUIT

(16) DEC p1 : $PRODUIT

(17) REG groupe (nu-prod (p1), prix-vente (p1), qte-vendue (p1))== p1

Commentaires et explication de l'exemple

- Les lignes 1 à 9 constituent le dossier de la spécification, composé d'un seul énoncé CALC - CHIFF - AFF - PROD.

- la ligne 1 est l'en-tête de l'énoncé et contient le résultat de l'énoncé (c - aff - prod), l'identificateur de l'énoncé (CALC - CHIFF - AFF - PROD) et l'argument (no-produit).

./...

- les lignes 2 à 9 forment le corps de l'énoncé CALC - CHIFF - AFF - PROD. On y trouve des définitions informelles d'objets(lignes 2, 5 et 8), des définitions de type d'objets (lignes 3, 6 et 9) et des définitions formelles (lignes 4 et 7).

Les définitions informelles constituent le lexique des objets utilisés dans l'énoncé et sont introduites par l'utilisation du symbole "?".

Les définitions de type permettant de typer un objet utilisent le symbole ":" suivi d'un identificateur de type constitué par la concaténation du caractère "$" et d'une chaîne de lettres majuscules. $N. représente en SPES le type prédéfini entier, alors que $PRODUIT. est un type nouveau, que l'utilisateur devra par la suite définir.

Les définitions formelles précisent sous forme mathématique une propriété que doit vérifier l'objet, et sont caractérisées par l'emploi de "=" ou de "tq".

Les expressions "prix-vente (prod)", "qte-vendue (prod)" et "nu-prod (prod)" (lignes 4 et 7) sont des références de fonctions définies dans la bibliothèque.

- Les lignes 10 à 17 de la spécification forment la bibliothèque.

- la ligne 10 définit le type $PRODUIT par utilisation du type générique struct paramétré par trois objets de type entier : nu-prod, prix-vente et qte-vendue. Le type générique (ou constructeur de type ou type paramétré) struct permet de définir des types d'une façon analogue au RECORD Pascal.

- les lignes 11 à 17 définissent le type $PRODUIT. par les opérations que l'on peut effectuer sur les objets de ce type.

./...

- les lignes 12 à 15 donnent le profil des opérations définies, c.à.d. le type de leurs arguments et le type du résultat. Ainsi, la fonction qte-vendue a un argument de type ∅PRODUIT, et fournit un résultat de type ∅N .

Les opérations à définir sont les accès aux différents champs d'une structure où un champ est un paramètre du constructeur de type.

Les trois premières lignes correspondent aux trois champs de la structure, et la quatrième sera utilisée pour définir les trois précédentes.

- la ligne 16 indique le nom et le type de l'objet qui sera utilisé pour la définition des opérations.

- la ligne 17 définit simultanément les trois opérations d'accès en signalant que leur utilisation en tant qu'argument de la fonction groupe fournit l'objet qu'elles utilisent comme argument.

3.3 Les énoncés

3.3.1 Structure générale des énoncés

3.3.1.1 Présentation informelle

Un énoncé est composé de deux parties : un en-tête et un corps.

Un en-tête d'énoncé comporte :

1) une liste non vide de résultats;
2) l'identificateur de l'énoncé;
3) éventuellement, une liste parenthétisée d'arguments.

La différence entre résultat et argument est la suivante : le résultat est à destination de l'extérieur de l'énoncé et sa valeur est établie par l'énoncé défini, tandis que l'argument est en provenance de l'extérieur où sa valeur est donc définie.

Un énoncé ne peut être défini qu'une seule fois, et est reconnu par son identificateur, qui ne peut par conséquent être présent que dans un et un seul en-tête d'énoncé.

Le corps d'un énoncé regroupe des commentaires libres et les définitions des objets utilisés dans l'énoncé. L'ordre de ces définitions est arbitraire mais il est conseillé de suivre un ordre produit par une démarche déductive ([PAI, 79]) correspondant à l'algorithme suivant :

- définir le ou les résultat(s) par une relation le(s) caractérisant en fonction d'objets intermédiaires;
- considérer les objets intermédiaires introduits comme de nouveaux résultats, et appliquer récursivement cette démarche;
- s'arrêter lorsque les objets intermédiaires sont en provenance de l'extérieur de l'énoncé.

La majorité des objets utilisés dans un énoncé possèdent dans cet énoncé trois définitions :

- une définition informelle, texte libre décrivant l'objet défini;
- une définition de type indiquant le type de l'objet défini;
- une définition formelle qui indique sous forme mathématique une propriété que l'objet doit vérifier.

Les arguments d'un énoncé ne peuvent avoir de définition formelle, leurs valeurs étant définies à l'extérieur de l'énoncé.

./...

3.3.1.2 Structure générale : formalisation

```
<spécification > ::= <dossier><bibliothèque >
<dossier> ::= < énoncé > |<énoncé> <dossier>
< énoncé > ::= <en-tête> <corps >
< en-tête> ::= < résultats ><identificateur d'énoncé >
             |< résultats><identificateur d'énoncé >
                                         ( < arguments > )
< corps> ::= < commentaire > <corps >
           | <définition > < corps >
           | <commentaire >
           | < définition >
< résultats > ::= < résultat>
                  < résultat > , < résultats >
< arguments > ::= < argument >
                | <argument > , < arguments >
< argument > ::= < identificateur d'objet >
<résultat> ::= <identificateur d'objet >
< commentaire > ::= % <texte libre >
< définition > ::= <définition formelle >
                 | < définition informelle >
                 | <définition de type >
```

3.3.2 Les définitions informelles

3.3.2.1 Présentation informelle

Une définition informelle décrit en langue naturelle un ou plusieurs objets. Cette définition obligatoire contiendra, par exemple, la signification de l'objet et ses propriétés.

Exemples

b'c ? bon de commande reçu du client

x, y ? entiers dont on veut calculer le PGCD
énergie ? l'énergie est égale au produit de la masse et du carré de la chaleur

3.3.2.2 Les définitions informelles : formalisation

<définition informelle> ::= <objets>? <texte libre>
<objets> ::= <objet>
| <objet>, <objets>
<objet> ::= <identificateur d'objet>

3.3.3. Les définitions de type

3.3.3.1 Présentation informelle

En SPES comme dans la plupart des langages informatiques, chaque objet doit être typé. Une définition de type détermine l'expression de type du ou des objets y définis. Une expression de type peut prendre trois formes :

a) L'expression directe

L'expression de type le plus simple consiste à citer un identificateur de type $YY , où YY est une chaîne de caractères majuscules. YY peut être soit l'un des quatre types prédéfinis (à savoir N pour les entiers, R pour les réels, C pour les caractères et B pour les booléens), soit un type que l'utilisateur doit définir lui-même dans la bibliothèque.

b) Utilisation de constructeurs unaires

L'expression de type peut être un constructeur de type unaire, c.à.d. n'ayant qu'un seul argument, qui ici est une expression de type. Les constructeurs unaires sont pile de, file de, ens de (ensemble de) et suite de (permettant la référence d'objets par indexation).

./...

c) Utilisation de constructeur n-aires

L'expression de type peut également être un constructeur de type n-aire, c.à.d. un constructeur paramétré par un nombre indifférent d'arguments. Il existe en SPES deux constructeurs n-aires : struct et structfonct. Le premier permet de construire des objets structurés, ressemblant donc à la notion de RECORD en Pascal. Le second reprend cette notion mais en y ajoutant la propriété d'identificateur pour le premier de ces arguments. Les arguments d'un constructeur n-aire sont appelés champs, et sont composés d'un attribut et d'une expression de type.

Exemples de définition de type

a) avec expression directe

x, y : ØN (x et y sont du type prédéfini N)
w, z : ØTYPE (où TYPE sera défini dans la bibliothèque).

b) avec constructeur unaire

x : pile de /‾ØK_7

y : file de /‾pile de /‾ens de /‾ØT_7_7_7

k : file de /‾struct /‾a : ØA, b : ØB_7_7

c) avec constructeur n-aire

x : struct /‾date : ØDATE, id-cli :ØID-CLI, ligne :
ØLIGNE, montant : ØMON_7

cli : structfonct /‾no-cli : ØNU, nom : ØNOM,
adresse : ØADR_7

où no-cli est identifiant de cli.

./...

3.3.3.2 Les définitions de type : formalisation

<définition de type> ::= <objets> :<type>

<type> ::= Ø <ident majuscule>
| <constructeur unaire>[<type>]
| <constructeur n-aire>[<l-champ>]

<l champ> ::= <champs>
<champs> ; <l-champs>

<champs> ::= <ident minuscule> :<type>

<constructeur unaire> ::= file de
| suite de
| pile de
| ens de

<constructeur n-aire> ::= struct
| structfonct

3.3.4 Les définitions formelles

3.3.4.1 Présentation informelle

Une définition formelle d'un objet précise sous forme mathématique une propriété que doit vérifier l'objet.

Il existe deux catégories de définitions formelles : les définitions implicites et les définitions explicites.

3.3.4.2 Les définitions formelles implicites

Une définition implicite précise la ou les valeurs que peuvent prendre un ou plusieurs objets, en y associant un prédicat. La partie gauche d'une telle définition contient une liste d'objets et la partie droite un prédicat contenant ou non des

occurences d'objets de la partie gauche. Il n'y a aucune contrainte quant à l'existence ou à l'unicité des valeurs pour un objet : on admet que le prédicat ait 0, 1 ou plusieurs solutions (éventuellement une infinité).

Exemples

- La définition implicite "r tq 4>0" admet une infinité de valeurs pour r, car aucune valeur de r ne rend le prédicat "4 0" faux.
- La définition implicite "r tq (r>0) et (r<2)", avec r de type ∅ N, admet une et une seule valeur pour r.
- La définition implicite "r tq (r<0) et(r >0)" n'admet aucune solution.

Les prédicats se trouvant en partie droite des définitions implicites peuvent utiliser, outre les opérateurs logiques habituels (et, ou...) ou plus rares (implications, équivalence), des expressions avec quantificateurs. Les quantificateurs existentiels (ex)et universels (qq)sont à utiliser avec la syntaxe "ex objet : type(prédicat) ":
qq

L'objet lié au quantificateur, dont le type doit être fourni,est local et a la même portée que la quantificateur associé. Il ne peut, pour éviter l'ambiguïté, y avoir d'autre objet de même nom dans l'énoncé.

3.3.4.3 Les définitions formelles explicites

Les définitions explicites sont des structures syntaxiques plus développées que les définitions implicites, et se prêtent davantage à la traduction d'une spécification SPES en un langage de programmation (ce qui est une des étapes ultérieures du projet SPES).

./...

A l'opposé des définitions implicites, une définition explicite d'un objet ne peut produire qu'une et une seule valeur pour cet objet, et de plus un objet ne peut se trouver à la fois en partie gauche et en partie droite d'une définition explicite.

On peut distinguer quatre classes de définitions explicites : les définitions simples, les définitions conditionnelles, les définitions de suites et les définitions externes.

a) Les définitions simples

Une définition simple est une suite d'opérandes séparées par des opérateurs SPES, éventuellement parenthétisés. Les opérateurs SPES sont des opérations infixées à deux arguments, les symboles les représentant sont classiques (+,*,=, et,...) ou nouveaux (+%, *%%, ...)

La notion d'opérande couvre cinq entités différentes :

- les constantes (par exemple : 12, "abc",vrai, ...);
- les objets (par exemple : x, énergie, ...):
- les objets indexés (par exemple : a[i], val [i][j] [5],...)
- Les références de fonctions (par exemple : som (a,b)...).
  Une référence à une fonction introduit un objet dont la définition est celle du résultat de la fonction appliquée aux arguments cités;
- les références d'énoncés (par exemple : VAL (arg 1),...).
  Une référence d'énoncé introduit un objet dont la définition est celle du résultat de l'énoncé appliqué aux arguments cités.

Exemples de définitions simples

. masse* (chaleur^2)

. f(x,y,z) + g(z) + END (x,y)

. x > 0 et 0 + 3 < 12

./...

b) Les définitions conditionnelles

Il est permis de définir un objet de manière conditionnelle. On peut ainsi envisager un nombre arbitraire d'alternatives, et une alternative excluant toutes les précédentes. Une définition explicite conditionnelle s'écrira donc par exemple :

r = si cond1 alors simple1,
    si cond2 alors simple2
    sinon simple3

où cond1 et cond2 sont des définitions simples à résultat booléen, et simple 1, 2 et 3 des définitions simples, que l'on appelera définition d'objet.
Les définitions conditionnelles faisant partie des définitions explicites, aucune occurence des objets définis n'est admise dans le corps de leurs définitions.
Les conditions exprimées sont à lire séquentiellement, et la définition d'objet utilisée est la première dont la condition est validée.
L'exemple précédent est donc équivalent à :

r = si cond1 alors simple1,
    si cond2 et non (cond1) alors simple2
    si non (cond1) et non(cond2) alors simple3

Il est à remarquer que donner un ordre d'évaluation des conditions est malheureux dans un langage de spécification.

c) Les définitions de suites

Il y a deux sortes de définitions de suites : la forme "jusqu'à" et la forme "pour". Ces structures étant destinées à l'application successive de définitions sur des éléments faisant partie de suites, il existe la possibilité dans chacune des deux formes de donner une définition initiale du ou des premiers éléments de la suite.

- La définition "jusqu'à"

  La forme canonique de ce type de définition est "r=jqa cond rep objet = simple init objet = simple".
  La définition présente derrière rep est appliquée successivement, jusqu'à ce que la condition cond soit établie.

- La définition "pour"

  Sa forme canonique est "r=dep indice dans intervalle rep objet = simple init objet = simple".
  La signification de la définition pour est particulière. Il s'agit de définir la suite r dont le domaine d'indiçage est l'intervalle donné. La définition de la suite est réalisée par la définition initiale d'un élément de la suite, et par définition successive des éléments, soit l'un par rapport à l'autre, soit par rapport à d'autres objets.
  L'intervalle n'est donc pas l'intervalle de variation de l'indice, mais le domaine d'indiçage. L'indice repris dans la forme canonique est un objet local à la définition, ne pouvant posséder aucune autre définition dans l'énoncé. C'est un objet qui prendra toutes les valeurs entières ne contredisant pas le domaine d'indiçage.

Exemple :

Soit la définition pour "r = dep i dans 0.. n rep r [i+1]= r [i] * r [i-1]". La suite "r" a un domaine d'indiçage de 0 à n. L'indice "i" prendra toutes les valeurs entières comprises entre 1 et n- 1, car pour les valeurs inférieures à 1, l'expression d'indiçage "i-1" fournit un résultat négatif, ne respectant pas le domaine d'indiçage, et pour les valeurs supérieures à n-1, l'expression d'indiçage "i+1" fournit un résultat supérieur à n, valeurs non autorisées par le domaine d'indiçage.

./...

d) Les définitions externes

Il est possible en SPES de définir un objet en signalant que sa valeur est externe à la spécification, qu'elle est considérée comme une donnée. Ce mode de définition se traduit par la présence du mot réservé donnée en tant que définition explicite de l'objet. Cette notion se rapproche des ordres de lecture des langages de programmation, mais peut également être utilisée pour la définition de constantes dont la valeur n'importe pas durant le processus de spécification. La définition "pi = donnée" peut donc traduire le fait que pi est une constante dont on ne se soucie pas dans la spécification de la valeur ou de la précision.

Remarque :

Dans le cas de définitions simples ou conditionnelles, l'objet défini et chacune des expressions le définissant doivent être de même type.

Pour les définitions de suites, l'objet défini doit être une suite, et il doit y avoir concordance de type entre les deux membres de chacune des définitions successives et initiales.

3.3.4.4 Définition d'une liste d'objets

Dans ce qui précède il est supposé, pour la simplicité de l'exposé, qu'un seul objet était défini par une définition formelle. En réalité, il est possible d'en définir plusieurs.

Pour les définitions implicites, rien n'est à changer : on va définir plusieurs objets par un prédicat commun, qui représente dès lors l'invariant de la liste d'objets.

Pour les définitions explicites, il doit y avoir autant de définitions d'objets que d'objets à définir, et l'ordre d'apparition dans la liste détermine l'ordre de correspondance entre défini et définissant.

Exemple :

r,s,t, = si c1 alors e1, e2,e3
        sinon a+b, vrai,e4.

La définition de r est e1 si c1 est vrai, a+b sinon, la définition de s est e2 si c1 est vrai, sinon vrai et t a comme définition e3 si c1 est vrai, e4 dans l'autre cas.

3.3.4.5 Formalisation de la syntaxe des définitions formelles

```
< définition formelle > ::= <définition explicite >
                           |<définition implicite>
<définition implicite>::= <objets > tq <prédicat>
<définition explicite > ::= <objets> = <corps explicite>
<prédicat> ::= ex <objet>: < type >( < prédicat >)
             | qq < objet > : <type > (< prédicat > )
             | < prédicat simple> ==> < prédicat>
             | <prédicat simple>
< prédicat-simple > := <prédicat simple> <== > <prédicat simple>
                     | < prédicat simple>< opérateur> <prédicat simple
                     | ( < prédicat > )
                     | non (< prédicat > )
                     | <opérande >
< opérateur >::= ou | et | = | >= | =< | <> | < | > | + | +% | +%% | - | -% | -%% |
                 | * | *% | *%% | / | /% | /%% | ^ | ^% | ^%%
```

Remarque :

Dans <opérateur>, cinq classes de priorité sont définies :

- ou, et
- =, ...,>
- +, ...,-%%
- *, ...,/%%
- ^, ...,^%%

< opérande>::= <constante>
| < objet >
| <objet indicé >
| < référence de fonction >
| < référence d'énoncé >
<référence de fonction>::=<ident de fonction >(<l-arguments >)
< référence d'énoncé > ::=<ident d'énoncé > (< l-arguments >)
| <objet >< ident d'énoncé > (< l-arguments >)
<l-arguments> ::= <expr-simple >
<expr -simple > ,<l-arguments >
< corps explicite > ::=<définition de suite >
| <définition conditionnelle >
| < définition simple >
< définition de suite >::=<définition pour>
| <définition pour>< init >
| < définition jqa >
| < définition jqa > <init >
< définition pour >::= dep <objet> dans <inter> rep <ob-bases>=<simples
< ob-bases> ::=<ob-base >
| < ob-base >, <ob-bases >
<inter >::=< opérande >.. <opérande>
<ob-base> ::= <objet >
| <objet indicé >
<définition jqa> ::= jqa <simple> rep <ob-bases> = <simples>
< init> ::= init<ob-bases> =<simples>
< définition conditionnelle>::= <liste-si> sinon <simples >
< liste-si >

./...

<simples> ::= <définition simple>

< liste-si > ::= <si>
| < si> , <liste-si >

< si > ::= si <expr-simple> alors <simples >

< definition simple > ::= < simple>
| < simple > , <définition simple >

< simple >::= donnée | <expr-simple>

< expr-simple >::= <opérande > <opérateur> <expr-simple >
| < opérande >
| (<expr-simple >)

3.4 Les descriptions de type

Face au dossier qui est une collection d'énoncés, se trouve la bibliothèque, collection de descriptions de type. Ces descriptions peuvent prendre trois formes :

a) Les descriptions d'expressions : définition d'un type à partir de constructeurs de types et d'autres types
b) Les descriptions strictes : définition d'un type par les opérations que l'on peut effectuer sur ce type.
c) Les descriptions d'adjonctions: définition d'un type comme étant un autre type auquel on ajoute des opérations.

3.4.1 Les description d'expressions

3.4.1.1 Présentation informelle

Une description d'expression s'exprime à l'aide de constructeur de types et d'autres types.

Exemples

SORTE ØT == pile de /‾file de /‾ØU_7_7

SORTE ØW== ØN

Les circuits sont interdits dans le graphe

./...

de définition de type, c-à-d que partant du type à définir, et en analysant sa définition, on ne peut retrouver ce type à définir.

Contre-exemple

SORTE ØT == pile de [SU]
SORTE ØU == ens de [ØW]
SORTE ØW == suite de [ØT]

Il est possible de restreindre le domaine du type à définir en lui faisant correspondre un prédicat.

Exemple : La description d'expression suivante définit un type ne contenant que les multiples entiers de 3.

SORTE ØMUL3 == ØN lim x tq ex i:ØN (i*3 = x)

3.4.1.2 Formalisation

<Bibliothèque>::= <description>
| <description> <bibliothèque>
<description> ::= <description d'expression>
| <description stricte>
| <description d'adjonction>
<description d'expression>::= SORTE Ø <ident majuscule>== <descr>
<descr>::=<type>
| <type> lim <objet> tq <predicat>

3.4.2 Les descriptions strictes

3.4.2.1 Présentation informelle

Les types utilisés sont des types abstraits de données. Dans ce cadre, chaque type est défini par un ensemble d'opérations dont le résultat ou des arguments sont de ce type.

./...

A chaque type est associé une description stricte regroupant les fonctions et opérateurs attachés à ce type. Chacun des opérateurs et fonctions possède un profil précisant son domaine, c-à-d le type de chacun de ces arguments, et son codomaine, c-à-d le type du résultat. Dans une description stricte d'un type ne pourront se trouver que des opérateurs et fonctions dont un au moins des résultats ou arguments est du type considéré.

Une description stricte comprend quatre parties :

- un en-tête
- une zone profils
- une zone déclarations
- une zone règles.

- L'en-tête indique le nom du type auquel sont rattachés les opérateurs ou fonctions
- La zone profils contient pour chaque opérateur ou fonction son nom et son profil. Opérateurs et fonctions peuvent être surchargés; cependant à un couple nom, domaine ne peut être associé qu'un et un seul codomaine.
- La zone règles décrit la sémantique algébrique des opérateurs présents dans la zone profils.
- La zone déclaration précise le type des objets introduits dans la définition de la sémantique.

Exemple

On veut définir le type ∅ANNUAIRE en le caractérisant par les opérations que l'on peut effectuer sur un objet de ce type.

On peut distinguer trois classes d'opérations : les opérations d'accès, les opérations de modification, et l'opération de création.

Les opérations d'accès sont -vérifier si un abonné se trouve dans dans un annuaire

-connaître le nombre d'abonnés d'un annuaire

-connaître le numéro d'un abonné dans un annuaire.

Les opérations de modification sont -l'ajout d'un abonné dans un annuaire

-le retrait d'un abonné d'un annuaire.

```
SORTE  Ø ANNUAIRE
  OP   présent : ØNOM, ØANNUAIRE → ØB
       nbab : ØANNUAIRE → ØN
       numéro : ØNOM, ØANNUAIRE → ØNUM
       ajout : ØNOM, ØANNUAIRE, ØNUMTEL →   ØANNUAIRE
       destr : ØNOM, ØANNUAIRE → ØANNUAIRE
       avide :  →  ØANNUAIRE
DEC    ann : ØANNUAIRE
       nom, nom2 : ØNOM
       nutel : ØNUMTEL
REG    nbab(ajout(nom, ann,nutel))==nbab(ann)si present(nom,ann);
       nbab(ajout(nom,ann,nutel))==nbab(ann)+1 si non (présent(nom,
                     ann));
       nbab(destr(nom,ann)) == nbab(ann) si non (present(nom,ann));
       nbab(destr(nom,ann)) == nbab(ann)-1 si présent (nom,ann);
       nbab (avide())==0;
       numéro (nom,ajout(nom2, ann,nutel)) == nutel si nom = nom2;
       numéro(nom,ajout(nom2,ann,nutel))==numéro(nom,ann)
              si nom <> nom2;
       numéro (nom,destr(nom2,ann,nutel))==numero(nom,ann)
              si nom <> nom2;
       numéro(nom,destr(nom2,ann,nutel))==vide()si nom=nom2;
       numéro(nom, avide()) == vide();
       présent(nom,ann)==numéro(nom, ann)< > vide ()
```

./...

3.4.2.2 Formalisation

```
description stricte  ::= SORTE <descr>
                         OP <profils>
                         DEC <déclarations>
                         REG <règles>
<profils> ::= <profil>
            |<profil> <profils>
<déclarations> ::= <déclaration>
                 |<déclaration> <déclarations>
<déclaration> ::= <objets> :$ <ident majuscule>
<règles>::= <règle>
          |<règle> ; <règles>
<profil>::=<fonct ou op>: <id-types> ---> $ <ident majuscule>
          |<fonct ou op> :  ---> $<ident majuscule>
<id-types>::= $<ident majuscule>
            | $ <ident majuscule>,<id-types>
<règles>::= <expr-simple> == <expr-simple>
          | <expr-simple> == <expr-simple> si <exp-simple>
<fonct ou op>::=<ident de fonction>
              |<opérateur sans ou ni et>
<opérateur sans ou ni et>: cfr la définition de <opérateur>
          (section 3.3.4.2) dont on supprime ou et et
```

3.4.3 Les descriptions d'adjonctions

3.4.3.1 Présentation informelle

Les descriptions d'adjonctionspermettent de définir un type en utilisant les opérations autorisées sur un autre, en restreignant éventuellement le domaine de ce dernier, et en ajoutant une ou plusieurs opérations.

Exemple :

Soit à définir un type ayant toutes les opérations autorisées sur les entiers, mais sur lequel on veut en plus la fonction "factorielle"

```
SORTE  ØNOUV adj ØN lim x tq x>= 0
OP  fact : ØNOUV->ØNOUV
DEC  i:ØNOUV
REG  fact(i) == fact (i-1)*i  si i>0;
     fact(0) == 1
```

3.4.3.2 Formalisation

```
description d'adjonction  ::= SORTE Ø <ident majuscule> adj <descr>
                              OP <profils>
                              DEC <declarations>
                              REG <règles>
```

3.5 Portée des identificateurs

- Les identificateurs de types, d'énoncés et de fonctions ont une portée étendue à l'ensemble de la spécification.
- Les identificateurs d'objets présents dans les énoncés ont une portée limitée à l'énoncé dont ils font partie. Les objets liés que l'on retrouve dans les quantificateurs ont une portée localisée à l'expression sur laquelle porte la quantification, et les indices utilisés dans les définitions de suite ont une portée limitée à cette définition.
- Les identificateurs d'objets présents dans les descriptions strictes et utilisés pour définir la sémantique des types abstraits, ont une portée locale à la description stricte.

3.6 Mécanisme de référence d'énoncé

Dans l'en-tête d'un énoncé se trouvent une liste de

./...

résultats, l'identificateur de l'énoncé et éventuellement une liste d'arguments. Lors d'une référence à un énoncé, on fait suivre son nom par des expressions définissant la valeur des arguments.

Référencer un énoncé REFERENCE dans un énoncé REFERENCANT revient à créer un nouvel énoncé répondant aux critères suivants :

- Son en-tête est l'en-tête REFERENCANT;
- La première partie de son corps est le corps REFERENCANT, ou la référence de l'énoncé REFERENCE est remplacée par autant de nouveaux objets que REFERENCE possède de réultats;
- Ces nouveaux objets sont définis par le corps de REFERENCE, où chaque occurence d'un objet résultat est remplacé par un des nouveaux objets, et chaque argument de REFERENCE est remplacé par sa valeur lors de la référence, et en procédant au renommage des objets définis sous le même nom dans REFERENCANT et dans REFERENCE.

Pour que ce mécanisme puisse fonctionner, une règle est introduite interdisant les références cycliques. Des références sont qualifiées de cycliques si l'application du processus développé précédemment ne produit pas un énoncé fini, ce qui arrive si dans une chaîne de références un même énoncé apparaît deux fois.

DEUXIEME PARTIE : LES OUTILS DEVELOPPES

Cette seconde partie présente l'environnement de spécification créé autour du langage SPES, utilisant comme noyau le système MENTOR.

Le premier chapitre présente une session d'utilisation du système MENTOR pour le langage SPES, en précisant les fonctions des outils créés. L'architecture des outils formant l'environnement de spécification y est présentée, de même que leur philosophie de développement.

Dans le second chapitre seront précisés les deux réalisations de base de l'environnement de spécification, à savoir le programme permettant la génération de l'éditeur syntaxique pour SPES, et celui permettant le passage de la représentation interne d'une spécification à sa représentation externe.

Le chapitre suivant traite des outils permettant l'extraction d'informations contenues dans une spécification, afin de créer de nouvelles entités informationnelles d'usage divers.

Le quatrième chapitre présente les analyses sémantiques effectuées dans une spécification SPES syntaxiquement correcte.

Le dernier chapitre est consacré aux productions de récapitulatifs qu'il est possible de tirer de la spécification.

1. PRESENTATION GENERALE DE L'ENVIRONNEMENT DE SPECIFICATION

1.1 Scénario d'utilisation

Le SPES-ifieur peut pour entrer au terminal une SPES-ification utiliser l'éditeur syntaxique développé pour SPES sous MENTOR. Tout texte SPES introduit est analysé par MENTOR pour en vérifier la conformité syntaxique, et pour en créer la représentation interne.

Le passage de la représentation interne à la représentation textuelle est obtenu par l'intermédiaire du décompilateur.

Lors de la création d'un énoncé, l'utilisateur peut invoquer une procédure de création assistée, destinée à récupérer dans le dossier (liste d'énoncés) des informations relatives aux résultat(s) et argument(s) de l'énoncé à créer. Les informations récupérées sont les définitions de type et les définitions informelles des deux catégories d'objets.

Le texte obtenu pourra dès lors être complété par l'utilisateur qui établira une relation entre résultat(s) et argument(s), en utilisant éventuellement des objets intermédiaires.

L'énoncé achevé, le spécifieur peut invoquer un processus qui enclenche trois fonctions qui permettent respectivement le contrôle des règles sémantiques, la manipulation d'informations relatives aux références d'énoncé, et enfin l'archivage de l'énoncé.

L'éditeur syntaxique ne vérifiant que la cohérence syntaxique de l'énoncé, il est nécessaire de vérifier le respect des règles sémantiques du langage. Ainsi, il est vérifié que

./...

chaque objet introduit dans un énoncé possède une définition de type, une informelle et une formelle, que le type du résultat d'une expression correspond à l'attente, ...

D'autres propriétés ayant trait à la logique d'une spécification sont vérifiées. Par exemple, il est contrôlé que tout objet introduit dans un énoncé contribue au(x) résultat(s) de cet énoncé.

La deuxième fonction attachée à ce processus est la gestion de messages, les pseudo-commentaires, reprenant sous forme textuelle les relations existant entre énoncés, à savoir la référence d'un énoncé par un autre. Ces pseudo-commentaires, insérés dans les énoncés référencés, permettent lors de la lecture d'un énoncé de connaître les énoncés le référençant, ainsi que le statut de cette référence (c.à.d. son contexte).

La dernière fonction enclenchée consiste à archiver l'énoncé, c.à.d. à insérer celui-ci dans le dossier.

Afin de corriger les erreurs sémantiques signalées ou pour modifier le texte d'un énoncé, l'utilisateur peut récupérer un énoncé archivé par une procédure spécifique, enclenchant également le processus de gestion de pseudo-commentaires.

L'utilisateur peut également effectuer à tout moment des contrôles sémantiques sur l'ensemble du dossier. Outre les contrôles déjà mentionnés, il peut être vérifié que tous les énoncés référencés font partie du dossier, qu'il n'existe pas de cycles de référence entre énoncés, ...

Face au dossier se trouve dans la spécification une bibliothèque contenant une liste de descriptions de type. L'utilisateur ayant introduit une description de type peut vouloir insérer celle-ci dans la bibliothèque.

./...

Avant cet archivage, certaines règles sémantiques et logiques sont contrôlées. Par exemple, il est vérifié que les types ne se définissent pas entre eux de façon récurrente.

L'archivage entraîne aussi une phase de génération d'opérations, dont la sémantique formelle est définie par des règles de types abstraits, dans le cas où la description de type contient un constructeur de type.

L'utilisateur peut dans un premier temps ne pas se soucier de la description des types qu'il utilise dans le dossier. Un processus d'analyse du dossier lui soumettra des propositions de descriptions pour ces types, propositions basées sur les manipulations effectuées dans le dossier des objets de ces types.

A tout moment, l'utilisateur peut demander la création d'un récapitulatif de profils d'énoncés, décrivant pour chacun d'eux le type de leurs résultat(s) et argument(s).

Une procédure d'impression permet d'obtenir un récapitulatif de la spécification comprenant le dossier, le récapitulatif des profils d'énoncés, et la bibliothèque des types.

1.2 Philosophie de développement

Lors du développement de l'environnement de spécification, l'accent a été mis sur certaines propriétés à vérifier par les outils créés .

La première de ces propriétés est l'aspect non-dirigiste des outils développés. De manière générale, le non-dirigisme se marque par l'absence de séquence d'utilisation imposée pour les différents outils, par le libre choix laissé au spécifieur quant à leur emploi.

./...

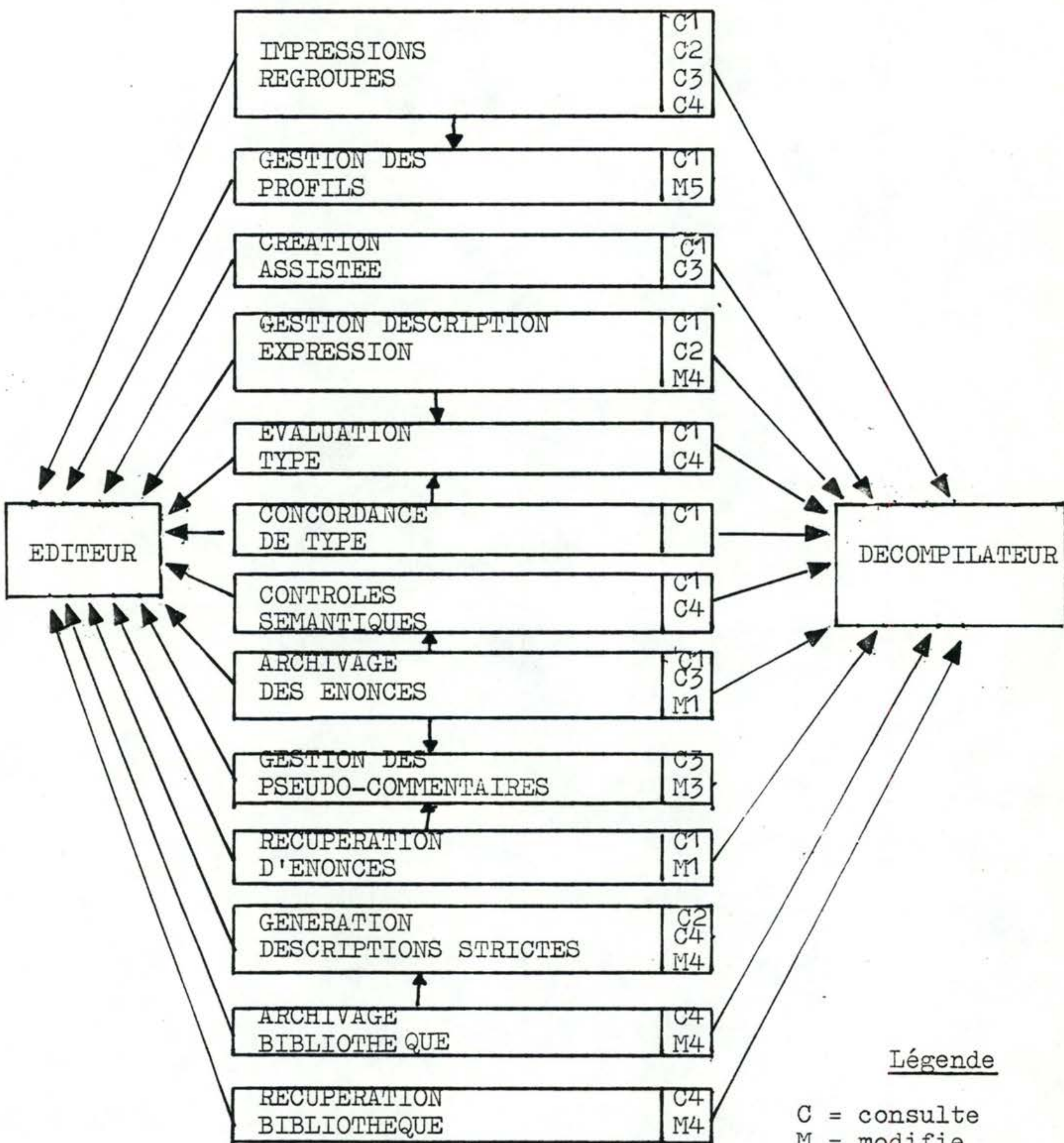
Ce non-dirigisme se traduit par un faible degré de couplage entre les outils, mais aussi par la redondance de certains contrôles.

Une seconde caractéristique de l'environnement est qu'il doit servir d'aide au spécifieur et non d'en limiter l'action. Par exemple, les outils n'imposent pas une correction immédiate des erreurs détectées; le spécifieur restant libre de choisir le moment de la correction. De même, les informations créées sont toujours soumises à l'approbation du spécifieur.

1.3 Architecture globale de l'environnement de spécification

Dans le schéma présenté, ci-dessous, nous avons mis en évidence les outils developpés, les inter-actions existantes (hiérarchie" utilise"), ainsi que les données manipulées par chacun d'eux, exception faite pour l'éditeur et le décompilateur.

Les données utilisées sont numérotées de 1 à 5. Elles peuvent, soit être consultées ou modifiées par ces outils, ces deux possibilités étant exprimées dans le schéma respectivement par les lettres "C" ou "M".



Légende

C = consulte
M = modifie
1 = dossier
2 = méta-types
3 = pseudo-commentaires
4 = bibliothèque
5 = profils

A ⟶ B = A utilise B

./...

2 LES OUTILS DE BASE

Ce chapitre présente les outils de base qui constituent le noyau de l'environnement de spécification pour le langage SPES, à savoir l'éditeur syntaxique pour le langage, généré à partir du programme METAl écrit et le décompilateur.

2.1 Génération de l'éditeur syntaxique pour le langage SPES

Nous avons vu aux chapitres précédents comment était défini le langage SPES et la manière de générer un éditeur syntaxique sous Mentor. Ayant la définition de SPES, il ne nous reste donc plus qu'à écrire le programme Metal qui nous permettra de générer l'éditeur syntaxique pour SPES.

L'écriture de ce programme s'est faite en tenant compte d'une part de contraintes liées à Metal et, d'autre part, de la volonté d'effectuer un maximum de contrôle au niveau syntaxique.

En ce qui concerne les contraintes, la première d'entre-elles impose qu'un opérateur d'arité fixe ne puisse avoir plus de 3 fils. Cette contrainte nous oblige à scinder les opérateurs d'arité supérieurs à trois en deux parties, la deuxième partie étant référencée par un nouvel opérateur. Ceci revient à introduire des règles auxiliaires dans la syntaxe concrète.

Prenons par exemple la règle suivante tirée du langage SPES ;

<définition pour> ::= <u>def</u> <objet> <u>dans</u> <inter> <u>rep</u> <ob-bases> = <simples>

L'opérateur d'arigé fixe qui aurait pu représenter cette règle aurait eu 4 fils, ce qui n'est pas admis en Metal. Nous avons résolu le problème en scindant le membre de droite en deux parties, ce qui a provoqué l'adjonction d'un nouvel opérateur dans la grammaire (c'est l'operateur "cor-it").

Dans le programme Metal nous trouverons :

<u>Syntaxe concrète</u> :

<it-pour> ::= <u>dep</u> <id-minus> <u>dans</u> <inter> <u>rep</u> <corps-iter> ;
it-pour (<id-minus>, <inter>, <corps-iter>)

<corps-iter> ::= <var-ev-lst> #= <for-lst> ;
cor-it (<var-ev-lst>, <for-lst>)

./...

Syntaxe abstraite

it-pour ⟶ ID-MINUS INTER CORPS-ITER;
cor-it ⟶ VAR-EV-LST FOR-LST
it-pour est grâce à cela un opérateur d'arité fixe égal à 3 et cor-it, un operateur d'arité fixe égal à 2.

Une deuxième contrainte de Metal est que l'utilisateur ne dispose pas de moyens d'exprimer dans une règle le caractère optionnel d'un terminal ou non-terminal du langage, si ce n'est en introduisant explicitement une règle par option.

Prenons par exemple la règle SPES suivante :
<en-tête> ::= <résultats > <identificateur d'énoncé> [(<arguments>)]

où <résultats > et <arguments> font référence à des listes respectivement de résultats et d'arguments.

En Metal cette règle va donner lieu à la définition de deux règles, de deux fonctions de construction d'arbre et de deux opérateurs distincts :

Syntaxe concrète

<en-tête> :: <IMIN-LST> <ID-MAJUS>
en tête(< IMIN-LST>, <ID-MAJUS>)
<en-tête> ::= <IMIN-LST> , <ID-MAJUS> (< IMIN-LST >)
en-tête-b(< IMIN-LST> , <ID-MAJUS> , <IMIN-LST >)

Syntaxe abstraite

en-tête ⟶ IMIN-LST ID-MAJUS
en-tête-b ⟶ IMIN-LST ID-MAJUS IMIN-LST

Ces contraintes, nous le voyons tout de suite, ont pour effet d'augmenter le nombre de règles et le nombre d'opérateurs, ce qui complique d'autant la grammaire SPES.

En ce qui concerne les contrôles,nous voulions en effectuer un maximum au niveau syntaxique plutôt que de les faire après vérification syntaxique au moyen de procédures Mentol écrites à cet effet, parce que les erreurs sont signalées au moment où l'édition du texte se fait et donc corrigibles immédiatement.

Reprenons l'exemple traité précédemment en y ajoutant la définition de la liste des résultats. La liste des résultats dans l'en-tête de l'énoncé ne peut être constituée que d'objets simples, c.à.d. des objets sans indices.

Pour traiter cette définition, il y a deux manière de procéder :

- soit au niveau syntaxique, nous admettons d'avoir des listes de résultats dans lesquelles il y a des objets avec indices et nous faisons intervenir ensuite une procédure Mentol qui vérifiera si les objets sont tous des objets simples;
- soit au niveau syntaxique, nous n'admettons pas d'avoir d'objets avec indices dans une liste de résultats. Nous devons alors modifier la règle de définition en précisant qu'il s'agit d'une liste composée d'objets simples. C'est ce que nous avons fait en traduisant <résultats> dans la définition de SPES, en <IMIN-LST> dans le programme Metal.

Ces deux stratégies diffèrent au point de vue résultats : la première ne signalera pas l'erreur au moment de l'édition, tandis que la seconde la signalera. C'est pourquoi nous avons choisi cette méthode parce que l'utilisateur peut corriger son erreur immédiatement après l'avoir commise, ce qui nous semble fortavantageux.

Après ces quelques remarques sur les idées prises en

compte lors de l'écriture du programme Metal, il nous reste à décrire les différentes étapes de sa réalisation.

La première étape consistait à définir la syntaxe concrète et abstraite de SPES dans la forme attendue par Metal. Pour cela, nous avons repris chaque règle décrite dans la définition du langage SPES, déterminé la production et la fonction de construction d'arbre correspondante, en tenant compte des adaptations évoquées ci-dessus. Nous avons ensuite déterminé le profil des opérateurs, c.à.d. préciser pour chaque opérande le phylum auquel elle appartient, puis définir en extension les phylums en énumérant les opérateurs qui en font partie.

Pour terminer l'écriture du programme Metal, il nous fallait définir les points d'entrée dans la grammaire. Notre optique a été de définir un maximum de points d'entrée de manière à permettre à l'utilisateur d'introduire du texte à un haut niveau (par exemple, celui d'un énoncé), comme à un bas niveau (par exemple, celui d'un identificateur). La raison en est que lorsqu'un utilisateur introduit pour la première fois un texte, il le fait généralement à un haut niveau, tandis que lorsqu'il modifie un texte, les éléments à modifier peuvent souvent être de bas niveau.

Ayant écrit le programme Metal pour SPES, il nous reste à le compiler et à générer l'analyseur lexical et syntaxique. La compilation du programme Metal crée les tables de SPES qui sont utilisées par le système Mentor pour manipuler du formalisme SPES, et une description du formalisme SPES dans la forme attendue par le générateur d'analyseur (SYNTAX). A partir de cette description, nous déclenchons la génération de l'analyseur syntaxique pour SPES. Lors de cette étape, le système SYNTAX contrôle la non-ambiguité de la grammaire. En cas d'ambiguité, il faut alors reprendre la définition du langage SPES dans le programme Metal et

./...

essayer de lever ces ambiguités. Ce processus est itéré jusqu'à ne plus avoir d'ambiguités.

Une fois l'analyseur syntaxique créé, nous devons définir l'analyseur lexical, du moins décrire les expressions régulières qui reconnaissent les terminaux génériques de SPES. Rappelons que les terminaux non-génériques (c.à.d. les mots-réservés du langage)seront incorporés automatiquement à la définition de l'analyseur lexical. Cette étape franchie, nous obtenons l'éditeur syntaxique pour le langage SPES.

Ceci nous permet, dès lors, d'éditer du texte SPES tout en créant les arborescences correspondantes. Il nous fallait ensuite écrire le décompilateur pour SPES. C'est ce qui sera décrit dans le paragraphe suivant.

2.2 Le décompilateur

2.2.1 Présentation

Le décompilateur est un programme réalisant l'affichage de la forme textuelle d'un texte à partir de sa forme abstraite (la représentation interne arborescente, créée et manipulée par l'éditeur syntaxique).

Le passage de la seconde forme vers la première est réalisé par la lecture de l'arborescence initiale avec affichage des atomes présents dans celle-ci, et adjonction de mots et symboles réservés (qui se retrouvent dans la forme abstraite) et de caractères particuliers (blancs et sauts de ligne). L'adjonction des mots et symboles réservés est destinée à l'obtention en sortie d'un texte respectant la syntaxe du langage auquel est associé l'arbre à décompiler. L'adjonction des caractères particuliers a pour but de fournir un texte de présentation instructive et de lecture agréable.

Chaque noeud de l'arborescence est caractérisé par son niveau, c-à-d par le nombre de noeuds le séparant de la racine. Le décompilateur va utiliser cette notion pour permettre l'affichage de la structure d'un texte avec un degré de finesse quelconque. Celà est réalisé par les règles suivantes :

- si le niveau d'un noeud est inférieur à une profondeur donnée en paramètre, la décompilation de ce noeud se déroule normalement.
- si le niveau d'un noeud est égal à la profondeur donnée, alors la décompilation de ce noeud consiste en la chaîne de caractères "..." s'il s'agit d'un noeud à arité variable (d'une liste) ou est le caractère "# "s'il s'agit d'un noeud à arité fixe, ou l'atome s'il s'agit d'un noeud atomique.
- si le niveau d'un noeud est supérieur à la profondeur donnée, le noeud n'est pas décompilé.

./...

Exemple

Soit l'énoncé suivant

```
a    EXEMPLE
a :ØS1
a = si non (c=o) alors 5
    si c=o  alors 10
```

La représentation sous forme abstraite de cet énoncé SPES est la figure 2.2.1, où les atomes sont soulignés, à l'opposé des noeuds, et la première ligne indique les niveaux (de 0 à 8).

L'impression de cette arborescence jusqu'à une profondeur 3 donne :

```
# EXEMPLE
... : #
... = ...
```

L'impression jusqu'à une profondeur 5 donne :

```
a    EXEMPLE
a :ØS1
a = si non (#) alors #
    si # alors #
```

2.2.2 Spécification

Le décompilateur est un programme transformant une arborescence SPES fournie par Mentor en un texte, et affichant ce dernier. Le texte obtenu doit être la représentation textuelle de l'arborescence initiale, et doit être en concordance avec la syntaxe du langage SPES, à ceci près que ne sont transformés et affichés que les noeuds dont le nombre d'ascendants est inférieur à un entier fourni.

Niveau

0 1 2 3 4 5 6 7 8

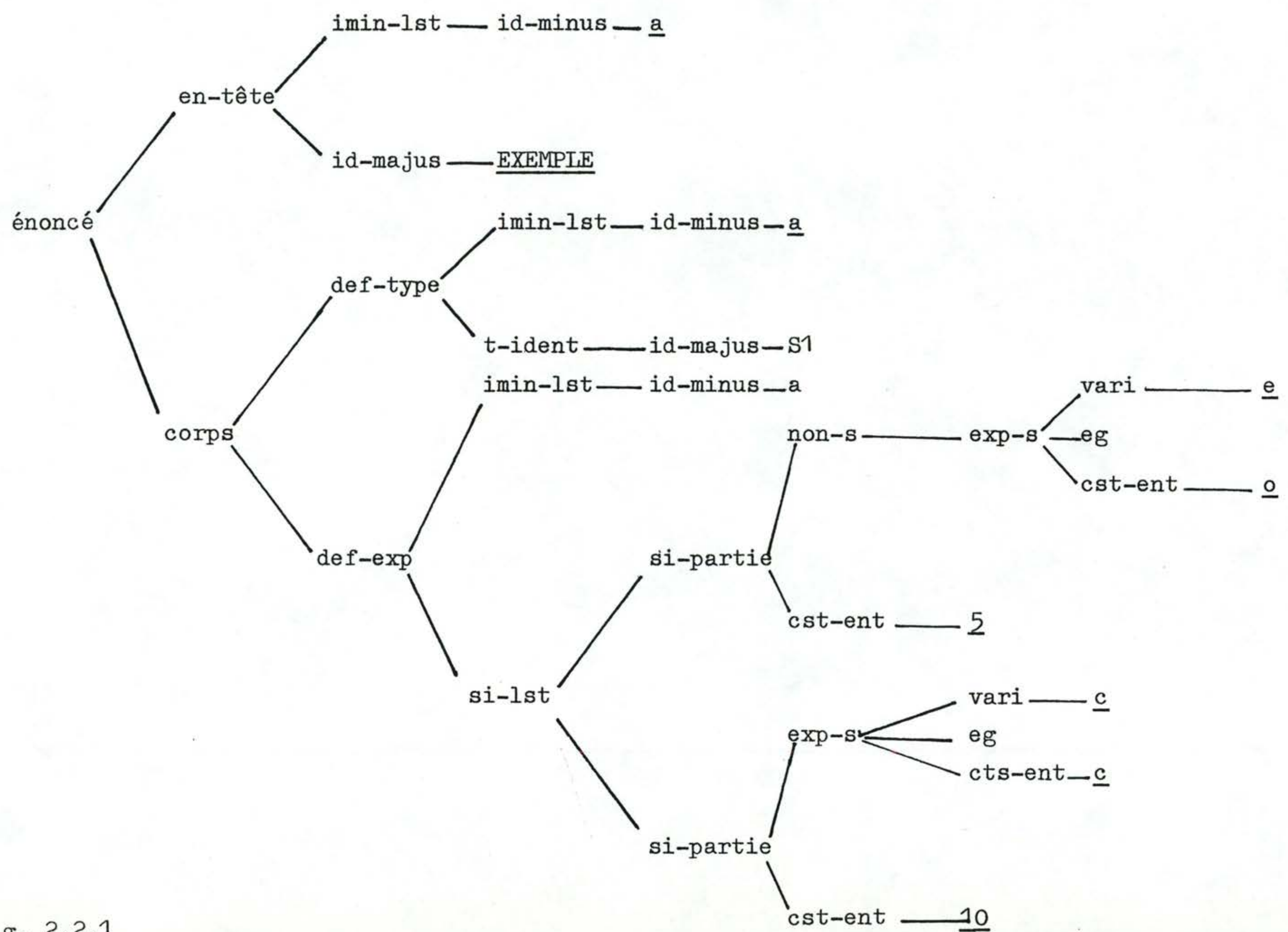


Fig. 2.2.1

./...

Les noeuds dont le nombre d'ascendants est égal à l'entier fourni seront soit affichés normalement s'il s'agit d'atomes du langage, soit sont transformés en la chaîne de caractères "...", s'il s'agit de noeuds à arité variable (les listes), soit transformés en caractère "#" dans le cas de noeuds à arité fixe.

Les noeuds dont le nombre d'ascendants est supérieur à l'entier donné doivent être ignorés.

Classiquement, la décompilation d'un noeud consiste donc en la décompilation de chacun des fils de ce noeud, en insérant entre ceux-ci des symboles réservés afin de fournir un texte répondant à la syntaxe du langage, et en insérant des caractères destinés à obtenir une présentation agréable en sortie. La décompilation s'arrête soit lorsque le noeud est un atome, ou soit lorsque le nombre d'ascendants du noeud est supérieur ou égal à un entier donné, auquel cas la règle précédente est d'application.

Le décompilateur doit être codé en Pascal.

2.2.3 Conception

2.2.3.1 Les outils disponibles

La syntaxe abstraite d'une spécification SPES apparaît sous Mentor comme un type abstrait : la représentation interne des objets est invisible à l'utilisateur, et ne peut être manipulée que via l'utilisation des procédures assurant les fonctions suivantes :

- donner le numéro de production associé à un noeud donné. Ce numéro de production permet d'identifier le noeud de la syntaxe abstraite dont le noeud donné est l'instantiation : num (noeud);
- accéder au ième fils d'un noeud à arité fixe : fils (i, noeud);

- accéder au premier élément d'une liste (noeuds à arité variable) :tête(liste);
- décapiter une liste : couper (liste);
- déterminer si une liste est vide : vide (liste);
- déterminer si un noeud est à arité variable : liste (noeud);
- accéder à la valeur d'un atome : valeur (atome).

2.2.3.2 Construction du décompilateur

Algorithme du processus principal de décompilation d'un arbre
(c'est cette fonction qui est initialement appelée par le système).

```
Decomp (arbre,profondeur)
  Si profondeur < 0
     alors si liste (arbre)
              alors si non vide (arbre)
                       alors afficher "..."
                    fin-si
              sinon afficher "#"
          fin-si
    sinon si liste (arbre)
             alors si non vide (arbre)
                      alors dec-liste (arbre,profondeur-1)
                   fin-si
             sinon dec-fixe (arbre, profondeur-1)
          fin-si
  fin-si.
```

Algorithme du processus de décompilation de listes

```
Dec-liste (liste,profondeur)
(précondition : la liste est non vide)
 décomp (tête(liste),profondeur);
 couper(liste);
 tant que non(vide(liste))
      faire afficher opérateurs et symboles particuliers en fonction
            du type de liste;
```

./...

```
          décomp(tête(liste,profondeur);
          couper(liste);
  finfaire;.
```

Raisonnement du processus de décompilation des noeuds à arité fixe

Dec-fixe (noeud,profondeur)

(Le noeud reçu est de l'une des cinq formes suivantes : atomique, zéroaire, unaire, binaire ou tertiaire. La procédure dec-fixe consiste en une suite de tests portant sur le numéro de production associé au noeud, tests permettant d'invoquer la procédure décompilant les noeuds d'une forme particulière parmi les cinq possibles, en fournissant éventuellement des paramètres fonction du numéro de production.

Les procédures de décompilation des formes atomique et zéroaire demandent un seul argument : respectivement le noeud, et les symboles réserves pour la forme zéroaire. Les trois autres procédures demandent respectivement 4, 5 et 6 arguments. Le premier de ceux-ci est le noeud à décompiler. Le second indique la profondeur jusqu'à laquelle il faut poursuivre le processus. Les derniers sont fonction du numéro du noeud, et représentent les séparateurs qu'il faut utiliser pour la décompilation. Ainsi, par exemple, un noeud binaire de fils f1 et f2, peut posséder trois séparateurs s1, s2 et s3, donnant la règle "s1 f1 s2 f2 s3". Les valeurs des séparateurs sont trouvées en fonction du numéro de la règle.

Les procédures invoquées sont : atome, zéroaire, unaire, binaire et tertiaire).

atome (noeud)

(La valeur de profondeur est positive ou nulle lors de l'appel. Dans les deux cas, la règle demande l'affichage de l'atome)
valeur (noeud).

./...

```
zéroaire (séparateur)
   afficher séparateur.
unaire (noeud,profondeur,sep1,sep2)
   afficher sep1;
   décomp (fils(1,noeud),profondeur);
   afficher sep2.
binaire (noeud,profondeur, sep1,sep2,sep3)
   afficher sep1;
   décomp (fils(1,noeud),profondeur);
   afficher sep2;
   décomp (fils(2,noeud),profondeur);
   afficher sep3.
tertiaire (noeud,profondeur,sep1,sep2,sep3,sep4)
   afficher sep1;
   decomp (fils(1,noeud),profondeur);
   afficher sep2;
   decomp (fils(2,noeud),profondeur);
   afficher sep3;
   decomp (fils(3,noeud),profondeur)
   afficher sep4.
```

3. OUTILS DE GENERATION DE TEXTE SPES

Ce chapitre présente quatre réalisations ayant en commun l'utilisation d'une arborescence SPES fournie par l'éditeur syntaxique, afin, soit de faire ressortir de l'information s'y trouvant de façon plus ou moins cachée, soit d'engendrer du texte fournissant une information nouvelle mais pouvant être construite à partir d'éléments contenus dans l'arborescence initiale.

Le premier processus, basé sur la mise en évidence d'une information connue, gère, lors de l'ajout ou du retrait d'un énoncé du dossier, des pseudo-commentaires, reprenant sous forme de texte lisible, les liens de référence existant entre les énoncés.

La deuxième section présente un processus permettant, lors de la création d'un énoncé, d'extraire des énoncés connus de l'information sur le type et la sémantique informelle des résultats et arguments de l'énoncé à créer.

La troisième section décrit un procédé de création essentiellement différent des autres, en ce sens que l'information qu'il génère est totalement nouvelle. Son objectif est la création, lors du sauvetage d'une description de type, des règles de types abstraits permettant de caractériser ce type.

La quatrième partie de ce chapitre présente un processus qui, par analyse du dossier, va permettre de déterminer la structure des types utilisés, à partir des manipulations effectuées sur les objets de ce type.

L'objectif de ces différentes procédures est l'automatisation de tâches non créatives, ou nécessitant la manipulation d'une grande quantité d'informations. La codification de ces quatre générateurs a été effectuée en Mentol, langage de commande de Mentor, dont les données sont uniquement des arborescences.

./...

3.1 La gestion de pseudo-commentaires

3.1.1 Présentation

Dans les définitions formelles d'objets présentés dans un énoncé, il est permis d'utiliser le ou les résultat(s) d'autres énoncés, via le mécanisme de référence d'énoncé. Trois classes de références d'énoncé peuvent être distinguées : les références conditionnées, conditionnantes et simples.

Il existe en SPES deux définitions d'objets dans lesquelles des conditions interviennent : les définitions conditionnelles, et les définitions de suite de la forme "jusqu'à". Dans ces deux constructions, deux parties sont contenues. La première rassemble les conditions, et la seconde les expressions de définition.

Si une référence d'énoncé est présente dans une expression de définition, on qualifiera cette référence de conditionnée.

Une référence d'énoncé située dans une condition dont l'expression de définition correspondante contient une référence d'énoncé, et dont le résultat est de type booléen est appelée référence conditionnante.

Toute référence d'énoncé présente dans une définition, et n'étant ni conditionnée ni conditionnante est appelée référence simple.

Exemples

Soit les définitions suivantes présentes dans un énoncé :

"r = si COND (x,y) alors VAL (z)
     sinon NEG (z)
 x = SUM(w,k)"

./...

Les références aux énoncés VAL et NEG sont conditionnées, la référence à COND est conditionnante, et SUM est une référence simple.

Lors de la maintenance ou de la lecture d'une spécification de taille importante, ou dans laquelle interviennent de nombreuses références d'énoncés, il devient fastidieux de retrouver pour un énoncé donné tous les énoncés le référençant.

Afin d'éviter ce travail à l'utilisateur, un mécanisme a été créé qui réalise l'insertion dans chaque énoncé de constructions dont la lecture permet de connaître le nom de tous les énoncés référençant cet énoncé, et donne la classe de la référence.

Ces constructions, qui portent le nom de pseudo-commentaires, peuvent prendre l'une des quatre formes suivantes, où les parties soulignées sont des textes fixes et dont la signification figure entre parenthèses.

- A <u>référencé par</u> B (il existe une référence simple de l'énoncé A dans l'énoncé B);
- A <u>est gardé par</u> cond <u>dans</u> B (il y a une référence conditionnée par "cond" de l'énoncé A dans l'énoncé B);
- A <u>est garde positive</u> de C <u>dans</u> B (il existe une référence de l'énoncé A conditionnant la référence de l'énoncé C dans l'énoncé B, et le résultat de A doit être "<u>vrai</u>" pour que la référence à C soit utilisée);
- A <u>est garde négative de</u> C <u>dans</u> B (la référence à A est conditionnante à celle de C dans B, et le résultat de A doit être "<u>faux</u>" pour que la référence à C soit effectuée).

Dans les quatre formes de pseudo-commentaires, on distinguera deux éléments : l'énoncé origine et l'énoncé destinataire du pseudo-commentaire. L'énoncé origine est celui dans lequel existe

la référence ayant entraîné la création du pseudo-commentaire, l'énoncé destinataire étant celui qui est référencé.

Un énoncé sera qualifié d'archivé lorsqu'il est présent dans le dossier.

3.1.2 Spécification

Le gestionnaire de pseudo-commentaire est un processus activé soit par l'ajout, soit par le retrait d'un énoncé du dossier. Il doit satisfaire à l'exigence suivante : à tout moment, l'ensemble formé des énoncés du dossier et des pseudo-commentaires doit être dans un état cohérent, au sens que chaque énoncé du dossier doit contenir tous et uniquement les pseudo-commentaires dont il est destinataire et dont l'énoncé origine est dans le dossier. La réunion de tous les pseudo-commentaires générés présents dans le dossier doit représenter toutes les références aux énoncés du dossier contenues dans tous les énoncés du dossier.

3.1.3 Concepts

Le dossier, constitué d'énoncés dans lesquels sont inclus des pseudo-commentaires est inaccessible à l'utisateur sauf par l'emploi de processus réalisant l'ajout d'un énoncé, le retrait d'un énoncé, l'affichage du texte du dossier...

Supposons l'ensemble formé des énoncés du dossier et des pseudo-commentaires générés dans un état cohérent à un instant donné. Remarquons d'ailleurs que cette cohérence est satisfaite lorsque l'ensemble est vide.

On peut vouloir effectuer sur les énoncés archivés deux opérations qui risquent de modifier cet état de cohérence :

./...

l'insertion et le retrait d'un énoncé du dossier.

Pour que l'extraction d'un énoncé du dossier préserve la cohérence, il faut et il suffit que tout pseudo-commentaire dont il est l'origine disparaisse du dossier

Pour que l'insertion d'un nouvel énoncé dans le dossier préserve la cohérence, il faut et il suffit que :

1) Les pseudo-commentaires dont il est destinataire et dont l'origine est archivée soient créés et insérés dans ce nouvel énoncé;

2) Les pseudo-commentaires dont il est origine et dont la destination est archivée soient créés et insérés dans l'énoncé destination.

Si la cohérence était établie à un moment quelconque, alors une suite aléatoire d'insertions et d'extractions dont l'exécution respecterait les règles précédentes ne modifiera pas l'état de cohérence du système. Comme l'ensemble vide vérifie la cohérence, alors tout ensemble obtenu à partir de lui la vérifiera également.

3.1.4 Choix de conception

Deux types d'implémentation sont envisageables :

a) Chaque insertion d'énoncé donne lieu à un parcours du dossier, afin de déterminer à partir de l'arborescence SPES les pseudo-commentaires à générer dans l'énoncé à insérer. Ensuite il faut à partir de l'énoncé que l'on veut insérer construire les pseudo-commentaires à destination des énoncés du dossier et les y insérer. Ensuite, seulement l'énoncé peut être inséré dans le dossier. Symétriquement, l'extraction d'un énoncé du

dossier donne lieu à l'opération suivante : à partir de l'énoncé que l'on veut extraire, il y a reconstruction de tous les pseudo-commentaires et destruction dans les destinataires archivés de ces pseudo-commentaires.

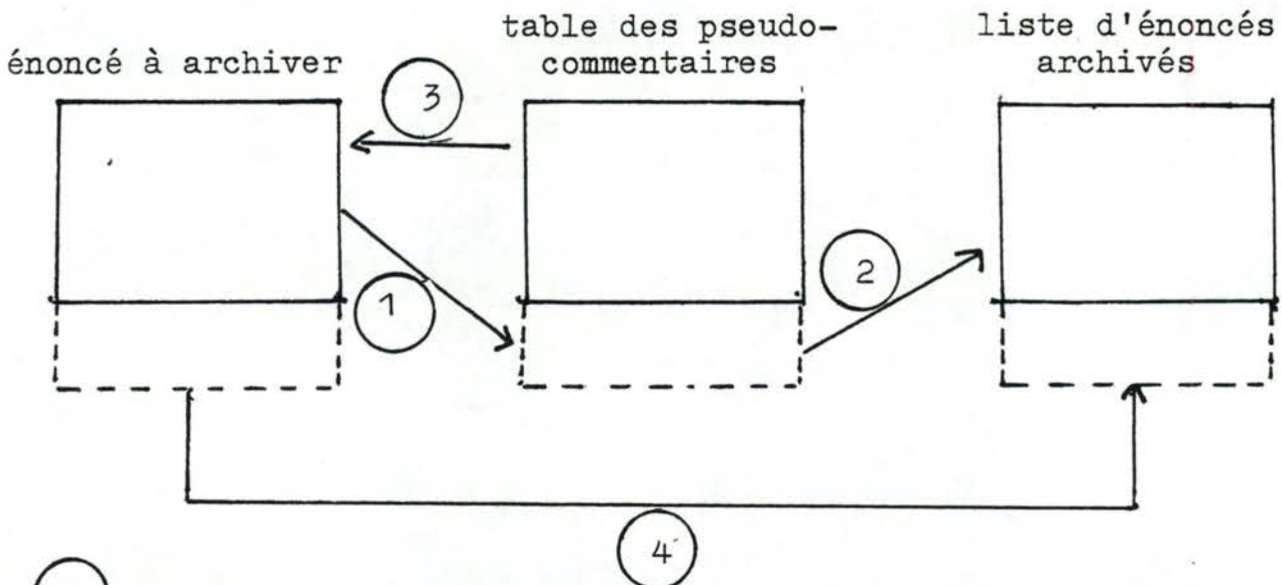
b) Chaque insertion d'énoncé donne lieu à la création de tous les pseudo-commentaires, que le destinataire soit ou non archivé, et à leur stockage dans une table ne contenant que des pseudo-commentaires. Si de plus, le destinaitaire est archivé, il y a copie du pseudo-commentaire dans le corps du destinataire. Ensuite, il faut récupérer dans la table tous les pseudo-commentaires dont l'énoncé à insérer est destinataire.
L'extraction d'un énoncé du dossier entraîne la recherche dans la table des pseudo-commentaires dont il est origine, destruction de ceux-ci, et si le destinataire est dans le dossier, destruction du pseudo-commentaire dans le corps du destinataire.
La table contient donc en permanence tous les pseudo-commentaires générés par tous les énoncés archivés.

La première implémentation permettrait le gain de la place occupée par la table, mais obligerait à chaque insertion et retrait le parcours de l'ensemble du dossier pour y chercher soit des références, soit des pseudo-commentaires.

La seconde implémentation a comme désavantage le supplément de place nécessité, mais les opérations de construction de pseudo-commentaires et de récupération sont plus efficaces, car effectuées une seule fois ou dans un espace plus réduit. Le langage dans lequel ce processus doit être codé étant un langage interprété, et les problèmes de place mémoire négligeables, la seconde implémentation a été choisie car elle semble devoir offir un temps d'éxécution meilleur.

./...

Fig. 3.1.a : l'insertion d'un énoncé dans le dossier



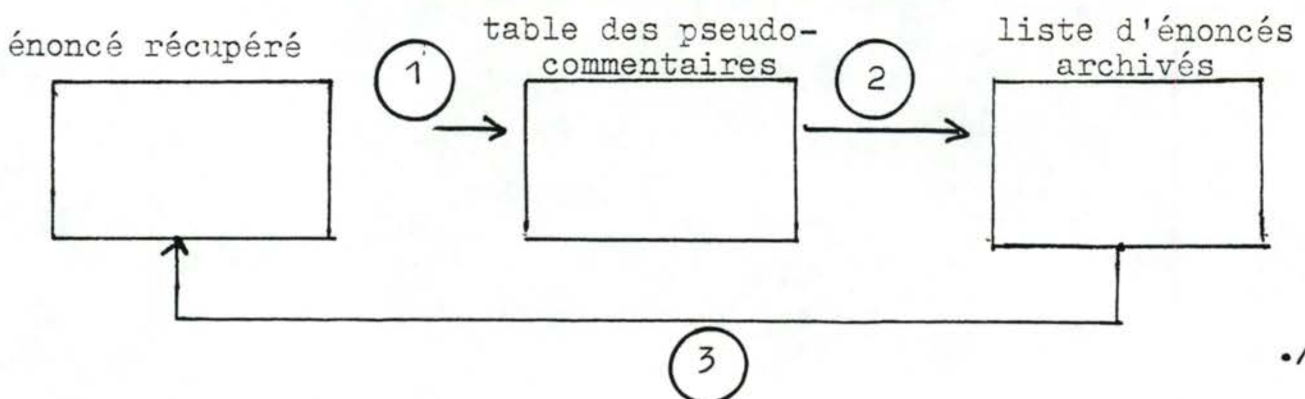
1 génération des pseudo-commentaires dont l'énoncé à archiver est origine. Ces pseudo-commentaires sont insérés dans la table des pseudo-commentaires.

2 Si l'énoncé destinataire d'un pseudo-commentaire est archivé, il y a insertion de ce pseudo-commentaire dans le corps de l'énoncé destinataire.

3 Insertion dans l'énoncé à archiver des pseudo-commentaires dont il est destinataire.

4 Insertion dans le dossier de l'énoncé à archiver.
Les points 1 à 3 font partie de la gestion des pseudo-commentaires; le dernier point étant réalisé par le processus d'archivage.

Fig. 3.1.b : extraction d'un énoncé du dossier



./...

(1) Destruction dans la table des pseudo-commentaires de ceux dont l'énoncé à extraire du dossier est origine.

(2) Destruction dans les énoncés archivés des pseudo-commentaires dont l'énoncé à extraire du dossier est origine.

(3) Renvoi de l'énoncé à extraire.

3.1.4.1 L'insertion d'un énoncé dans la liste d'énoncés archivés

De ce qui précède, on voit apparaître deux actions différentes devant être exécutées lors de l'archivage, et devant être réalisées dans l'ordre :

- la génération des pseudo-commentaires dont l'énoncé à insérer est origine;
- la récupération des pseudo-commentaires dont il est destinataire.

Cette séquence doit être respectée car la première phase pourrait créer des pseudo-commentaires qu'il faudrait insérer dans l'énoncé lui-même. C'est le cas si l'énoncé est à la fois origine et destinataire du pseudo-commentaire, c-à-d si l'énoncé se fait référence à lui-même. Les références cycliques étant interdites, une telle hypothèse est testée avant l'archivage, mais compte-tenu du non dirigisme des procédures écrites, l'erreur sera signalée sans empêcher l'archivage de l'énoncé concerné.

a) La génération des pseudo-commentaires

L'entrée de ce module consiste en l'énoncé qu'il faut archiver, et l'effet de son exécution est l'analyse du texte de l'énoncé, afin de générer les pseudo-commentaires relatifs aux références reconnues dans ce texte, d'insérer ceux-ci dans la table

./...

des pseudo-commentaires et si l'énoncé référencé est déjà archivé, dupliquer dans cet énoncé le ou les pseudo-commentaire(s) dont il est destinataire.

1) Préconditions

La première précondition d'exécution est qu'il ne peut y avoir dans les énoncés archivés un énoncé de même nom que l'énoncé que l'on veut archiver.

De plus, le processus de génération donnera des résultats corrects si, et seulement si, lorsqu'il y a un appel d'énoncé dans une condition , cet appel est le seul élément de la condition (interdiction des conditions hybrides).

La première précondition est destinée à éviter de trancher l'ambiguïté suivante : s'agit-il d'une nouvelle version de l'énoncé archivé, on s'agit-il d'un nouvel énoncé d'identificateur incorrect ?

La seconde précondition a pour but de faciliter la tâche de génération en n'acceptant qu'un seul schéma pour les conditions, schéma permettant la génération plus facile du pseudo-commentaire "est garde positive" ou "est garde négative".

2) Concepts

Lors de la génération de pseudo-commentaires relatifs à des définitions conditionnelles, chaque alternative peut s'exprimer sous forme de sa conjonction avec la négation des alternatives précédentes.

L'on appelera condition-cumulée, la condition contenant l'explicitation complète du contexte, à savoir la conjonction de la condition initiale avec la négation des alternatives précédentes.

./...

Exemple

Les deux définitions conditionnelles suivantes sont équivalentes, la partie condition de la seconde étant une condition cumulée.

"r = si A alors B,
    si C alors D
    sinon E "

"r = si A alors B
    si non (A) et C alors D
    si non (A) et non (C) alors E"

- Afin de ne pas surcharger tant la table de pseudo-commentaires que le corps des énoncés, on ne peut avoir deux pseudo-commentaires identiques tant dans la table que dans les énoncés.

  Exemple

  Les références suivantes, figurant dans un même énoncé, ne donneront lieu qu'à la création d'un seul pseudo-commentaire :

  "r = A(b)
   t = A(c) + A(x) + w".

3) Exemple

   Soit dans un énoncé de nom RAC, le texte suivant :

   "r = A(x,y)
    x = si B(a) alors (b),
        si D(j,k) alors E (e,m,n),
        si F (o,p,23) alors G ("vrai",q)
        sinon H (s,t,u)"

   Après une transformation basée sur le concept de cumul des conditions, ce texte est transformé en

   "r = A(x,y)
    x = si B(a) alors C(b),
    si non (B(a))et D(j,k) alors E(e,m,n),
    si non (B(a))et non (D(j,k))et F(o,p,23) alors G("vrai",q),
    si non (B(a)) et non (D(j,k)) et non (F(o,p,23))alors H(s,t,u)"

./...

Les pseudo-commentaires générés sur base de ce texte seront les suivants :

"A référencé par RAC

C est gardé par B(a) dans RAC

B est garde positive de C dans RAC

E est gardé par non(B(a)) et D(j,k) dans RAC

B est garde négative de E dans RAC

D est garde positive de E dans RAC

G est gardé par non(B(a)) et non (D(j,k)) et F(o,p,23) dans RAC

B est garde négative de C dans RAC

D est garde négative de G dans RAC

F est garde positive de G dans RAC

H est gardé par non (B(a)) et non (D(j,k)) et non(F(o,p,23)) dans RAC

B est garde négative de H dans RAC

D est garde négative de H dans RAC

F est garde négative de H dans RAC"

Ces pseudo-commentaires seront donc inscrits dans une table et, en plus, dans le destinataire correspondant si celui-ci est déjà archivé.

4) Algorithme

Procédure globale : génération-référence (énoncé-à-archiver)

* génération-référence (énoncé à archiver)

détruire tous les pseudo-commentaires contenus dans énoncé-à-archiver;

pour chaque définition formelle def-for de énoncé-à-archiver

faire si def-for est une définition conditionnelle

```
                          alors analyse-conditionnelle(def-for,nom de l'énoncé
                                  à archiver)
                          sinon si def-for est une définition "jusqu'à"
                                   alors an-jusqu'à(def-for,nom de l'énoncé
                                    à archiver)
                                   sinon pour chaque référence ref
                                           de def-for
                                          faire ref-simple (ref,nom
                                                de l'énoncé à analyser,
                                                pseudo);
                                                insérer (pseudo)
                                          fin-faire
                                   fin-si
                          fin-si
                    fin-faire.
```

* insérer (pseudo)

```
  si pseudo n'est pas dans la table
     alors insérer pseudo dans la table des pseudo-commentaires;
           si le destinataire de pseudo est archivé
              alors insérer pseudo dans le corps du destinataire
           fin-si;
fin-si.
```

* ref-simple (ref, nom, pseudo)

(génère un pseudo-commentaire contenu dans pseudo de la forme "ref référencé par nom")

* an-jusqu'à (def, nom)

(an-jusqu'à génére des pseudo commentaires à partir des références d'énoncés contenues dans "def", qui est une définition de suite de la forme "jusqu'à" contenue dans un énoncé d'identificateur "nom".

./...

Les définitions de suite "jusqu'à" ont la forme "jqa cond rep defin1 init defin2")

Si def contient au moins une référence d'énoncé dans defin1 ou defin2
alors pour chaque référence d'énonce ref de defin1 ou de defin2
faire ref-conditionnée (ref,non(cond),nom,pseudo);
insérer (pseudo);
pour chaque référence d'énoncé refcond de cond
faire ref-conditionnante(refcond,non(cond),
ref,nom, pseudo);
insérer(pseudo);
fin-faire;
fin-faire;
sinon pour chaque référence d'énoncé ref de cond
faire ref-simple (ref,nom, pseudo);
insérer (pseudo);
fin-faire;
fin-si.

* analyse conditionnelle (def,nom)
(génére les pseudo-commentaires relatifs aux références d'énoncé présentes dans def, qui est une définition conditionnelle.
Une définition conditionnelle est formée d'une suite d'alternatives, et, éventuellement, d'une alternative excluant toutes les autres.
Les alternatives contiennent une partie condition et une partie définition d'objets .
La forme des conditionnelles est :
"si cond alors def-obj,
...
si cond alors def-obj
sinon def-obj")
cond-cumulée ← vide;

./...

```
pour chaque alternative alter de def
     faire si cond-cumulée = vide
              alors analyse-alter(cond de alter,def-obj de alter,nom);
                    cond-cumulée non (cond de alter);
              sinon analyse-alter(cond-cumulée et cond de alter, def-
                      obj de alter, nom);
                    cond-cumulée ← cond-cumulée et non(cond de alter)
              fin-si;
     fin-faire;
si il y a une alternative-excluante alt-exc
   alors analyse-alter (cond-cumulée, déf-obj de alt-exc,nom);
   fin-si.
```

* analyse-alter (cond,def-obj,nom)

```
  si def-obj contient au moins une référence d'énoncé
     alors pour chaque référence d'énoncé ref de def-obj
          faire ref-conditionnée (ref,cond,nom,pseudo);
                insérer (pseudo);
                pour chaque référence d'énoncé refcond de cond
                     faire ref-conditionnante(refcond,cond,ref, nom,
                                              pseudo);
                           insérer(pseudo);
                fin-faire;
          fin-faire;
  sinon pour chaque référence d'énoncé ref de cond
          faire ref-simple (ref,nom,pseudo);
                insérer (pseudo);
          fin-faire
  fin-si .
```

* ref-conditionnée(ref, cond, nom, pseudo)

(génère un pseudo commentaire contenu dans pseudo de la forme "ref est gardé par cond dans nom")

./...

* ref-conditionnante(refcond,cond,ref,nom,pseudo)

(si le nombre de négations logiques obtenu en partant de refcond dans l'arborescence cond et en remontant jusqu'à la racine de cond est pair, génère un pseudo-commentaire contenu dans pseudo de la forme "refcond est garde positive de ref dans nom"
Si le nombre de négations est impair, le pseudo-commentaire "refcond est garde négative de ref dans nom" est généré et renvoyé dans pseudo.)

3.1.4.2 Extraction d'un énoncé de la liste d'énoncés archivés

En ce qui concerne la gestionnaire des pseudo-commentaires, l'extraction d'un énoncé du dossier entraine .
- la destruction des pseudo-commentaires dont il est l'origine dans les énoncés archivés (pour ne pas violer la contrainte de cohérence de l'ensemble formé par les énoncés archivés et les pseudo-commentaires qu'ils contiennent, à savoir que ces derniers doivent avoir pour origine un énoncé archivé)
- la destruction des pseudo-commentaires correspondants dans la table des pseudo-commentaires (afin de ne pas violer la règle définissant cette table comme contenant tous les pseudo-commentaires générés par tous les énoncés archivés, et uniquement ceux-ci).

Algorithme

(Cet algorithme est facilité par la forme standard des pseudo-commentaires, à savoir que la première partie d'un pseudo-commentaire reprend toujours son destinataire, et la dernière partie représente l'origine du pseudo-commentaire).

```
pour chaque pseudo-commentaire de la table
faire si l'énoncé à extraire est l'origine de ce pseudo-commentaire
        alors si le destinataire du pseudo-commentaire est archivé
                  alors détruire le pseudo-commentaire dans le
                                destinataire;
                  fin-si;
                  détruire le pseudo-commentaire dans la table;
      fin-si;
fin-faire;
```

3.2 Création assistée de parties d'énoncés déjà référencés

3.2.1 Opportunité

Il arrivera un moment dans le travail du concepteur où il sera amené à éditer un nouvel énoncé et à l'adjoindre ensuite au dossier existant en mémoire. Certains éléments de ce nouvel énoncé peuvent être trouvés dans les références à l'énoncé et dans les spécifications des énoncés où elles apparaissent, pour autant, bien entendu, que ces références existent dans le dossier. Une référence est accompagnée d'une description des résultats et des arguments y figurant. Ces objets doivent évidemment être mis en correspondance avec ceux décrits dans l'en-tête du nouvel énoncé. Dès lors, les définitions informelles et les définitions de type associées à ces objets et figurant dans l'énoncé où apparaît la référence, seront aussi celles des objets décrits dans l'en-tête du nouvel énoncé. Nous pourrons donc générer automatiquement les définitions informelles et les définitions de type pour tous les objets décrits dans l'en-tête du nouvel énoncé en fonction des définitions correspondantes dans l'énoncé où apparaît la référence. L'utilisateur ne sera donc pas contraint à les donner.

On peut y voir deux avantages :

- Une telle génération permet à l'utilisateur de gagner du temps puisqu'il ne devra pas rechercher dans le dossier les définitions informelles et les définitions de type correspondant aux objets à décrire dans l'en-tête du nouvel énoncé;
- Une telle génération permet d'assurer une certaine cohérence entre les énoncés où apparaissent les références et les énoncés auxquels il est fait référence, puisque les objets à décrire (c.à.d. les objets de l'en-tête du nouvel énoncé), auront la même définition informelle et la même définition de type que ceux décrits dans les références.

./...

3.2.2 Spécification de la procédure

La procédure permettra à l'utilisateur de créer automatiquement une partie de texte d'un nouvel énoncé en fonction des références à l'énoncé, des énoncés où apparaissent ces références et de l'en-tête de ce nouvel énoncé.

3.2.2.1 Structure d'un énoncé SPES

Pour rappel, un énoncé SPES a la structure suivante :

En-tête de l'énoncé	LISTE DES RESULTATS		IDENTIFICATEUR D'ENONCE	LISTE DES RESULTATS
corps de l'énoncé	Liste d'objets	?	texte libre	(définition informelle)
	Liste d'objets	:	type	(définition de type)
	Liste d'objets	tq	prédicat	(définition formelle implicite
	Liste d'objets	=	définition simple définition conditionnelle définition de suite définition externe	(définition formelle **explicite**)

(Les symboles entourés sont des mots-réservés du langage SPES)

Les listes de résultats et d'arguments comprennent un nombre quelconque d'objets, seule la liste des arguments peut être vide. Les résultats et les arguments doivent être des objets différenciés. Chacun de ces objets possède dans le corps de l'énoncé une définition informelle, **une définition de type et une définition formelle.**

Les arguments ont des valeurs définies à l'extérieur de l'énoncé, tandis que les résultats auront une valeur définie à l'intérieur de l'énoncé,par l'intermédiaire de la définition formelle explicite. L'identificateur d'énoncé existe toujours dans l'en-tête de l'énoncé.

Le corps de l'énoncé comprend un nombre quelconque de définitions formelles (implicites ou explicites), de définitions informelles et de définitions de type. Les références d'énoncé apparaissent principalement dans les définitions formelles implicites et explicites. Elles ont la structure suivante :

LISTE DES RESULTATS	IDENTIFICATEUR D'ENONCE AUQUEL IL EST FAIT REFERENCE	LISTE D'EXPRESSIONS

La liste des résultats comprend un nombre quelconque d'objets et peut éventuellement être vide. Les résultats doivent être des objets différenciés. La liste des expressions contient un nombre quelconque d'expressions et peut éventuellement être vide. L'identificateur d'énoncé existe toujours dans la référence.

Une référence apparait :

- dans le prédicat d'une définition formelle implicite.

  Il peut être formé uniquement de la référence

  ex : tq cA(a,c)
  référence à l'énoncé A

  ou d'une expression faisant intervenir cette référence

  ex : a tq cA(a,c)=5 et c > 0
  référence à l'énoncé A
  Expression

./...

- dans le corps d'une définition formelle explicite qui peut être une définition simple, une définition conditionnelle ou une définition de suite.

  Dans le cas d'une définition simple, la référence apparaît seule

  Ex. : a = cB(b,h), ou dans une expression
  (cB(b,h) : référence à l'énoncé B)

  Ex : a = cB(b,h) + 5
  (cB(b,h) : référence à l'énoncé B ; cB(b,h) + 5 : Expression)

Rappelons qu'une définition simple peut aussi être une liste de définitions simples. Dans ce cas, la référence apparaîtra dans au moins une de ces définitions simples, soit seule, soit dans une expression.

ex. : a,b = cB(b,h),c+6
(cB(b,h) : référence à l'énoncé B ; cB(b,h),c+6 : Liste de définitions simples (2 définitions simples))

ou

a,b = cB(b,h)+5,c+6
(cB(b,h) : référence à l'énoncé B ; cB(b,h)+5 : Expression ; cB(b,h)+5,c+6 : Liste de définitions simples)

- dans le cas d'une définition conditionnelle, la référence apparaît
  - soit dans la condition, seule ou dans une expression

    ex : a = si vR(i.d) alors 5 sinon 6
    (vR(i.d) : référence à l'énoncé R)

    ou

    a = si vR(i,d) > 0 alors 5 sinon 6
    (vR(i,d) : référence à l'énoncé R ; vR(i,d) > 0 : Expression)

./...

-soit dans une des définitions simples correspondant aux alternatives de la conditionnelle

```
Ex. : a = si c > 0 alors 5,
          si c = 0 alors qU(j,k)
                         \_____/
                         référence à
                         l'énoncé U
               sinon 6
```

- dans le cas d'une définition de suite, la référence apparaît
  - soit dans la condition d'une définition "jusqu'à" avec ou sans **initialisation, soit seule soit dans une expression.**

```
Ex. : r = jqa rI(v,d) rep r [i] = r[i-1]+1
              \_____/
              référence
              à l'énoncé
                 I
```

ou

```
r = jqa rJ(v,d) > 0 rep r[i]= r[i-1]+1
        \_____/
        référence
        à l'énon-
        cé I
        \__________/
         Expression
```

  - soit dans l'intervalle d'une définition "pour" avec ou sans **initialisation.**

```
Ex. : a = dep hJ(k,l)..6 rep a[i]= a[i-1]+ 1
              \_____/
              référence
              à l'énon-
              cé J
```

  - soit dans une définition simple de la partie "initialisation" d'une définition "pour" ou "jusqu'à"

```
Ex. a = dep 1..6 rep a[i] = a[i-1]+1

               init a[1] = bH(w,z)
                           \_____/
                           référence
                           à l'énoncé
                              H
```

  - soit dans le corps d'une définition "pour" ou "jusqu'à" avec ou sans **initialisation**

Le corps d'une définition itérative est structuré de la façon suivante :

Liste d'objets (=) { définition conditionnelle / définition simple }

La liste des objets peut éventuellement être vide.
La référence apparaît soit dans une définition conditionnelle, soit dans une définition simple.

Ex. : a = dep 1..6 rep a[i]: = si b > 0 alors a[i-1]+1
sinon a[i-1]+bV(z,q)
(bV(z,q) : référence à l'énoncé V.)

3.2.2.2 Structure de l'en-tête et du corps du nouvel énoncé

L'en-tête du nouvel énoncé a la structure de celui d'un énoncé SPES (cf supra). Le nouvel énoncé a la structure d'un énoncé SPES (cf supra) mais le corps de celui-ci ne comprendra, après génération, que les définitions informelles et les définitions de type, des arguments et des résultats décrits sans son en-tête. Les définitions formelles, et éventuellement d'autres définitions (correspondant par exemple aux objets intermédiaires apparaissant dans le corps de l'énoncé), devront alors être introduites par l'utilisateur.

3.2.2.3 Illustration

Pour illustrer ce qui précède, prenons l'exemple suivant. Supposons que l'utilisateur dispose dans le dossier de l'énoncé suivant :

```
comm,val CONS-COM
comm ? bon de commande interne
val ? résultat booléen indiquant si la commande client est valide
      ou non
bc ? bon de commande client
comm : ØBC
val = ØB
bc = ØBONCOM
comm,val = si   COMMANDE-CORR(bc) alors
                CONS-COMM-VAL(bc), vrai
          sinon bc, faux
bc = donnée.
```

./...

Supposons à présent que l'utilisateur désire introduire l'énoncé CONS-COMM-VAL dans la spécification. L'en-tête de cet énoncé pourrait être :
commande-val   CONS-COMM-VAL(bon-de-commande).

En fonction de cet en-tête et de l'énoncé où apparaît la référence à l'énoncé "CONS-COM-VAL" (soit l'énoncé "CONS-COM"), la procédure générera les définitions informelles et les définitions de type suivantes :
commande-val ? bon de commande interne
commande-val : ØBC
bon-de-commande ? bon de commande client
bon-de-commande : ØBONCOM.

Nous observons donc les correspondances suivantes :
.commande-val et comm
.bon-de-commande et bc

3.2.2.4 Contrôles à effectuer

Parmi les éléments cités précédemment, seul l'en-tête du nouvel énoncé doit être introduit par l'utilisateur. Les autres éléments existent ou n'existent pas dans le dossier suivant qu'il y est fait ou non référence au nouvel énoncé. L'en-tête introduit doit être contrôlé :

- Il faut vérifier que l'identificateur attribué au nouvel énoncé est bien unique dans le dossier. S'il ne l'est pas, la procédure doit le signaler et s'interrompre afin d'empêcher la présence, après archivage, de deux énoncés de même nom dans le dossier.

- Il faut déterminer s'il existe dans le dossier au moins un énoncé qui contient une référence au nouvel énoncé. Cette condition est impérative pour la génération des définitions puisqu'elle se fait en fonction de ces énoncés. Si cette condition n'est pas vérifiée

./...

la procédure ne peut donc pas générer des définitions, le signalera et s'interrompera.

- Il faut vérifier que les objets décrits dans les listes de l'en-tête du nouvel énoncé sont tous distincts les uns des autres. Si tel n'est pas le cas, la procédure doit le signaler, permettre à l'utilisateur de modifier l'une ou l'autre des deux listes et effectuer à nouveau le contrôle pour s'assurer du bien-fondé de la modification.

Lorsqu'il existe dans le dossier plus d'une référence au nouvel énoncé, on peut obtenir dans le texte du nouvel énoncé, plus d'une définition informelle et de type pour un objet décrit dans son en-tête. Rappelons que dans le langage SPES, les objets ne peuvent avoir qu'une seule définition de type et une seule définition informelle. Il faudra donc :

- Supprimer les définitions redondantes (c.à.d. des définitions identiques à une définition donnée);

- Signaler à l'utilisateur le cas où un objet a des définitions informelles distinctes, ceci afin de lui permettre de ne faire qu'une définition informelle à partir de ces définitions;

- Signaler à l'utilisateur le cas où un objet a des définitions de type distinctes et supprimer toutes ces définitions.

Le traitement effectué, on sortira le résultat de l'écran.

3.2.3 Conception de l'algorithme

Avant d'aborder la structure générale et le détail de l'algorithme, il convient de signaler au lecteur l'utilisation de la liste des pseudo-commentaires pour le contrôle déterminant

l'existence d'au moins une référence à l'énoncé, et pour sélectionner les énoncés où apparaissent les références au nouvel énoncé.

Rappelons qu'un pseudo-commentaire peut avoir les formes suivantes : (les caractères entourés sont des mots-réservés du langage)

!	IDENTIFICATEUR D'ENONCE	EST GARDE PAR	CONDITION	DANS	IDENTIFICATEUR D'ENONCE

!	IDENTIFICATEUR D'ENONCE	REFERENCE PAR	IDENTIFICATEUR D'ENONCE

!	IDENTIFICATEUR D'ENONCE	EST GARDE POSITIVE DE	IDENTIFICATEUR D'ENONCE	DANS	IDENTIFICATEUR D'ENONCE

!	IDENTIFICATEUR D'ENONCE	EST GARDE NEGATIVE DE	IDENTIFICATEUR D'ENONCE	DANS	IDENTIFICATEUR D'ENONCE

Pour déterminer l'existence d'une référence au nouvel énoncé, il suffit de rechercher dans la liste des pseudo-commentaires l'existence **d'un pseudo-commentaire du type** :

!	IDENTIFICATEUR DU NOUVEL ENONCE	REFERENCE PAR	IDENTIFICATEUR D'ENONCE.

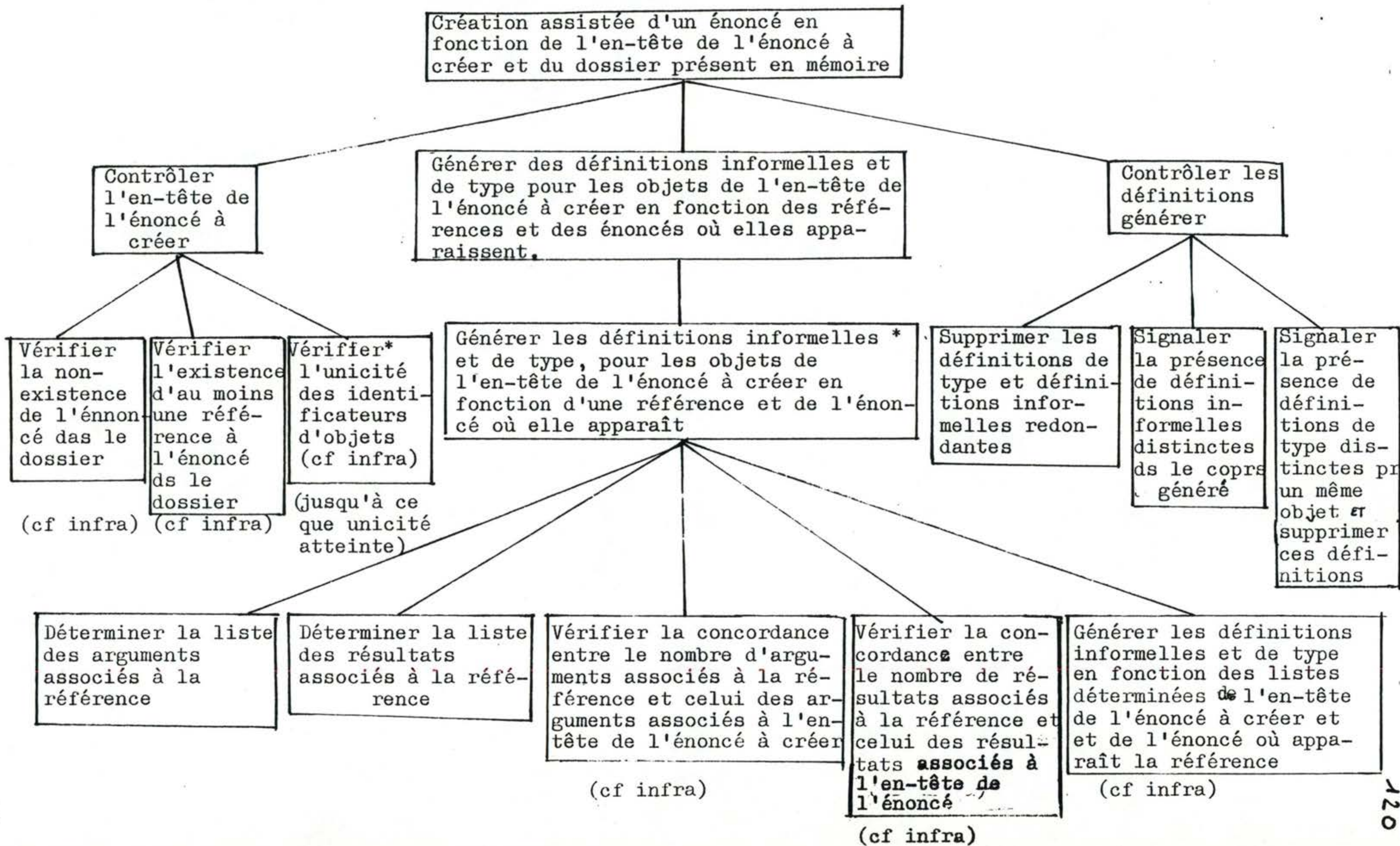
Pour sélectionner les énoncés où apparaissent les références au nouvel énoncé, il suffit de trouver toutes les occurences d'un pseudo commentaire de ce type et d'utiliser l'identificateur du 4ème champ pour déterminer ces énoncés. En fonction de ces informations, il sera relativement aisé de sélectionner les énoncés qui nous intéressent dans le dossier.

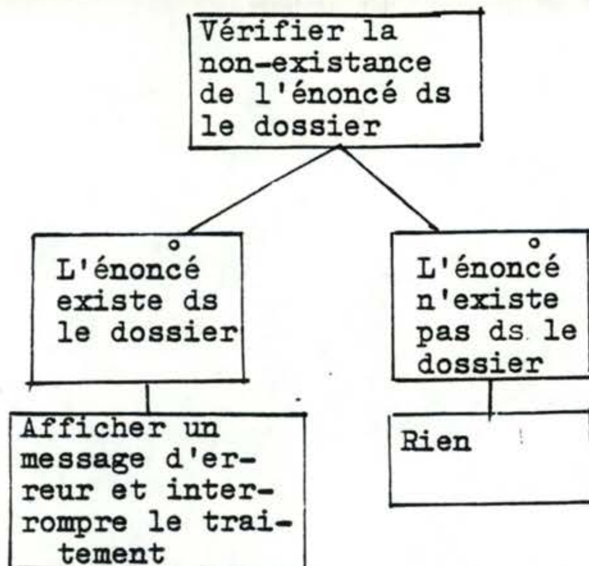
./...

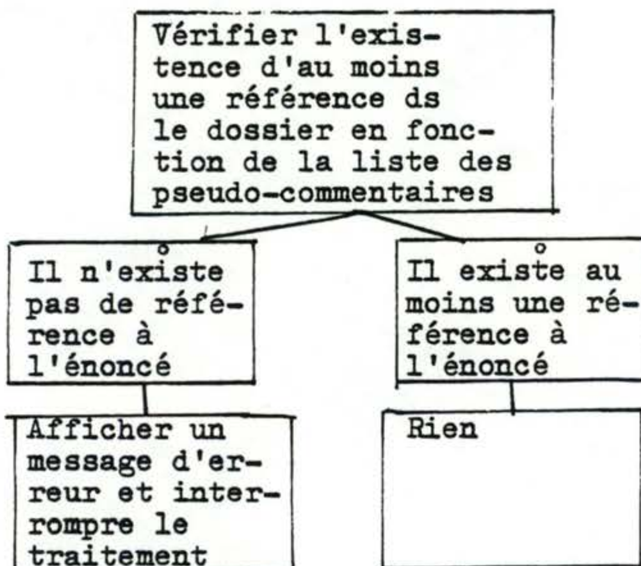
Rappelons que ces pseudo-commentaires existent également dans les énoncés, mais sont bien moins accessibles que ceux détenus dans la liste des pseudo-commentaires. Il faut en effet accéder au corps de chaque énoncé, vérifier l'existence du pseudo-commentaire recherché et ainsi de suite, jusqu'à ce que l'on ait pu en trouver un. Dans la liste des pseudo-commentaires, il suffit de passer en revue, l'un après l'autre, chacun des pseudo-commentaires, tant que l'on ne retrouve pas le pseudo-commentaire recherché. C'est pour la rapidité du traitement que nous avons choisi d'utiliser la liste des pseudo-commentaires plutôt que les énoncés eux-mêmes.

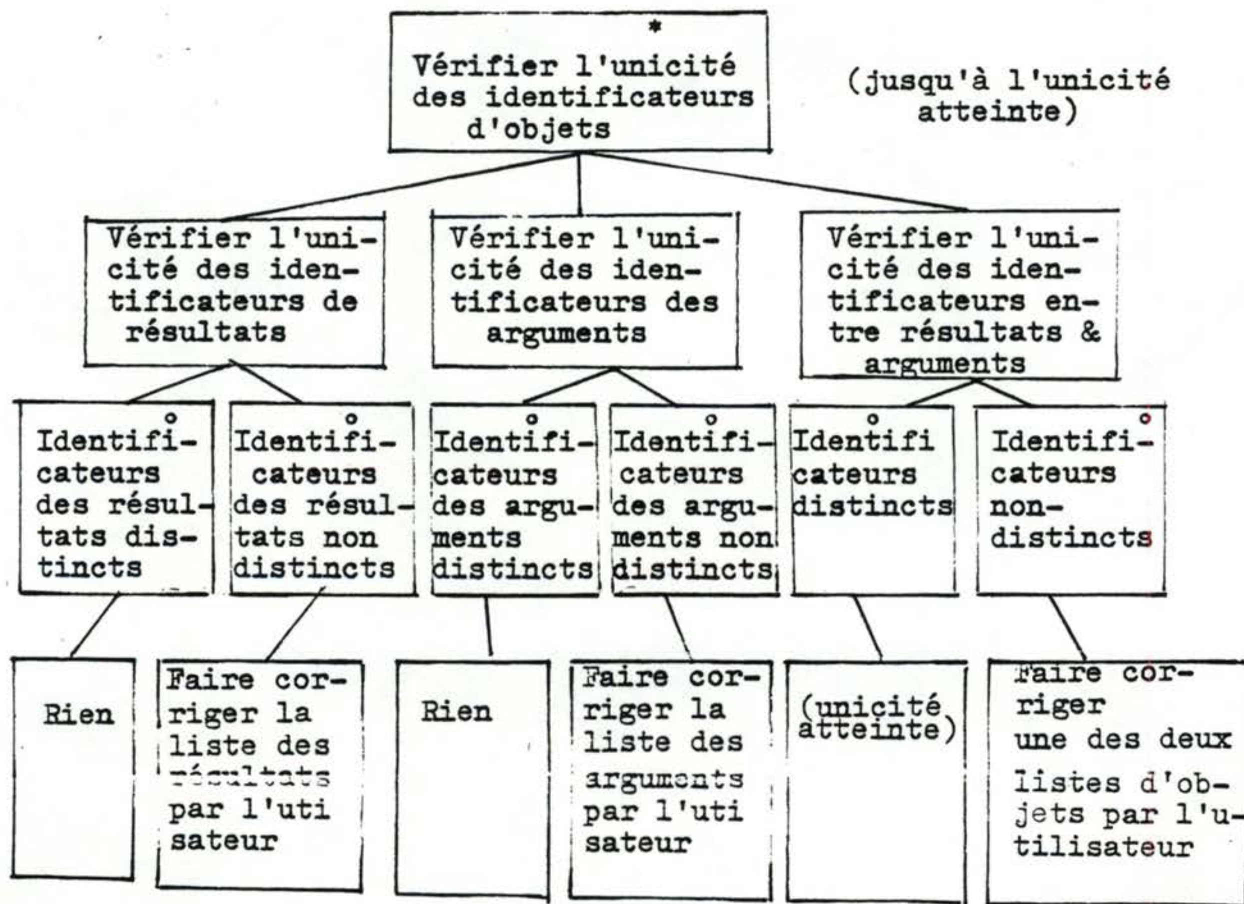
3.2.3.1 Structure générale de l'algorithme

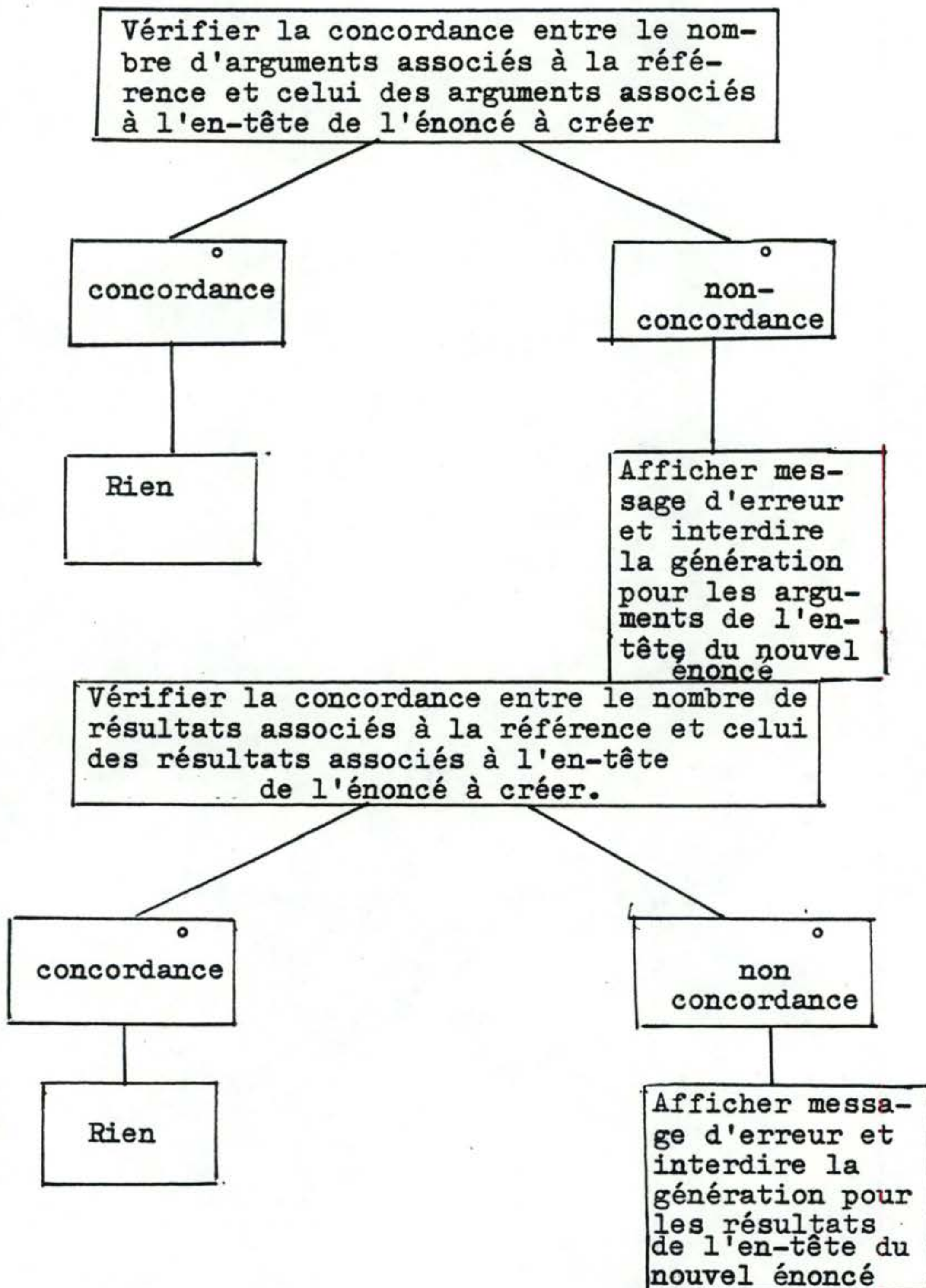
Nous pouvons donner la structure de l'algorithme en utilisant des diagrammes à la Jacskon /Jack,75/

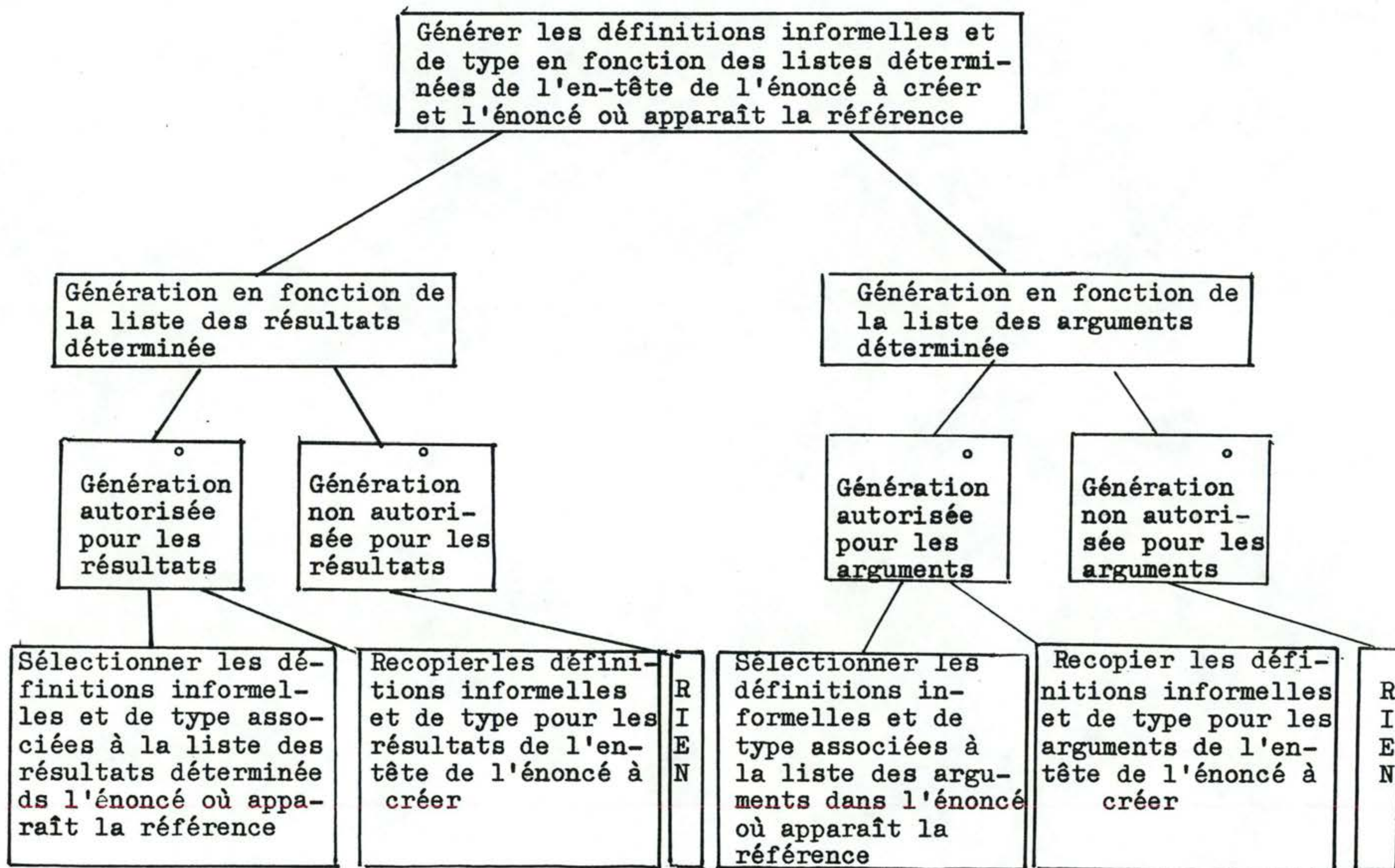



Vérifier la non-existance de l'énoncé ds le dossier
L'énoncé existe ds le dossier
L'énoncé n'existe pas ds le dossier
Afficher un message d'erreur et interrompre le traitement
Rien


Vérifier l'existence d'au moins une référence ds le dossier en fonction de la liste des pseudo-commentaires
Il n'existe pas de référence à l'énoncé
Il existe au moins une référence à l'énoncé
Afficher un message d'erreur et interrompre le traitement
Rien


Vérifier l'unicité des identificateurs d'objets
(jusqu'à l'unicité atteinte)
Vérifier l'unicité des identificateurs de résultats
Vérifier l'unicité des identificateurs des arguments
Vérifier l'unicité des identificateurs entre résultats & arguments
Identificateurs des résultats distincts
Identificateurs des résultats non distincts
Identificateurs des arguments distincts
Identificateurs des arguments non distincts
Identificateurs distincts
Identificateurs non-distincts
Rien
Faire corriger la liste des résultats par l'utisateur
Rien
Faire corriger la liste des arguments par l'utisateur
(unicité atteinte)
Faire corriger une des deux listes d'objets par l'utilisateur


Vérifier la concordance entre le nombre d'arguments associés à la référence et celui des arguments associés à l'en-tête de l'énoncé à créer
°
concordance
°
non-concordance
Rien
Afficher message d'erreur et interdire la génération pour les arguments de l'en-tête du nouvel énoncé
Vérifier la concordance entre le nombre de résultats associés à la référence et celui des résultats associés à l'en-tête de l'énoncé à créer.
°
concordance
°
non concordance
Rien
Afficher message d'erreur et interdire la génération pour les résultats de l'en-tête du nouvel énoncé


Générer les définitions informelles et de type en fonction des listes déterminées de l'en-tête de l'énoncé à créer et l'énoncé où apparaît la référence
Génération en fonction de la liste des résultats déterminée
Génération en fonction de la liste des arguments déterminée
o Génération autorisée pour les résultats
o Génération non autorisée pour les résultats
o Génération autorisée pour les arguments
o Génération non autorisée pour les arguments
Sélectionner les définitions informelles et de type associées à la liste des résultats déterminée ds l'énoncé où apparaît la référence
Recopierles définitions informelles et de type pour les résultats de l'en-tête de l'énoncé à créer
RIEN
Sélectionner les définitions informelles et de type associées à la liste des arguments dans l'énoncé où apparaît la référence
Recopier les définitions informelles et de type pour les arguments de l'en-tête de l'énoncé à créer
RIEN

3.2.3.2 - Détails de l'algorithme

```
PROCEDURE : création assistée
  *FAIRE;
     *CREER nouvel énoncé VIDE;
     *DEMANDER en-tête du nouvel énoncé A l'utilisateur;
     *FAIRE : contrôle de l'en-tête FIN-FAIRE;
     *ADJOINDRE en-tête du nouvel énoncé A nouvel énoncé;
     *FAIRE : génération des def-informelles et de def-de-types
                                                       FIN-FAIRE;
     *FAIRE : contrôle de définitions générées FIN-FAIRE;

     *AFFICHER nouvel énoncé;
FIN-FAIRE;
PROCEDURE : contrôle de l'en-tête
  *FAIRE;
     *DETERMINER DANS le dossier EXISTENCE d'un énoncé
      AYANT MEME identificateur d'énoncé QUE le nouvel énoncé;
      SI enoncé EXISTE
      ALORS *FAIRE :
                *AFFICHER un message d'erreur;
                *INTERROMPRE le traitement;
                FIN-FAIRE
  FIN-SI,
  *DETERMINER DANS liste des pseudo-commentaire EXISTENCE
   d'un pseudo-commentaire du type :! identificateur du nouvel énoncé
   référence-par identificateur d'énoncé;
  SI ce pseudo commentaire N'EXISTE PAS
     ALORS *FAIRE :
                * AFFICHER un message d'erreur;
                *INTERROMPRE le traitement;
                FIN-FAIRE;
  FIN-SI
     JUSQU'A unicité identificateurs d'objets atteinte *FAIRE vérifica-
                                                             tion uni-
                                                             cité des
                                                             identifi-
                                                             cateurs
                                                             d'objets
                                                       FIN-FAIRE;
FIN-FAIRE;

PROCEDURE : vérification unicité des identificateurs d'objets
  *FAIRE;
     *COMPARER les identificateur d'objets de la liste des résultats
               de l'en-tête du nouvel énoncé;
     SI identificateurs d'objets NON DISTINCTS
       ALORS *FAIRE :
                *AFFICHER liste des résultats de l'en-tête du nouvel
                                                            énoncé;
                *DEMANDER nouvelle liste des résultats A l'utilisateur;
                *REMPLACER liste des résultats de l'en-tête du nouvel
                           énoncé PAR nouvelle liste des résultats;
                FIN-FAIRE;
   FIN-SI;
```

./...

```
*COMPARER les identificateurs d'objets de la liste des arguments
          de l'en-tête du nouvel énoncé;
SI identificateursd'objets NON DISTINCTS
   ALORS*FAIRE;
          *AFFICHER liste des arguments de l'ent-ête;
          *DEMANDER nouvelle liste des arguments A l'utilisateur;
          *REMPLACER liste des arguments de l'en-tête du nouvel
                     énoncé;
           PAR nouvelle liste des arguments;
          FIN-FAIRE;
FIN-SI;
*COMPARER les identficateurs d'objets de la liste des résultats
          et de la liste des arguments de l'en-tête du nouvel
          énoncé;
SI identificateurs d'objet NON DISTINCTS
   ALORS*FAIRE;
          *AFFICHER en-tête du  nouvel énoncé
           DEMANDER nouvelle liste d'aguments OU nouvelle liste
                    de résultats A l'utilisateur;
          SI nouvelle liste d'agumennts ALORS
               *REMPLACER liste des arguments de l'en-tête du
                          nouvel énoncé
                PAR nouvelle liste d'aguments;
          FIN-SI;
          SI nouvelle liste de résultats ALORS
             *REMPLACER liste des résultats de l'en-tête du
                        nouvel énoncé
                 PAR nouvelle liste de résultats;
          FIN-SI;
        FIN-FAIRE;
  FIN-SI;
FIN-FAIRE;
```

```
PROCEDURE : génération des def-informelles et des def-types
*FAIRE :
*CREER corps d'énoncé VIDE
 POUR CHAQUE pseudo-commentaire de la liste de pseudo-commentaires
             du type :! identificateur-du-nouvel énoncé référencé-
             par identificateur d'énoncé.
     *FAIRE :
        *RECHERCHER l'énoncé IDENTIFIE PAR identificateur d'énoncé
                                           dans pseudo-commentaire
        DANS le dossier
     POUR CHAQUE référence au nouvel énoncé DANS l'énoncé
        *FAIRE :
         *FAIRE : Détermination de la liste des arguments FIN-FAIRE;
                  associée à la référence
         *FAIRE : Détermination de la liste des résultats FIN-FAIRE;
                  associée à la référence
         *FAIRE : Vérification concordance du nombre d'arguments de
                                            la référence FIN FAIRE;
                  avec celui de l'en-tête nouvel énoncé
         *FAIRE : Vérification concordance du nombre de résultats
                                                  de   FIN-FAIRE;
                  la référence avec celui de l'en-tête du nouvel
                  énoncé
         *FAIRE : Génération des définitions informelles et FIN FAIR
                  de type en fonction de la référence de
                  l'énoncé où elle apparaît
        FIN-FAIRE;
     FIN-FAIRE;
     ADJOINDRE corps d'énoncé A LA SUITE DE nouvel énoncé;
  FIN-FAIRE :
```

1) <u>Détermination de la liste des arguments</u>

Les arguments d'une référence sont toujours définis dans la liste d'arguments située immédiatement à la droite de l'identificateur de la référence.

2) <u>Détermination de la liste des résultats</u>

La détermination des résultats d'une référence pose davantage de problèmes.

En effet, plusieurs listes de résultats peuvent coexister dans une même définition formelle. Il s'agit donc de déterminer dans celle-ci quelle liste de résultats et quels résultats devront être considérés.

Nous passerons ci-dessous en revue les différents cas qui peuvent se présenter lors de ce traitement. Nous utiliserons à cet effet des tables de décisions chaînées.

De manière générale, nous accorderons priorité à la liste des résultats la plus proche de la référence. La raison en est la suivante :

La présence d'une liste de résultats en partie droite d'une définition formelle permet à l'utilisateur d'utiliser éventuellement des objets intermédiaires. Les objets peuvent avoir des définitions informelles plus précises que celles des résultats décrits en en-tête de l'énoncé. Il sera donc préférable d'effectuer la génération en fonction de ces définitions.

Table 1

Une liste de résultats est définie immédiatment à gauche de l'identificateur d'énoncé dans la référance	VRAI	FAUX
Faire correspondre les résultats de cette liste un par un, aux résultats de l'en-tête du nouvel énoncé	X	
Déterminer le contexte où apparaît la référence (table 2)		X

Table 2

La référence apparaît	dans une définition explicite	dans une définition implicite, un indice, une liste d'expressions d'une référence ou une liste d'expressions d'une fonction
Déterminer la nature de la définition explicite (table 3)	X	
(pas de correspondance) → afficher un message et interdire la génération des définitions informelles et des défintions de type pour les résultats de l'en-tête du nouvel énoncé.		X

Table 3 : Nature de la définition explicite

La partie droite de la définition explicite où apparaît la référence est	une conditionnelle	une itération	une liste de définitions simples
Analyser la conditionnelle (table 4)	X		
Analyser l'itération (table 5)		X	
Prendre en considération la liste des résultats décrite en partie gauche de la définition explicite			X
Analyser la liste de définitions simples (table 6)			X

Table 4 : Analyse de la conditionnelle

La référence apparaît dans la partie	condition d'un conditionnelle	Liste de définitions simples d'une conditionnelle
(pas de correspondance) ⟶ Afficher un message et interdire la génération des définitions informelles et des définitions de type pour les résultats de l'en-tête du nouvel énoncé	X	
Prendre en considération la liste des résultats décrite en partie gauche de la définition explicite		X
Analyser la liste des définitions simples (table 6)		X

./...

Table 5 : Analyse de l'itération

La référence apparaît	dans le corps d'itération d'une itération "pour" ou "jusqu'à" avec ou sans initialisation	dans l'initialisation d'une itération "pour" ou "jusqu'à" avec initialisation	dans la condition d'une itération "jusqu'à" avec ou sans initialisation ou ds l'intervall[e] d'une condition "pour"
Analyse du corps d'itération (table 7)	X		
Prendre en considération la liste des résultats décrite en partie gauche de l'initialisation		X	
Analyser la liste des définitions simples		X	
(pas de correspondance) ⟶ afficher un message et interdire la génération des définitions informelles et de types pour les résultats de l'en-tête du nouvel énoncé			X

Table 6 : Analyse de la liste de définitions simples

La définition simple où apparaît la référence est	une expression ne contenant que la référence	une expression contenant plus que la référence
Déterminer les résultats de la liste considérée qui sont associés à la définition simple où apparaît la référence	X	
(pas de correspondance) ⟶ afficher un message et interdire la génération des définitions informelles et de type pour les résultats décrits dans l'en-tête du nouvel énoncé		X

Table 7 : Analyse du corps d'itération

Le corps de l'itétion est une liste de définitions simples	oui	oui	non	non
Le corps de l'itération est une conditionnelle	non	non	oui	oui
Une liste de résultats est décrite en partie gauche du corps de l'itération	oui	non	oui	non
Prendre en considération la liste des résultats décrits en partie gauche du corps de l'itération	X		X	
(pas de correspondance) ⟶ afficher un message et interdire la génération des définitions informelles et des définitions de type pour les résultats de l'en-tête du nouvel énoncé		X		X
Analyse conditionnelle (table 4)			X	X
Analyser liste des définitions simples (table 6)	X	X		

Dans le cas où la référence au nouvel énoncé apparaît dans une liste de définitions simples, une partie seulement des éléments de la liste des résultats à considérer sont associés à la définition simple contenant la référence. Il faudra donc déterminer cette partie. Nous savons que chaque élément de la liste des résultats est associé à un et un seul élément de la liste des définitions simples. Il suffira donc de parcourir en même temps ces deux listes en associant de 1 à n résultats, selon la nature de la définition simple rencontrée.

Deux cas sont à distinguer : soit la définition simple est formée uniquement d'une référence, soit elle est formée d'une expression ou d'une fonction unique.

Définition simple est formée uniquement d'une référence

Dans ce cas, la procédure doit connaître le nombre de résultats rendus par l'énoncé auquel il est fait référence pour déterminer le nombre d'éléments de la liste des résultats à associer à la définition simple. Ce nombre peut être déterminé en comptant le nombre d'éléments contenus dans la liste des résultats de l'en-tête de l'énoncé auquel il est fait référence. Bien entendu, cela suppose que cet en-tête existe soit dans le dossier, soit dans le nouvel énoncé. Dans le cas contraire, l'énoncé auquel il est fait référence n'aura pas encore été introduit. Par conséquent, le nombre de résultats rendus par cet énoncé est indéterminé. Si le cas se présente avant que l'on ait rencontré la définition simple contenant la référence du nouvel énoncé, on ne pourra pas déterminer avec certitude les résultats associés à cette définition. On effectuera alors le même traitement, mais en parcourant les deux listes dans le sens opposé à celui utilisé pour le permier parcours. Si à ce moment-là le même problème se rencontre, la procédure ne sait pas déterminer les éléments de la liste des résultats qui sont associés à la définition simple contenant la référence au nouvel énoncé.

./...

Elle le signalera donc et interdira la génération des définitions informelles et des définitions de type pour les résultats décrits dans l'en-tête du nouvel énoncé.

Définition simple et formée d'une expression ou d'une fonction seule

Dans ce cas, on associera un et un seul résultat de la liste des résultats à cette définition simple,(vis-à-vis de la sémantique, les expressions et les fonctionsne peuvent avoir qu'un et un seul résultat).

En même temps que l'on effectue ces associations, on contrôle que le nombre d'éléments de la liste des résultats est au moins égal à la somme des résultats associés à chacune des définitions simples de la liste. Ce contrôle s'effectue simplement en vérifiant que dans la liste des résultats on dispose toujours d'un nombre suffisant d'éléments pour effectuer l'association. Signalons dans le cas où il y a indétermination du nombre de résultats, nous considérerons que ce nombre est par défaut égal à 1. (Pour rappel, tout énoncé a au minimum un résultat). Si on ne dispose pas d'un nombre suffisant d'éléments, la procédure le signalera et interdira la génération des définitions informelles et des définitions de type pour les résultats décrits dans l'en-tête du nouvel énoncé.

```
PROCEDURE : Vérification du nombre d'arguments de la référence avec
            celui de l'en-tête du nouvel énoncé.
  *FAIRE:
     SI nombre d'arguments associés à la référence
     NE CORRESPOND PAS au nombre d'arguments associés au nouvel
                       énoncé
     ALORS*FAIRE
           *AFFICHER message d'erreur ;
           *INTERDIRE la génération des définitions informelles
                      et des définitions de type pour les arguments
                      dans l'en-tête de l'énoncé;
           FIN-FAIRE;
  FIN-SI;
FIN-FAIRE;
```

./...

```
PROCEDURE : Vérification  du nombre de résultats de la référence
            avec celui de l'en-tête du nouvel énoncé.
   FAIRE:
      SI nombre de résultats associés à la référence
        NE CORRESPOND PAS au nombre de résultats associés au nouvel
                                  énoncé
        ALORS*FAIRE
             *AFFICHER message d'erreur
             *INTERDIRE la génération des définitions informelles et
                        des définitions de type pour les résultats
                        décrits dans l'en-tête du nouvel énoncé.
             FIN-FAIRE;
     FIN-SI;
FIN-FAIRE;

PROCEDURE : Génération des définitions informelles et de type en
            fonction de la référence et de l'énoncé où elle apparaît
  *FAIRE;
      SI génération autorisée POUR les résultats décrits dans l'en-
                                   tête du nouvel énoncé
         ALORS*FAIRE
             *RECHERCHER def-informelles et def-de-type CORRESPONDANT
                        AUX objets de la liste des résultats déterminée;
          POUR CHAQUE définition trouvée
               *FAIRE
                    *RECOPIER définition trouvée A LA SUITE DE
                              corps de l'énoncé;
                    *REMPLACER identificateur d'objets de cette
                               définition
                        PAR identificateur d'objets lui correspondant
                            dans l'en-tête du nouvel énoncé;
               FIN-FAIRE;
          FIN-FAIRE;
      FIN-SI;
      SI génération autorisée POUR les arguments décrits dans l'en-
                                   tête du nouvel énoncé;
         ALORS*FAIRE
             *RECHERCHER def-informelles et def-de-type CORRESPONDANT
                        AUX objets de la liste des arguments déterminée;
          POUR CHAQUE définition trouvée
               *FAIRE;
                    *RECOPIER définition trouvée A LA SUITE DE corps
                              de l'énoncé;
                    *REMPLACER identificateur d'objets de cette défi-
                               nition
                        PAR identificateur d'objets lui correspondant
                            dans l'en-tête du nouvel énoncé;
               FIN-FAIRE;
          FIN-FAIRE;
     FIN-SI;
FIN-FAIRE;
```

./...

```
PROCEDURE : Contrôle des définitions générées
  *FAIRE;
     *COMPARER définitions générées DANS corps de l'énoncé
     SI définition identiques EXISTENT
       ALORS*SUPPRIMER définitions redondantes DANS corps de
                                                  l'énoncé;
     FIN-SI;
     *DETERMINER DANS corps de l'énoncé EXISTENCE de plusieurs
                      définitions informelles POUR un même objet;
     SI plusieurs définitions informelles EXISTENT POUR un même
                      objet
       ALORS*AFFICHER un message d'erreur
     FIN-SI;
     *DETERMINER DANS corps de l'énoncé EXISTENCE de plusieurs
                      définitions de type POUR un même objet;
      SI plusieurs définitions de type EXISTENT POUR un même objet
        ALORS*FAIRE
                 *AFFICHER un message d'erreur;
                 *SUPPRIMER toutes les définitions de type POUR
                                   cet objet;
              FIN-FAIRE;
      FIN-SI;
```

3.2.4 Quelques détails d'implémentation

L'implémentation de l'algorithme s'est faite sous forme d'une procédure MENTOL pour plusieurs raisons :

- Elle permet l'utilisation de l'arbre abstrait créé à partir de la liste des énoncés lors de la phase d'édition. Un avantage de l'arbre abstrait par rapport à la forme textuelle, est qu'il renferme de l'information sur la structure du texte (Ex. : identificateur d'énoncé, référence d'un énoncé, ...), ce qui rend beaucoup plus aisée la recherche de certains éléments du texte.

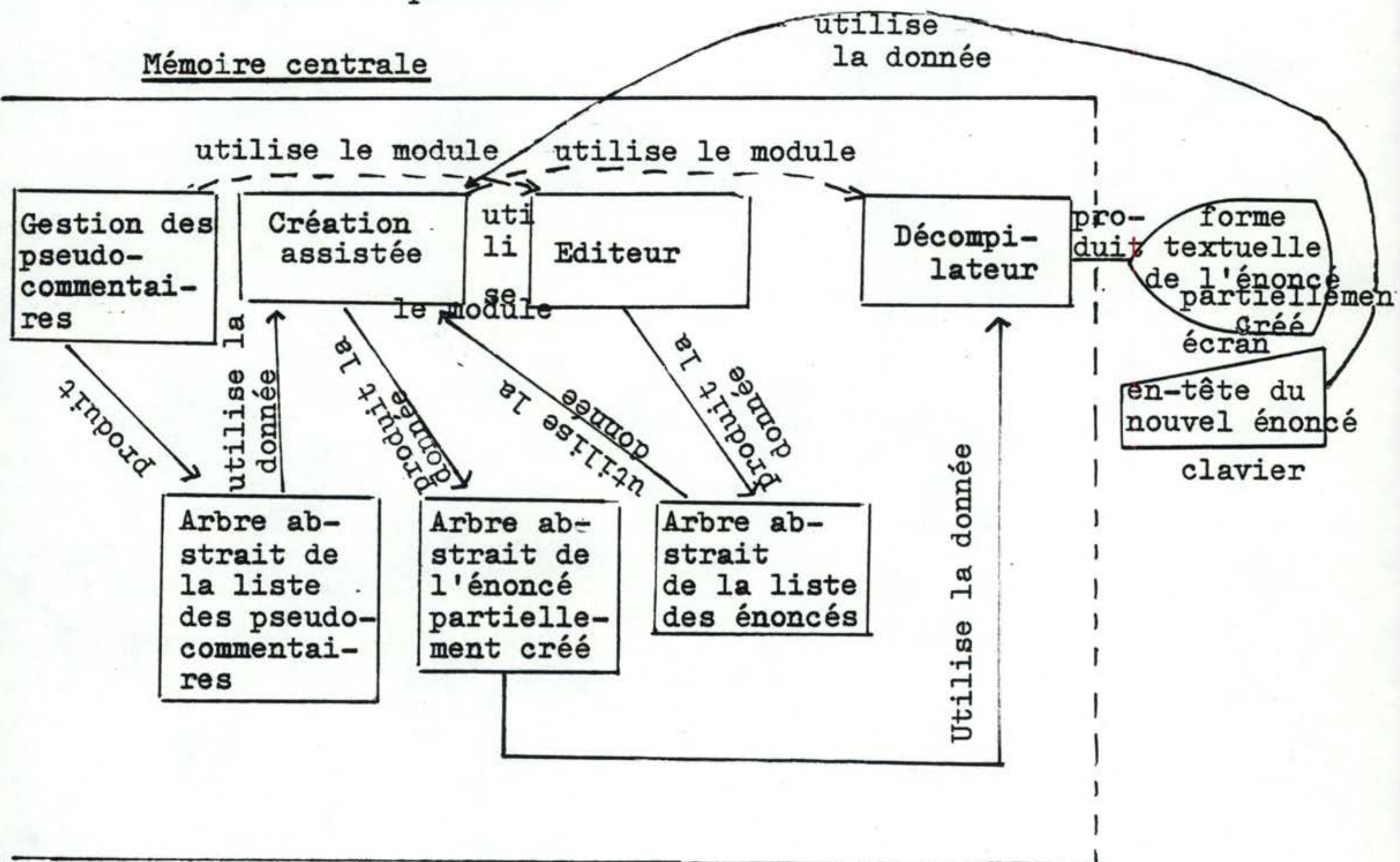
- L'arbre abstrait ne sait être manipulé qu'au moyen du langage MENTOL.

La liste des énoncés, les énoncés et leur composants, la liste des pseudo-commentaires seront représentés en mémoire par des arbres.

La forme textuelle du nouvel énoncé est obtenue par décompilation de l'arbre lui correspondant.

./...

Nous pouvons à présent donner un schéma qui précise le contexte de la procédure



La procédure doit disposer pour s'exécuter :

- de l'arbre abstrait de la liste des énoncés
- de l'arbre abstrait des pseudo-commentaires en mémoire.

3.3 Génération de définitions de type abstraits pour les constructeurs de type

3.3.1 Introduction

Le SPES-ifieur a à sa disposition des constructeurs de types unaires (pile de, file de, ens de, suite de)et n-aires (struct, structfonct) lui permettant de définir un type comme étant fonction de tels constructeurs et d'autres types.

Ainsi, la description d'expression "SORTE ØT == pile de /ØN7" définit le type ØT comme étant une pile d'entiers.

Outre la description d'expression, un type peut posséder une description stricte, définissant un type par un ensemble d'opérations, comme le montre l'exemple du point 3.4.2.1 de la première partie.

L'utilisateur moyen utilisera souvent le même ensemble d'opérations pour les objets dont la description d'expression du type utilise les constructeurs de type. Ainsi, les opérations couramment utilisées avec les objets dont le type utilise le constructeur "ens de"sont l'union et l'intersection de deux ensembles, le cardinal d'un ensemble, adjonction . ou la suppression d'un élément, le test d'appartenance, ...

L'utilisateur se verrait donc obligé de redéfinir souvent les mêmes opérations et les règles associées. Pour lui éviter ce travail inutile, non créatif et source d'erreurs, il semble intéressant d'automatiser la création de descriptions strictes pour les types utilisant les constructeurs de type. Les descriptions strictes créées de cette façon reprendraient les opérations de base permettant la définition du type et des opérations dont l'usage semble courant.

./...

N.B. - Il aurait été possible d'adopter une autre politique : l'introduction dans le langage des différentes opérations, en supposant leurs sémantiques comme étant établies.
Cette solution a deux désavantages :
- les opérations autorisées sur les constructeurs de type seraient ainsi figées;
- deux classes d'opérations seraient créées, les unes devant être présentes dans des descriptions strictes et possédant dès lors, profils et règles sémantiques, les autres devant être absentes de celles-ci.

L'introduction de ces opérations dans le langage aurait donc créé une discrimination arbitraire entre opérations.
Une telle approche n'a dès lors pas été retenue.

3.3.2 Spécification

La bibliothèque qui regroupe toutes les descriptions de type n'est accessible à l'utilisateur que par l'intermédiaire de primitives d'insertion ou de retrait d'une description dans la bibliothèque, d'affichage de description...

A chaque demande d'insertion d'une description d'expression ("SORTE ØT == expression type"), le processus de génération de description stricte est invoqué avec comme argument la description d'expression.

L'effet de cet appel est la création pour chaque constructeur de type présent dans l'argument d'une description stricte de ce constructeur.

Les descriptions strictes ainsi générées contiendront les opérations couramment invoquées sur les objets dont le type utilise ce constructeur.

./...

De plus, les descriptions initiales et générées seront insérées dans la bibliothèque.

Ainsi, lors de la réception de la description d'expression "SORTE ØA == pile de [ØN]", le processus va générer les règles sémantiques des opérations couramment référencées pour les objets de type "pile d'entier", à savoir l'ajout ou la suppression d'un élément au sommet de la pile, l'évaluation de l'élément au sommet ou de la hauteur de la pile, le test pour savoir si la pile est vide, et le test d'égalité entre deux piles.

3.3.3 Conception

On peut distinguer deux degrés d'utilisation des constructeurs de type dans les descriptions d'expression (qui constituent l'entrée du générateur) selon qu'il y ait dans celle-ci au plus un, ou plus d'un constructeur de type.

Exemples

SORTE ØT == ØW
SORTE ØT2 == struct [a:ØA,b:ØB, c:ØC]
SORTE ST3 == pile de [struct[a:ØA,b:pile de [ØB],c:struct [d:ØD,e:ØE]]]

Si plus d'un constructeur de type est présent dans une description d'expression, une transformation préalable est rendue nécessaire de par la grammaire des descriptions strictes. En effet, le profil d'une opération est composé d'identificateurs de type, et les constructeurs de type y sont interdits. Une difficulté surgit donc lorsqu'un constructeur a un autre constructeur parmi ses paramètres. On se ramènera dès lors à la première forme par une normalisation des descriptions.

La normalisation consiste à remplacer les constructeurs de type utilisés comme paramètres d'autres constructeurs par un nouveau type.

./...

Le troisième exemple sera normalisé sous la forme :
SORTE ØT3 == pile de /¯ØNOUV1_7
SORTE ØNOUV1 == struct/¯A:ØA,b:ØNOUV2,c:ØNOUV3_7
SORTE ØNOUV2 == pile de /¯ØB_7
SORTE ØNOUV3 == struct /¯d:ØD,e:ØE_7

C'est à partir de cette entrée transformée que pourront être créés les opérations et leurs règles sémantiques.

La création des opérations et règles peut être envisagée de deux façons, soit comme une construction algorithmique pure, soit comme l'instantiation d'une liste d'opérateurs et de règles données.

L'instantation fait appel à une arborescence pré-existante, sur laquelle des transformations simples sont effectuées afin d'obtenir la description stricte correspondant à l'argument donné. Les transformations dont il est question sont des substitutions de noms par d'autres noms (identificateur d'opération, identificateur de type). L'arborescence pré-existante est appelée méta-type. Le méta-type associé à un constructeur, est dès lors une description stricte, permettant, par substitution d'identificateur, la génération d'une description stricte pour toute description d'expression utilisant ce constructeur.

La construction algorithmique pure consiste en la construction d'une nouvelle arborescence par analyse de la description d'expression fournie. Dans ce cas, il n'y a donc pas substitution, mais bien création d'une description stricte.

Les avantages de la solution utilisant l'instantiation sont son faible temps d'exécution et la facilité de la maintenance, car il suffit de transformer un méta-type pour ajouter ou modifier des règles et/ou des opérations. Le désavantage de cette solution est la perte de place due à la présence des méta-types.

La création algorithmique a l'avantage qu'à priori la description d'entrée peut être arbitrairement compliquée, mais a comme désavantage un temps d'exécution plus long, et surtout une maintenance (pour l'ajout ou la suppression de règles) qui est réalisée par transformation d'algorithme.

C'est pourquoi, nous préférerons l'aternative d'instantiation pour les constructeurs le permettant, c.à.d. les constructeurs unaires comme nous le verrons par la suite.

La première étape consistera donc à trouver pour chaque constructeur un méta-type, et en cas d'échec déterminer les opérations et les règles sémantiques associées au constructeur. Ensuite, les différents algorithmes réalisant la génération des descriptions strictes seront présentés.

3.3.4 Recherche des méta-types

Une description d'expression contenant un constructeur est de la forme "SORTE $NOM==constr [paramètres]". Les constructeurs unaires (*pile de*, *file de*, *ens de*, *suite de*) ont un seul paramètre que nous supposerons (cfr. phase de normalisation) être un identificateur de type. Les constructeurs n-aires (*struct*, *structfonct*) ont un nombre quelconque de paramètres appelés champs, un champ étant composé d'un identificateur d'objet et d'un identificateur de type.

Un méta-type pour le constructeur constr revient à trouver une description stricte reprenant l'ensemble des règles et opérations couramment utilisés, cette description permettant d'obtenir par substitution de certains noms, la description stricte associée à chaque occurence de constr.

./...

3.3.4.1 Recherche du méta-type associé au constructeur "pile"

Soit à créer la description stricte correspondant à "SORTE ØSQ == pile de /_ØPARQ_7" . Les opérations courantes associées au constructeur "pile" sont : l'ajout ou la suppression d'un élement au sommet d'une pile, l'évaluation de la hauteur de la pile, l'évaluation de l'élément au sommet de la pile, le test permettant de savoir si la pile est vide et le test d'égalité entre deux piles.

- le méta-type "pile"

SORTE ØSQ

(la partie OP contient le nom des opérations, le profil de ses paramètres et du résultat)

```
OP  pv : ——>ØSQ   (pv crée une pile vide)
    emp : ØSQ,SPARQ——>ØSQ (adjonction d'un élément au sommet d'une
                           pile)
    depi : ØSQ ——>ØSQ  (suppression de l'élément sommet d'une pile
    som : ØSQ ——> ØPARQ (évaluation de l'élément au sommet)
    hau : ØSQ ——>ØN  (évaluation de la hauteur d'une pile)
    pvide : ØSQ ——>ØB (test pour savoir si la pile est vide)
    = : ØSQ,ØSQ ——> ØB (test d'égalité de deux piles)
    w : ——>ØPARQ (produit l'élément indéfini de type ØPARQ)
    (la zone DEC contient le typage des objets utilisés pour
    définir les règles).
DEC  p1,p2 : SSQ
     v1,v2 : ØPARQ
(la zone REG contient les définitions sémantiques des opéra-
                                                       tions )
REG  depi(pv()) ==pv();
     depi (emp(p1,v1))==p1;
     som (pv())==w();
     com (emp(p1,v1))==v1;
     hau (pv())==0;
     hau (emp(p1,v1))==hau(p1)+1;
     pvide (pv())==vrai;
     pvide (emp(p1,v1)==faux;
     p1 = p2 == vrai si pvide(p1) et pvide(p2);
     p1 = p2 == faux si (pvide(p1) et non (pvide(p2))) ou
                (non(pvide(p1) et pvide(p2));
     emp(p1,v1) = emp(p2,v2) == faux si v1 <> v2;
     emp (p1,v1) = emp (p2,v2) == p1 = p2 si v1 = v2
```

./...

Conclusions

Toutes les descriptions d'expressions utilisant le constructeur de type "pile" s'écriront, à une réappelation près des identificateurs de type : "SORTE ØT == pile de /ØQ/". Pour obtenir la description stricte correspondant à une telle description d'expression, il faudra et il suffira de remplacer chaque occurence de "SQ" par "ØT", et chaque occurence de "ØPARQ" par "ØQ". Dès lors, l'instantiation sera choisie pour la génération des descriptions strictes du contructeur "pile".

3.3.4.2 Recherche du méta-type associé au constructeur "file"

Soit à créer la description stricte correspondant à "SORTE ØSQ == file de /ØPARQ/". Les opérations courantes associées au constructeur "file" sont : l'ajout d'un élément dans une file, le retrait ou l'évaluation du premier élément de la file, l'évaluation de la longueur de la file, le test d'égalité de deux files, le test pour savoir si une file est vide.

- le méta-type "file"

SORTE ØSQ

OP fv : ⟶ ØSQ (fv crée une file vide)
menf : ØSQ,ØPARQ ⟶ ØSQ (mise d'un élément dans une file)
defi : ØSQ ⟶ ØSQ (retrait du premier élément de la file)
tête : ØSQ ⟶ ØPARQ (valeur du premier élément de la file)
long : ØSQ ⟶ ØN (longueur de la file)
fvide : ØSQ ⟶ ØB (teste si une pile est vide)
= : ØSQ,ØSQ ⟶ ØB (teste l'égalité de deux piles)
w : ⟶ ØPARQ (produit l'élément indéfini de type ØPARQ)

DEC f1,f2 : ØSQ
v1,v2 : ØPARQ

./...

```
REG  defi(fv()) ==fv();
     defi (menf(fv(),v1)) == fv();
     defi (menf(f1,v1) == (defi(f1),(v1);
     tête (fv()) == w();
     tête (menf(fv(),v1) == tête(f1);
     long (fv()) == 0;
     long (menf(f1,v1) == long(F1)+1;
     fvide (fv()) == vrai;
     fvide (menf(f1,v1) == faux:
     f1 = f2 == vrai si fvide(f1) et fvide(f2);
     f1 = f2 == faux si (fvide(f1) et non (fvide(f2)))ou(fvide(f2) et
                non(fvide(f1)));
     menf(f1,v1) = menf(f2,v2) == faux si v1   v2;
     menf(f1,v1) = menf(f2,v2) == f1 = f2 si v1 = v2
```

Conclusions

Toutes les descriptions d'expressions utilisant le constructeur de type "file" s'écriront, à une réappelation près des identificateurs de type "SORTE ØT == file de /ØQ/. Pour obtenir la description stricte correspondant à une telle description d'expression, il faudra et il suffira de remplacer chaque occurence de "SQ" par "ØT", et chaque occurence de "ØPARQ" par "ØQ". Dès lors, l'instantiation sera choisie pour la génération des descriptions strictes du constructeur "file"

3.3.4.3 Recherche du méta-type associé du constructeur "ensemble"

Soit à créer la description stricte correspondant à "SORTE ØSQ == ens de /ØPARQ/". Les opérations courantes associées au constructeur "ensemble" sont l'ajout ou le retrait d'un élément à un ensemble, le test d'appartenance d'un élément à un ensemble, l'évaluation du cardinal d'un ensemble, le test d'inclusion d'un ensemble dans un autre, l'union, la différence et l'intersection de deux ensembles, le test d'égalité de deux ensembles.

- le méta-type "ensemble"

  SORTE ØSQ

```
OP  ev : ——> ØSQ (fournit l'ensemble vide)
    ajout : ØSQ,ØPARQ ——> ØSQ (ajout d'un élément à un ensemble)
    app : ØSQ,ØPARQ ——> ØB (test d'appartenance)
    retrait : ØSQ,ØPARQ ——> ØSQ(retirer un élément d'un ensemble)
    cardi : ØSQ ——> ØN (évalue le cardinal d'un ensemble)
    inclus : ØSQ,ØSQ ——> ØB (texte d'inclusion d'un ensemble dans un
                                                              autre)
    union : ØSQ,ØSQ ——> ØSQ(union de deux ensembles)
    diff : ØSQ,ØSQ ——> ØSQ (différence de deux ensembles)
    inter : ØSQ,ØSQ ——> ØSQ (intersection de deux ensembles)
    = : ØSQ,ØSQ ——> ØB (teste l'égalité de deux ensembles)
    w : ——> ØPARQ (produit l'élément indéfini de type ØPARQ)
DEC e1 , e2 : ØSQ
    t1 , t2 : ØPARQ
REG retrait (ev(),t1) == ev();
    retrait (ajout(e1,t1),t2) == e1  si  t1 = t2;
    retrait (ajout(e1,t1),t2) == ajout(retrait(e1,t2),t1)si t1<>t2;
    cardi (ev()) == 0;
    cardi (ajout(e1,t1)) == cardi(e1)+1;
    inclus (ev(),e1) == vrai;
    inclus (ajout(e1,t1),e2) == inclus (e1,e2) si app(e2,t1) = vrai;
    inclus (ajout(e1,t1),e2) == faux si app(e2,t1) = faux;
    app (ev(),t1) == faux;
    app (ajout(e1,t1),t2) == vrai si t1 = t2;
    app (ajout(e1,t1),t2) == app(e1,t2) si t1<> t2;
    union (e1,ev()) == e1;
    union (e1,ajout(e2,t1)) == union (ajout(e1,t1),e2) si app(e1, 1)=
                                                              faux;
    union (e1,ajout(e2,t1)) == union(e1,e2) si app (e1,t1) = vrai;
    diff (e1,ev()) ==e1;
    diff (e1,ajout(e2,t1))== diff(retrait(e1,t1),e2) si app(e1,t1) =
                                                              vrai;
    diff (e1,ajout(e2,t1))== diff(e1,e2) si app(e1,e2) = faux;
    inter (e1,e2) == diff(e1. diff(e1,e2));
    e1 = e2 == inclus(e1,e2) et inclus(e2,e1)
```

Conclusions

Toutes les descriptions d'expressions utilisant le constructeur "ensemble" s'écriront, à une réappellation près des identificateurs de type : "SORTE ØT == ens de /ØQ/". Pour obtenir la description stricte correspondant à une telle description, il faudra e il suffira de remplacer chaque occurence de "ØSQ" par"ØT", et chaque occurence de "ØPARQ" par "ØQ". Dès lors, l'instantiation sera choisie pour la génération des descriptions strictes du constructeur"ensemble"

3.3.4.4 Recherche du méta-type associé au constructeur "suite"

Soit à créer la description stricte correspondant à "SORTE ∅SQ == suite de /∅PARQ/". Les opérations courantes associées au constructeur "suite" sont l'ajout d'un élément à une suite, l'évaluation du ième ou du dernier élément d'une suite, le retrait du dernier élément de la suite, l'évaluation de la longueur de la suite et le test d'égalité de deux suites.

- le méta-type "suite"

```
SORTE ∅SQ
OP sv : ──> ∅SQ  (création d'une suite vide)
   missui : ∅SQ,∅PARQ ──> ∅SQ (ajout d'un élément en fin de suite)
   retsui : ∅SQ ──> ∅SQ (retrait du dernier élément de la suite)
   derel : ∅SQ ──> ∅PARQ (évaluation du dernier élément de la suite
   longsui : ∅SQ ──> ∅N (évaluation de la longueur de la suite)
   acces : ∅SQ,∅N ──>∅PARQ (évaluation du ième élément d'une suite
   w : ──> ∅PARQ (produit l'élément indéfini de type ∅PARQ)
   = : ∅SQ,∅SQ ──>∅B
DEC s1,s2 : ∅SQ
    i : ∅N
    e1,e2 : ∅PARQ
REG  retsui (sv()) == sv();
     retsui (missui(s1,e1)) == s1;
     derel (sv()) == w();
     derel (missui(s1,e1)) ==e1;
     longsui (sv()) == 0;
     longsui (missui(s1,e1)) == longsui(s1)+1;
     acces (s1,i) == w() si i> longsui(s1);
     acces (s1,i) == derel(s1) si  i=longsui(s1);
     acces (missui(s1,e1),i) == acces(s1,i) si  i<longsui(s1);
     s1 = s2 == vrai si longsui(s1) = 0 et longsui(s2) = 0;
     s1 = s2 == faux si (longsui(s1)=0 et non (longsui(s2)=0)) ou
                        (longsui(s2)=0 et non (longsui(s1)=0));
     missui(s1,e1) = missui(s2,e2) == faux si e1<>e2;
     missui(s1,e1 = missui(s2,e2) == s1 = s2 si  e1 = e2;
```

Conclusions

Toutes les constructions d'expressions utilisant le constructeur "suite" s'écriront, à une réappelation près des

./...

identificateurs de type "SORTE ØT == suite de /ØQ/". Pour obtenir la description stricte correspondant à une telle description, il faudra et il suffira de remplacer chaque occurence de "ØSQ" par "ØT", et chaque occurence de "ØPARQ" par "ØQ". Dès lors, l'instantiation sera choisie pour la génération des descriptions strictes du constructeur "suite".

3.3.4.5 Recherche du méta-type associé au constructeur "struct"

Les constructeurs étudiés ci-dessus étaient paramétrés par deux identificateurs de type (ØSQ et ØPARQ); par contraste le constructeur "struct" est paramétré par un identificateur de type et par une liste de couples (objet, identificateur de type).
Soit à créer la description stricte correspondant à "SORTE ØSQ == struct/ob1 : ØT1,ob2 : ST2, ...,obn : ØTN/"On aimerait disposer de fonctions permettant l'accès aux différents champs d'une structure composite, p.ex. sur base du nom de l'objet figurant dans le couple (objet, identificateur de type); chacune de ces fonctions prendrait alors le nom de l'objet associé. Les autres opérations dont on aimeraït disposer sont le test d'égalité de deux structures et la formation d'une structure à partir de champs.
- non-existence du méta-type "struct"

Le fait que "struct" soit paramétré par des couples dont l'un des éléments est le nom d'une fonction à générer, ne gêne évidemment pas la définition d'un méta-type"struct". Le problème provient du nombre aléatoire de couples et donc de fonctions à produire. On pourrait envisager de construire n méta-types,où n peut être arbitrairement grand, et où le ième méta-type correspondrait à une liste de i couples comme paramètres de la description considérée. Cette solution étant loin d'être satisfaisante, nous n'utiliserons pas le procédé d'instantiation dans le cas du

./...

constructeur "struct"; dès lors nous générerons les descriptions strictes de descriptions d'expression par un procédé algorithmique.

Exemple de description stricte à générer par l'automate

Soit la description d'expression suivante :

SORTE ∅BON-DE-COMM == struct /ident-cli:∅ID-CLI, date:∅DATE,
lignes : ∅LIGNES, montant : ∅MONTANT/

La description stricte à générer sera la suivante :

SORTE ∅BON-DE-COMM

OP : ident-cli : ∅BON-DE-COMM ⟶ ∅ID-CLI

(accès au champ ident-cli d'un objet de type ∅BON-DE-COMM)

date : ∅BON-DE-COMM ⟶ ∅DATE

lignes : ∅BON-DE-COMM ⟶ ∅LIGNES

montant : ∅BON-DE-COMM ⟶ ∅MONTANT

= : ∅BON-DE-COMM, ∅BON-DE-COMM ⟶ ∅B

gr : ∅ID-CLI,∅DATE,∅LIGNES,∅MONTANT ⟶ ∅BON-DE-COMM

(le regroupement des différents champs forme un objet de type ∅BON-DE-COMM)

DEC b1,b2 : ∅BON-DE-COMM

REG gr(ident-cli(b1),date(b1),lignes(b1),montant(b1))==b1;

b1 = b2 == (ident-cli(b1) = ident-cli(b2)) et
(date(b1) = date(b2)) et
(lignes(b1) = lignes(b2)) et
(montant(b1) = montant(b2))

3.3.4.6 Recherche du méta-type associé au constructeur "structfonct".

Le constructeur structfonct est également paramétré par un identificateur de type et par une liste de couples (objet, identificateur de type). Les fonctions dont on aimerait disposer sont identiques à celles du constructeur "struct", en ajoutant dans les règles que si deux objets ont une même valeur pour leurs premiers champs, alors ces objets sont identiques.

- non-existence du méta-type "struct fonct"

  Le raisonnement appliqué au constructeur "struct" est utilisable ici, et dès lors, les descriptions strictes seront générées par un procédé algorithmique.

Exemple de description à générer par l'automate

Soit la description d'expression suivante :

```
SORTE $BON-DE-COMM==struct fonct [ident-cli:$ID-CLI,
      date:$DATE,lignes:$LIGNES,montant:$MONTANT].
```

La partie "règles" de la description est la suivante :

```
REG gr(ident-cli(b1),date(b1),lignes(b1),montant(b1))==b1 ;
    date(b1)==date(b2) si ident-cli(b1)=ident-cli(b2) ;
    lignes(b1)==lignes(b2)si ident-cli(b1)=ident-cli(b2) ;
    montant(b1)==montant(b2) si ident-cli(b1)=ident-cli(b2) ;
    b1=b2==ident-cli(b1)=ident-cli(b2)
```

3.3.5 Réalisation

L'entrée du processus global est une description d'expression quelconque. Le premier traitement à réaliser sur la description est sa normalisation, qui éventuellement modifiera la description d'expression, et en créera de nouvelles. Aux résultats de ce premier processus sera appliquée ensuite la phase de génération des descriptions strictes pour toutes les descriptions d'expressions contenant un constructeur de type.

3.3.5.1 La normalisation des descriptions d'expressions.

Le but de la normalisation est de fournir des descriptions d'expressions contenant au plus un constructeur de type.

Pour réaliser cet objectif, l'algorithme de transformation suivant est utilisé : lorsqu'un paramètre d'un constructeur de type est lui-même un constructeur de type, il y a substitution de ce dernier par un identificateur de type, et création d'une nouvelle description d'expression correspondante,

à laquelle cet algorithme sera à nouveau appliqué ; le point de terminaison est atteint lorsqu'aucun paramètre d'un constructeur n'est un constructeur de type.

Le premier argument de cette procédure est une liste de descriptions à analyser, et le second qui servira d'entrée au processus suivant est une file intermédiaire dans laquelle toutes les descriptions sont normalisées, et sont celles créées par transformations successives des descriptions initiales.

Exemple

La description d'expression "SORTE $S1==pile de[ens de[struct [a:$A,b:pile de [$B],c:$C]]]" sera transformée par le passage dans ce processus, en la liste de descriptions d'expressions normalisées suivante :

SORTE $S1==pile de [$T1]
SORTE $T1==ens de [$T2]
SORTE $T2==struct [a:$A,b:$T3,c:$C]
SORTE $T3==pile de [$B]

Algorithme de normalisation

```
*Normalisation (a,b)
 tant que a est non vide
   faire analysé ← tête de a ;
         ôter la tête de a;
         si analyse contient un constructeur
            alors s'il s'agit d'un constructeur unaire
                  alors analyse (paramètre du constructeur,resu,a)
                        paramètre du constructeur   resu;
                  sinon pour chaque champs du constructeur
                               faire analyse (2ième élément du couple,
                                          resu a);
                                     2ième élément du couple← resu ;
                               fin-faire;
                  fin-si;
            fin-si;
         b←b+analysé;
   fin-faire;.
```

./...

analyse (paramètre,res,a)

(le premier argument est un identificateur de type ou un constructeur. Dans le premier cas, res reçoit comme valeur cet identificateur, et a est inchangé. Si paramètre est un constructeur, alors res reçoit comme valeur un identificateur de type différent de tous les autres, et a est augmenté de la description d'expression de res en fonction de paramètre)

```
si paramètre est un identificateur de type
   alors res ← paramètre;
   sinon res ← nouvel identificateur;
         a ← a+description de res par paramètre;
   fin-si;.
```

3.3.5.2 La génération des descriptions d'expressions

Algorithme

*génération(b)

```
 pour chaque description présente dans b
      faire si la description contient un constructeur
                 alors si le constructeur est unaire
                            alors -récupération de l'arborescence du
                                   méta-type correspondant;
                                  -remplacer "$SQ" par le type décrit;
                                  -remplacer "$PARQ" par le paramètre
                                   du constructeur;
                            sinon construction algorithmique;
                            fin-si;
                            archivage de la description stricte;
            fin-si;
            archivage de la description d'expression;
      fin-faire;.
```

*construction algorithmique

(3 étapes : construction de la partie "profils", des déclarations et enfin des règles.

Le nom des fonctions est le nom de l'objet du champs, son domaine le type décrit, et son codomaine le second élément du champs.

Les objets déclarés sont choisis de façon à avoir des identificateurs différents des fonctions à définir.)

3.4 Génération de descriptions de type à partir d'énoncés

3.4.1 Introduction

Le typage des objets SPES peut se faire soit par la mention d'un identificateur de type, soit par l'utilisation de constructeurs et d'identificateurs de types. Dans le cas où l'identificateur de type n'est pas prédéfini (ØN, ØR, ØC, ØB), l'utilisateur doit en donner la signification par une description ajoutée dans la bibliothèque. Ainsi, si la définition de type "r:ØQ" est présente dans un énoncé, la bibliothèque devra contenir une description du type ØQ.

Le but de cette réalisation est la détermination de la description d'expression (c.à.d. en fonction de constructeurs et d'identificateurs de types) de chaque identificateur de type non prédéfini présent dans le dossier, et celà à partir des manipulations réalisées sur les objets de ce type dans l'ensemble du dossier.

Le processus va soumettre à l'utilisateur 0, 1 ou plusieurs définitions possibles des types rencontrés et, les définitions pouvant être incompatibles, c'est alors à l'utilisateur de choisir l'alternative lui semblant adéquate ou d'en prescrire une nouvelle.

Exemple 1 (un exemple plus consistant est donné par la suite).
Soit les fragments d'énoncés :
"r : ØW
 s = r/i/ et cond"
Le processus proposera comme définition de ØW :
"SORTE ØW == suite de /ØB/" .

Une solution envisagée par l'automate reste donc une proposition, car il est impossible de générer pour tout type une

solution absolument sûre. En effet, les algorithmes conduisant à la définition d'un type sont basés sur des analyses en fonction des opérations effectuées sur les objets de ce type. Il n'y a unicité de solution que si toutes les opérations sont définies et possèdent un et un seul domaine (ou profil des opérandes) et un et un seul co-domaine (ou profil du résultat). La plupart des langages, dont SPES, autorisant la surcharge des opérations (c.à.d. la possibilité pour une opération d'avoir différents domaines et co-domaines), et SPES en particulier autorisant la redéfinition par l'utilisateur de chaque opération, toute solution proposée restera une proposition basée sur un modèle décrivant les opérations de bases associées aux constructeurs de type. C'est ainsi que l'on supposera par exemple que l'utilisateur référençant la fonction "menf" songe à la fonction définie dans le méta-type "file".

Cette réalisation se base sur l'analyse à posteriori du dossier, et doit être demandée par l'utilisateur. Elle n'est que le premier pas vers la détermination du type d'un objet en fonction des opérations effectuées sur cet objet; une telle détermination ne s'effectuant plus à la demande, mais soumettant automatiquement des propositions à l'utilisateur en fonction du texte déjà introduit, et éventuellement d'une bibliothèque contenant les constructions fréquemment utilisées dans le domaine d'application (gestion, mathématique, statistiques, ...)

3.4.2 Spécification

A partir d'une liste d'énoncés archivés, le générateur de descriptions d'expression doit essayer de trouver pour chaque identificateur de type présent dans ces énoncés et non-standard (les identificateurs standards étant ØN, ØB, ØR et SC), une définition exprimant le type comme étant fonction de constructeurs et d'autres identificateurs de type. Ces derniers seront soit des identificateurs standards, soit des identificateurs de type dont

./...

une description a déjà été trouvée, soit des identificateurs créés.

Cette dernière possibilité permettra, par exemple, d'informer l'utilisateur que le type analysé représente une pile sans que l'on ait su inférer le paramètre du contructeur pile (c.à.d. sans que l'on connaisse le type des éléments de la pile).

Le processus de détermination d'une description d'expression d'un type se basera sur les manipulateurs effectuées dans le dossier sur les objets de ce type, manipulations permettant d'inférer la structure d'un objet, c.à.d. le construteur associé à son type et le ou les paramètres de ce constructeur.

Les descriptions générées seront soumises à l'utilisateur afin qu'il en sélectionne une, ou en prescrive une nouvelle.

3.4.3 Conception

Le processus de détermination de la structure d'un objet est invoqué par le générateur pour chaque objet du dossier dont le type n'est pas prédéfini, et doit créer par analyse des utilisations de l'objet, toutes les structures possibles de celui-ci.

Deux sous-problèmes peuvent être distingués : la détermination du constructeur et la détermination du ou des paramètres de celui-ci.

Les paramètres d'un constructeur sont pour les unaires (pile de,...) un identificateur de type, et pour les n-aires (struct,...) une liste de couples formés d'un identificateur d'objet et d'un identificateur de type.

Afin de pouvoir déterminer la partie identificateur de type des paramètres d'un constructeur, nous utiliserons un processus non encore décrit, évaluant le type du résultat d'une expression quelconque dans un contexte donné. Nous supposerons

./...

ici que ce processus d'évaluation de type indique soit qu'il ne sait pas évaluer le type de l'expression, soit qu'il a su l'évaluer, auquel cas il renvoie un identificateur de type. Ce processus sera expliqué plus précisément dans le chapitre 4.

Nous distinguons deux méthodes permettant la détermination du constructeur d'une structure, selon qu'il s'agisse d'un constructeur unaire ou d'un n-aire. Les premiers possèdent en effet une bibliothèque d'opérations de base présentes dans les méta-types; l'utilisation d'une de ces opérations permettra d'inférer le constructeur associé au type de l'objet et éventuellement ses paramètres. Les constructeurs n-aires ne disposent pas de méta-types, et dès lors la détection d'objet dont le type utilise ces constructeurs se fera par une recherche basée sur la forme des définitions formelles des objets.

3.4.3.1. Les constructeurs unaires

Comme nous l'avons vu, les constructeurs unaires ont la particularité de tous posséder un méta-type, et donc une liste d'opérations pré-existantes associées à chaque constructeur; l'unicité des noms de fonctions présentes dans les méta-types permet d'identifier un constructeur de type à partir d'un nom de fonction. Ainsi par exemple, l'identificateur de fonction "retrait" permet d'identifier le méta-type "ensemble". L'occurence d'un objet comme argument d'une telle fonction permet donc d'inférer la structure de cet objet au sens défini ci-dessus.

Le but étant de retrouver tant le constructeur que ses paramètres, il convient de remarquer que toutes les fonctions n'y contribuent pas de la même manière. Ainsi, la fonction depi(a) suggère que l'objet a est une pile, sans que l'on puisse inférer le type du paramètre pile. Par contre, la fonction retrait (b,c) suggère que l'objet b est un ensemble d'objets dont le type est celui de l'objet c.

Dans la suite, nous appelerons fonctions atypées toute fonction présente dans les méta-types qui ne permet que d'inférer le constructeur, et fonction typée toute fonction qui permet d'inférer constructeur et paramètre. Parmi ces dernières, le paramètre peut être inféré soit d'un argument de la fonction (ex : retrait(b,c)), soit du résultat de celle-ci (tête(d) suggère que d est une pile, dont le type des éléments est le type du résultat de la fonction). Nous parlerons dès lors de fonction para-typée dans le premier cas, et de fonction résu-typée dans le second.

Pour rappel, les méta-types ont été construits sur base d'une description d'expression de la forme "SORTE ØSQ == contr [ØPARQ]". La règle permettant de classifier une fonction dans l'une de ces trois catégories est la suivante : si aucune occurence de ØPARQ n'intervient dans le profil de la fonction, alors celle-ci est atypée. Dans le cas contraire, si le type d'un argument est ØPARQ, alors il s'agit d'une fonction para-typée, sinon d'une résu-typée.

- Une fonction atypée ne permet donc que l'identification du constructeur. La sortie du processus, dans ce cas, est le constructeur auquel on ajoute un paramètre créé, différent de tous ceux employés dans le dossier et dans les descriptions précédemment inferées, et n'ayant aucune signification.

- Pour les fonctions para-typées, le constructeur est également retrouvé de par l'unicité des noms de fonctions. La fonction étant paratypée, il existe une occurence en ième position de ØPARQ dans le profil de cette fonction. Le ième argument voit dès lors son type évaluer. Si l'évaluation réussit, le paramètre du constructeur est le résultat de cette évaluation; sinon l'on applique le principe expliqué pour les fonctions atypées.

./...

- Les fonctions résu-typées subissent un traitement totalement différent, car le processus d'évaluation de type ne permet que l'analyse d'expression dont le profil est connu, ce qui n'est pas le cas ici car l'on cherche à déterminer le type du résultat. Le traitement sera axé sur l'information que l'on peut tirer de la position dans l'énoncé de la référence de fonction.

Signalons au préalable une fonction résu-typée particulière à la notation s[i] représentant la fonction acces (s,i). Lorsque ce raccourci d'écriture est rencontré, on sait que s est défini au moyen du constructeur suite dont le paramètre est le type du résultat de s[i].

Soit f une fonction résu-typée. Pour déterminer le type de son résultat, nous procéderons de la façon suivante :

- si f est utilisé comme argument de certaines fonctions SPES non redéfinissables (<u>non</u>, <u>et</u>, <u>ou</u>), il est possible d'évaluer le type de son résultat en extrapolant que celui-ci est le type de l'argument de la fonction standard. Ainsi, <u>non</u>(f), f <u>et</u> ... , f <u>ou</u> ... permettent d'inférer que f est à résultat de type booléen.

- si f est utilisé comme argument dans une référence à un énoncé, le type du résultat de f est le type de l'argument "formel" lui correspondant dans cet énoncé. Ainsi, si l'on trouve r = A(f,j), le type du résultat de f est le type du premier paramètre "formel" de A.

- si f est la seule expression d'un membre d'une égalité, le type de f est le type obtenu par le processus d'évaluation de type de l'expression figurant dans l'autre membre de l'égalité.

- si f est la seule expression d'une condition, alors le type de son résultat est ØB.

./...

- si la position de f ne rentre dans aucun des cas précités, alors le principe des fonctions atypées est appliqué.

3.4.3.2 Les constructeurs n-aires

Les constructeurs n-aires (struct et structfonct) ne possèdent pas de méta-types. Leur particularité commune réside dans le fait que pour chaque objet paramètre du constructeur est définie une fonction d'accès portant le nom de cet objet. De plus, la définition d'un objet structuré se fait par définition de ses différents champs, accessibles donc uniquement via les fonctions d'accès.

Exemple

Soit la description "SORTE ØS == struct/_a:ØA,b:ØB,c:ØC_7";
la définition d'un objet r de type ØS doit obligatoirement avoir la forme "r tq a(r) = a' et b(r) = b' et c(r) = c' ", où a',b' et c' sont des expressions quelconques dont les types respectifs sont ØA, ØB et ØC.

Lors de l'analyse d'une liste d'énoncés, on déduira qu'un objet r est d'un type défini à l'aide du constructeur struct si l'on trouve une définition implicite de r formée de références de fonctions figurant dans des égalités à des expressions quelconques.

Les références de fonction devront répondre aux critères suivants :

- avoir comme seul argument l'identificateur de l'objet r;

- ne pas être une fonction SPES ou une fonction présente dans un méta-type.

Si l'on trouve une définition implicite répondant

à ce critère, on en déduit que le constructeur de type est "struct". Il faut remarquer que l'on ne peut à la lecture des énoncés préciser s'il s'agit du constructeur structfonct; c'est pourquoi le constructeur général struct sera toujours choisi.

Le paramètre d'un constructeur n-aire est une liste de couples formés d'un objet et d'un identificateur de type. La reconstitution de chacun des couples est réalisée comme suit : l'objet a comme identificateur l'identificateur de fonction, et l'identificateur de type est soit celui trouvé par le processus d'évaluation du type d'une expression appliqué au second membre de l'égalité, soit un identificateur sans signification si l'évaluation du type est infructueuse.

3.4.3.3. Exemple

Soit la liste de fragments d'énoncés suivante :

```
r,s A
r:ØT
r tq a(r)=ancien et date(r) = date-du-jour
date-du-jour : ØDATE
ancien : ØNOM
s : ØM
s = der[10]
der : ØW
der = rep i dans 1..10 der[i] = der[i-1] et B(tête(p)) init
            der[1] = vrai
p : ØTRUC
...
r B(p)
r : ØB
P : ØB
```

Les descriptions proposées sont les suivantes :

```
SORTE ØT == struct[a : ØNOM,date : ØDATE]
SORTE ØW == suite de[ØB]
SORTE ØTRUC == file de[ØB]
```

./...

3.4.4 Réalisation du processus de génération de description de type *def-type (dossier)

```
(*dossier est la liste d'énoncés à analyser)
définition-proposée ← vide (*contient toutes les définitions
                            déjà générées*)
pour chaque énoncé eno de dossier
     faire pour chaque définition de type defi-ty de eno
           faire pour chaque objet ob de defi-ty
                      faire générer (ob,type de defi-ty,eno,défini-
                                                  tion-proposée);
                      fin-faire;
           fin-faire;
     fin-faire;
afficher définition-proposée pour sélection.

* générer (ob,type,eno,def-prop)
si la définition formelle de f de ob dans eno est implicite
   alors paramètre ← vide;
         pour chaque fonction f de def dont ob est le seul paramètre
              faire si la fonction est un membre d'une égalité
                       alors - créer un champ c dont le nom de l'obje
                               est identificateur de fonction;
                             - évaluer-type (autre membre égalité,
                                                         résu);
                             - si évaluation non concluante
                                  alors resu ← identificateur
                                                          bidon;
                                  fin-si;
                             - identificateur de type de c ← résu;
                             - paramètre ← paramètre + c;
                       fin-si;
              fin-faire;
              - créer une description de type struct avec paramètre;
              - ajout (struct, def-prop);
   fin-si;
```

./...

```
pour chaque fonction f ayant ob comme paramètre
    faire type-fonct( f,type,constr,profil)
          si type = atypée
             alors résu ←— identificateur de type  bidon;
                   ajout (constr avec resu,def-prop);
          ou si type = para-typée
             alors pour chaque occurende de"ØPARQ" dans profil
                      faire évaluer-type(paramètre correspon-
                                         dant, résu);
                            si évaluation non-concluante
                               alors resu ←— identificateur
                                                   "bidon";
                            fin-si;
                            ajout (constr avec resu,def-prop);
                      fin-faire;
             ou si type = résu-typée
                alors recherche-position (f,resu);
                      si recherche non concluante
                         alors resu ←—identificateur "bidon";
                      fin-si;
                      ajout(constr avec resu, def-prop);
             fin-si ;
    fin-faire;
```

* type fonct (fonct,type,constr,profil)

(détermine si une fonct°est para, resu ou atypée, et le constructeur contenant une fonction de ce nom).

```
si fonct = objet indexé
   alors type ←— resu-typée;
         constr ←— suite de
   sinon si identificateur de fonction de fonct est présente dans un
                                                         méta-type
         alors profil ← profil de la fonct°;
               constr ← constructeur associé au méta-type;
               si profil ne contient pas d'occurence de ØPARQ
                  alors type ← atypée;
                  sinon si le domaine contient une occurence de ØPARQ
                           alors type ← para-typée;
                           sinon type ← resu-typée;
                        fin-si
               fin-si;
         sinon type ← inconnu;
    fin-si;
fin-si;
```

* ajout (constr,def-prop);

(Ajoute conditionnellement une proposition à l'ensemble existant. L'adjonction est réalisée s'il n'existe pas encore de définition identique déjà archivée. S'il y a déjà une définition du type archivée, avec le même constructeur, et si l'une contient un identificateur de type créé, l'autre est choisie et remplace la première).

* recherche position (f,resu)

(Essaie de déterminer le type du résultat de f selon sa position dans l'arborescence, en appliquant les règles définies pour les fonctions résu-typées).

4. OUTILS D'ANALYSE D'UNE SPECIFICATION

Du fait de l'utilisation d'un éditeur syntaxique, le texte introduit via l'éditeur est à tout moment en concordance avec la grammaire du langage. Il existe cependant un ensemble de propriétés qui ne peuvent être formalisées dans la grammaire, et donc que l'éditeur syntaxique ne peut analyser.

Il est donc souhaitable d'enrichir l'environnement de spécification de procédures vérifiant que des règles non grammaticales mais sémantiques sont vérifiées au sein d'une arborescence syntaxiquement correcte.

Les règles à vérifier sont des contraintes qu'on écrit habituellement en langage naturel lors de la description d'un langage.

Le présent chapitre est divisé en quatre parties afin de regrouper les vérifications effectuées au sein d'un énoncé, du dossier, de la bibliothèque et enfin de la spécification.

4.1 Vérifications effectuées au sein d'un énoncé

Au départ d'un énoncé, et en disposant de son environnement (bibliothèque et dossier), on peut vérifier certaines règles qui doivent être satisfaites dans le corps de cet énoncé. Ces vérifications peuvent être effectuées soit à la demande expresse de l'utilisateur qui désire savoir ce qu'il doit modifier ou ajouter dans l'énoncé courant, soit systématiquement lors de l'archivage, archivage qui de plus entraîne l'exécution du processus de gestion de pseudo-commentaires.

Les vérifications suivantes sont envisagées au sein d'un énoncé :

./...

- chaque objet dont la portée est l'énoncé doit posséder dans celui-ci une et une seule définition de chaque sorte (définition de type, formelle et informelle).
- chaque objet possédant une définition doit contribuer à l'élaboration du ou des résultats de cet énoncé. On peut ainsi détecter les objets définis mais non utilisés.
- On ne peut avoir de définitions formelles d'objets qui soient cycliques ; un objet est considéré comme ayant une définition cyclique si les objets qui le définissent (ses descendants) font référence à lui ou aux objets qu'il contribue à définir (ses ascendants) dans leurs propres définitions.
- Il peut être utile de signaler les types référenciés dans l'énoncé et dont la définition est inconnue.
- On peut également vérifier les concordances de types au sein des définitions formelles ; ainsi on peut vérifier que l'objet à définir et sa définition sont de même type, ou que le type d'une condition est bien booléen.

L'objet des sections suivantes est la présentation de la manière dont ont été envisagées ces vérifications.

4.1.1 Existence et unicité des définitions d'objets et détection des objets non utilisés

Rappels

- Dans un énoncé, chaque objet possède un identificateur, et la relation liant objet et identificateur est univoque dans une définition.
- Les objets ont soit une portée globale à l'énoncé, soit une portée locale. Les seuls objets ayant une portée locale sont
  - l'indice présent dans les définitions de suite de la forme "pour"
  - les objets liés aux quantificateurs universels (qq) ou existentiels (ex).

  La portée de ces objets est étendue dans le premier cas à la définition de suite, et, dans le second, à la formule associée au quantificateur.

- Les objets ayant une portée globale doivent posséder trois définitions (une définition formelle, une définition informelle, et une définition de type), sauf les arguments d'un énoncé qui ne peuvent avoir de définition formelle dans l'énoncé analysé.

Objectifs

Le premier objectif est de détecter tous les objets n'ayant pas une et une seule définition de chaque espèce et vérifier que les objets locaux ont des identificateurs distincts des objets globaux.

Le second objectif est de détecter tous les objets inutiles. On dira qu'un objet non résultat est inutile s'il ne contribue pas à l'élaboration d'un résultat, c-à-d si cet objet n'est ni un résultat, ni un objet intermédiaire, un objet intermédiaire étant présent dans la définition formelle du résultat ou d'un objet intermédiaire.

Ces deux vérifications sont présentées simultanément car il s'agit de contrôles que l'utilisateur demande fréquemment lors de l'écriture d'un énoncé, car ils lui indiquent les définitions qu'il doit encore introduire dans l'énoncé courant.

Ces vérifications sont à mettre en rapport avec celles effectuées dans les analyseurs statiques de programme (compilateurs de haut niveau) qui détectent des variables présentes dans une zone de déclaration sans jamais être utilisées dans le bloc de traitement correspondant, des variables utilisées sans être présentes dans les zones de déclarations englobantes, ou encore les variables auxquelles une valeur est affectée, sans que cette valeur ne soit jamais utilisée.

4.1.1.1 Exemple

L'énoncé à analyser est le suivant (on n'attachera aucune attention à sa pertinence) :

./...

```
res,res2 ENO (par,par2)
res,par:$M
par2:$N
res2,par,par2 ? res2 est la somme de par et de par2
res2=par+par2
res2=dep i dans o.. 10 rep res2=w
double-v=par+par2
double-v:$M
double-v ? somme des deux paramètres
par=5+f(15,10)
```

Les erreurs suivantes seront détectées et signalées :

pas de définition de type de:w,res2
pas de définition informelle de:res,w
pas de définition formelle de:res,w
plusieurs définitions formelles de:res2,par
(par est un argument de l'énoncé et ne peut donc y avoir de définition formelle)
objet non utilisé : double-v

4.1.1.2 Spécification

Il s'agit de construire un algorithme qui, par analyse de l'arborescence d'un énoncé reçue comme argument, devra signaler les objets ne vérifiant pas la règle d'existence et d'unicité des définitions, ou ne contribuant pas aux résultats ou enfin source d'ambiguïté.

Pour un énoncé donné, il faudra dès lors :

- signaler les objets globaux n'ayant pas une et une seule définition formelle, une et une seule définition informelle ou une et une seule définition de type. On prendra la convention que la définition d'un objet comme argument formel d'un énoncé constitue la définition formelle de cet objet.

./...

- signaler les objets inutiles. Un objet inutile est un objet non résultat ne pouvant être trouvé par application de la règle suivante : - définir les objets intermédiaires comme étant ceux utilisés dans la définition formelle du ou des résultats

  - considérer ces objets intermédiaires comme de nouveaux résultats, et appliquer récursivement cette démarche

  - s'arrêter lorsque ces objets intermédiaires sont en provenance de l'extérieur de l'énoncé.
- signaler les objets locaux ayant un identificateur ambigu, soit par rapport à un autre objet local également utilisable dans la définition, soit par rapport à un objet global.

4.1.1.3 Concepts

La réalisation est basée sur un parcours unique de l'arborescence de l'énoncé en mémorisant les objets rencontrés dans des files en fonction de l'information que l'on peut tirer de la position de l'objet dans l'arbre syntaxique et de sa présence dans d'autres files.

On disposera de files indiquant les objets possédant plus d'une définition formelle, informelle ou de type. Les objets n'ayant pas une des trois définitions seront les objets globaux que l'on a rencontrés dans l'énoncé ne se trouvant pas dans des files indiquant qu'ils possèdent au moins une définition de l'espèce considérée. Les objets non utilisés sont ceux rencontrés sans que l'on ait pu les trouver par application de la règle précitée. Enfin, les objets locaux dont les identificateurs sont ambigus seront ceux rencontrés à la fois comme objets locaux et globaux.

On utilisera dès lors les files suivantes afin de remplir l'objectif :

./...

- une file contenant les objets pour lesquels on a au moins une définition formelle : UFO
- une file contenant les objets pour lesquels on a au moins une définition informelle : UIN
- une file pour les objets dont on a au moins une définition de type : UTY
- une file contenant les objets pour lesquels on a plus d'une définition formelle : PFO
- une file contenant les objets pour lesquels on a plus d'une définition informelle : PIN
- une file contenant les objets pour lesquels on a plus d'une définition de type : PTY
- une file contenant les objets résultats et les objets pouvant être obtenus par la règle d'utilisation : UTI
- une file contenant tous les objets globaux rencontrés dans l'énoncé analysé : REN
- une file contenant tous les objets locaux rencontrés : LOC

Pour parvenir à vérifier les définitions des différentes files précédentes, d'autres files doivent être introduites : PROVISOIRE et INCERTAIN.

La file UTI est remplie initialement par les résultats de l'énoncé. Pour qu'un nouvel objet y soit introduit, il faut qu'il appartienne à la définition formelle d'un objet présent dans UTI. L'ordre des définitions formelles étant aléatoire (la démarche déductive étant souhaitée mais non imposée), un objet peut être présent dans la définition formelle d'un autre objet sans que l'on puisse savoir si cet objet sera ou non présent dans UTI. Dans ce cas, le couple (objet, objet dans la définition formelle duquel il est présent) est placé dans la file INCERTAIN.

Exemple

soit le fragment d'énoncé suivant :

./...

r: ENO
r=a+b
c=g+h
b=c+y.

Lors de la rencontre de "c=g+h", g et h sont placés dans la file INCERTAIN avec c, et en seront extraits lorsque l'analyse de la définition suivante indiquant que c contribue au résultat.

La file PROVISOIRE contient les objets locaux dont il ne faut pas tenir compte durant l'analyse de tout ou partie du membre droit d'une définition formelle. Elle est destinée à empêcher que ces objets soient placés dans les files REN, UTI et INCERTAIN.

Exemple

soit la définition suivante (à la signification de laquelle on n'attachera pas d'importance) :

`"r tq ex i:$N (i < 20 et qqy:$N(i+y+r > 100))".`
Une occurence de "i" dans le prédicat associé à "ex" ne signifie pas que l'on a rencontré un objet global "i", ni que celui-ci est utilisé pour l'élaboration du résultat. L'objet "i" sera donc mis dans la file PROVISOIRE avant l'analyse du prédicat `"i < 20 et qq y:$N (i+y+r > 100))"`, et "y" y sera placé avant l'analyse du prédicat "i+y+r > 100". Un objet est extrait de PROVISOIRE à la sortie du prédicat lié au quantificateur auquel il est associé.

(PROVISOIRE constitue donc une précondition d'entrée des files UTI, REN et INCERTAIN ou sens que la présence d'un objet dans PROVISOIRE interdit son introduction dans une des files précitées.)

./...

4.1.1.4 Construction

Remarque

Dans ce qui suit, il convient de ne pas confondre mise dans la file xx et insertion dans xx. Dans les 2 cas, si l'élément n'est pas présent dans la file xx, on l'y ajoute. La mise en file est différente de l'insertion dans le sens que la seconde renvoie un signal d'erreur si l'élément était déjà présent dans la file. Une insertion sera donc toujours suivie d'un traitement d'exception.

Algarithme principal

```
*analyse-énoncé (énoncé)
     pour chaque résultat res de énoncé
          faire mettre res dans UTI;
                mettre res dans REN;
          fin-faire;
     pour chaque paramètre par de énoncé
          faire insérer par dans UFO;
                si erreur
                   alors mettre par dans PFO
                fin-si;
                mettre par dans REN;
          fin-faire;
     pour chaque définition def de énoncé
          si def est une définition de type
             alors pour chaque objet ob de def
                          faire insérer ob dans UTY;
                                si erreur
                                   alors mettre ob dans PTY;
                                fin-si;
                                mettre ob dans REN;
                          fin-faire
          ou si def est une définition informelle
             alors pour chaque objet ob de def
                          faire insérer ob dans UIN;
                                si erreur
                                   alors mettre ob dans PIN;
                                fin-si;
                                mettre ob dans REN;
                          fin-faire;
```

./...

```
        ou si def est une définition formelle
              alors pour chaque objet ob de def
                        faire insérer ob dans UFO;
                              si erreur
                                 alors mettre ob dans PFO;
                              fin-si;
                              mettre ob dans REN;
                        fin-faire;
                    analyse-définition (def);
        fin-si;
fin-faire;
afficher-résultats;.

*afficher-résultat
     afficher "pas de définitions de type de:";
     afficher les objets de REN non présents dans UTY;
     afficher "pas de définition formelle de:";
     afficher les objets de REN non présents dans UFO;
     afficher "pas de définition informelle de:";
     afficher les objets de REN non présents dans UIN;
     afficher "plusieurs définitions de type de",PTY;
     afficher "plusieurs définitions formelles de",PFO;
     afficher "plusieurs définitions informelles de",PIN;
     afficher "objets ne contribuant pas au résultat";
     afficher les objets de REN non présents dans UTI;
     afficher "objets ayant signification locale et globale";
     afficher les objets de LOC présents dans REN.

*analyse-définition (def)
     si def est une définition implicite
        alors an-def-impl (partie gauche de def, partie droite
                           de def);
        sinon si def est une définition de suite "pour"
                 alors analyse-pour (def);
                 sinon analyse-normale (partie gauche de def,
                                        partie droite de def);
              fin-si;
        fin-si;.

*an-def-impl (objets, prédicat)
     si prédicat contient une expression quantifiée
        alors mettre l'objet lié dans PROVISOIRE;
              an-def-impl (objets, expression liée au quantifi-
                           cateur);
              retirer l'objet lié de PROVISOIRE;
              mettre l'objet lié dans LOC;
              an-def-impl (objets, prédicat sans le quantificateur
                           ni l'expression quantifiée);
        sinon si un objet ob de objets appartient à UTI
              alors pour chaque objet ob2 de prédicat
                        faire si ob2 n'appartient pas à PROVISOIRE
                                 alors adjonction-uti(ob2);
                                       mettre ob2 dans REN;
                              fin-si;
                        fin-faire;
```

./...

```
          sinon pour chaque objet ob de objets
                    faire pour chaque objet ob2 de prédicat
                              faire si ob2 n'appartient pas à
                                       PROVISOIRE
                                     alors mettre (ob,ob2)
                                           dans INCERTAIN;
                                           mettre ob2 dans
                                           REN;
                                    fin-si;
                              fin-faire;
                    fin-faire;
          fin-si;
     fin-si;.

*analyse-pour(def)
     mettre l'indice dans LOC;
     an-def-impl (partie gauche de def, bornes);
     mettre l'indice dans PROVISOIRE;
     analyse-normale (partie gauche de def, corps de la définition);
     analyse-normale (partie gauche de def, initialisations);
     retirer l'indice de PROVISOIRE;.

*analyse-normale (objets, définition)
     si définition est une conditionnelle
        alors pour chaque alternative de définition
 (*une alternative est de la forme "si cond alors def1 ,def2,...,
                                                              defn)
                    faire an-def-impl (objets, cond);
                          analyse-normale (objets, définitions
                                          d'objet)
                    fin-faire;
        sinon pour tout i
                    faire si l'objet i de objets est dans UTI
                             alors pour chaque objet ob2 de la ième
                                         définition
                                   faire si ob2 n'appartient pas
                                            à PROVISOIRE
                                         alors adjonction-uti
                                                (ob2);
                                               mettre ob2 dans
                                               REN;
                                         fin-si;
                                   fin-faire;
                             sinon pour chaque objet ob2 de la ième
                                         définition
                                   faire si ob2 n'appartient pas
                                            à PROVISOIRE
                                         alors mettre (ob,ob2)
                                               dans INCERTAIN;
                                               mettre ob2 dans
                                               REN;
                                         fin-si;
                                   fin-faire;
                             fin-si;
                    fin-faire;
     fin-si;.
```

./...

```
*adjonction-uti(ob)
    mettre ob dans UTI;
    pour chaque couple (ob1,ob2) de INCERTAIN
        faire si ob1=ob
              alors retirer (ob1,ob2) de INCERTAIN;
                    adjonction-uti(ob2);
              fin-si;
        fin-faire;.
```

4.1.2 Recherche des cycles de définitions d'objets

. Rappel

. Un objet est considéré comme ayant une définition cyclique si les objets contribuant à sa définition formelle (ses descendants) font référence à lui ou à l'un des objets qu'il contribue à définir (ses ascendants) dans leurs propres définitions formelles.

Considérons, par exemple, la suite de définitions :

```
r=s+t
s=u+v
u=1s
v=16
t=b+v
b=r+w
```

Dans cette suite, les objets r, t et b ont des définitions cycliques car r utilise t qui utilise b qui utilise r pour se définir.

De tels cycles rendent une spécification inconsistante, et il est donc souhaitable de les détecter.

Objectif

L'objectif de cet outil de contrôle est la détection de tous les cycles de définition au sein d'un énoncé.

./...

4.1.2.1 Spécification

Il s'agit de construire un algorithme qui, par analyse de l'arborescence d'un énoncé reçue comme argument, devra détecter tous les cycles de définitions présents dans cet énoncé, et, pour chaque cycle, les objets définis dans ce cycle.

Pour l'exemple donné au point précédent, le message suivant s'affichera sur la console de l'utilisateur :

"Les objets r,t,b ont des définitions formelles formant un cycle".

. Précondition

L'énoncé à analyser doit répondre aux exigences suivantes :

- tout objet possède une et une seule définition formelle
- les identificateurs des objets locaux sont différents des identificateurs des objets globaux
- tous les objets contribuent au(x) résultat(s) de l'énoncé.

4.1.2.2 Concepts

Comme il l'est indiqué dans la description du langage, les définitions formelles implicites se distinguent des définitions formelles explicites en ce qu'elles autorisent la présence de l'objet défini dans le corps de sa définition.

Ainsi la définition " x tq a* x^2+b*x+c=o" est une définition implicite correcte permettant de définir x comme étant les racines d'une équation du second degré.

./...

Par contre, la définition "r=f(r)+b" est incorrecte car r se trouve dans les deux membres de la définition explicite.

Les objets référencés dans la définition implicite précédente sont donc a, b et c, tandis que pour la définition explicite, on considère que r et b sont références.

L'objectif du présent algorithme est de détecter dans un énoncé tout objet dont l'ensemble des descendants (ou objets devant être définis pour que l'objet considéré puisse être défini) est d'intersection non vide avec l'ensemble de ses ascendants (objets ne pouvant être définis que si l'objet considéré l'est aussi).

Afin de rendre ce processus plus efficace, il sera établi pour chaque objet une liste dont les éléments sont constitués par les objets à retenir figurant dans le membre de droite de sa définition formelle.

Certains objets figurant dans le corps d'une définition formelle ne seront pas présents dans cette liste. Ainsi, pour les définitions implicites, toute occurence en partie droite des objets définis est ignorée ; les objets locaux à la définition ne sont eux non plus pas présents dans cette liste.

On appellera file r la liste des objets à retenir figurant dans le membre droit de la définition formelle de r. Ces listes sont établies par un seul parcours de l'arborescence initiale, et la présence d'objets à ignorer dans une définition introduit la notion de "cache", constitué par une liste contenant pour la définition analysée les objets qu'il faut ignorer (liste PROVISOIRE).

./...

4.1.2.3 Procédé de construction

```
*detect-cycle (énoncé)
    pour chaque définition formelle def de énoncé
        faire si def est une définition implicite
                 alors provisoire    liste des objets définis
                       par def;
                       an-impl (partie droite de def, provisoire,
                                res)
                       pour chaque objet ob de provisoire
                           faire créer file (ob, res);
                           fin-faire;
                       provisoire    vide;
                 sinon an-expl (def);
              fin-si;
        fin-faire
    pour chaque résultat r de énoncé
        faire f    r;
              analyser (f, file r);
        fin-faire;.

* an-compl (prédicat, provisoire, res)
      si prédicat contient un quantificateur
        alors an-impl (prédicat sans le quantificateur, provisoire,
              res);
              provisoire <- provisoire + objet lié au quantifi-
              cateur;
              an-impl (expression liée au quantificateur,
              provisoire, res);
              enlever le dernier élément de provisoire;
        sinon pour chaque objet ob de prédicat
              faire si ob n'est pas dans provisoire
                       alors res <- res + ob;
                    fin-si;
              fin-faire;
        fin-si;.

*an-expl (def)
    pour chaque objet ob défini par def
        faire créer une file ob;
              créer (ob, partie droite de la définition,
                     provisoire, file)
              pour chaque objet ob2 de file
                  faire si ob2 n'est pas dans provisoire
                           alors file ob <- file ob + ob2;
                        fin-si;
                  fin-faire;
              provisoire <- vide;
        fin-faire;.
```

./...

```
créer (obj, corps, provisoire, file)
    si corps est une définition de suite "pour"
        alors provisoire <- objet d'indiçage;
    fin-si;
    si corps est une définition conditionnelle
        alors pour chaque alternative alt de corps
                faire file <- file + cond de alt;
                    créer (obj, définition d'objet de alt,
                           provisoire, file)
                fin-faire;
        sinon déterminer la définition de corps correspondant à obj;
(dans une définition formelle, plusieurs objets peuvent être
définis, la ième définition d'objet correspond au ième objet à
définir)
        file    file + définition correspondante;
    fin-si;.

analyser (f1, f2)
    pour chaque objet obj de f2
        faire insérer obj dans f1;
            si erreur
                alors définition cyclique (f1,obj);
                sinon si obj n'est pas un argument de l'énoncé
                    alors analyser (f1,file obj);
                          retirer obj de f1;
                    fin-si;
                fin-si;
        fin-faire;.

*définition-cyclique (file,obj)
    déterminer la position de obj dans file;
    afficher "les objets", le reste de la file, "ont des
            définitions cycliques";.
```

4.1.3 Vérification des concordances de type au sein d'un énoncé

Certaines règles présentes dans la grammaire du langage ont trait au type du résultat d'une expression :

- l'objet défini et l'expression définissante doivent être de même type dans les définitions formelles explicite.
- La partie droite d'une définition formelle implicite est un prédicat, et donc une expression dont le type du résultat est booléen.

./...

L'objectif de cette réalisation est la vérification des contraintes portant sur le type des résultats d'expressions .

Deux fonctions ont été distinguées pour remplir cet objectif :

- un processus qui, recevant une expression et l'énoncé dont elle est extraite, évalue le type du résultat de cette expression (l'évaluateur de type)
- un programme qui analyse un énoncé, détermine dans celui-ci les expressions à évaluer, appelle l'évaluateur de type pour les expressions sélectionnées, et vérifie ensuite la concordance entre type attendu et résultat de l'évaluation de type.

SPES étant un langage utilisant les types abstraits algébriques, la vérification de la concordance (ou de l'équivalence) entre deux types abstraits est un problème complexe, relevant encore du domaine de la recherche. Une troisième fonction devrait donc être isolée, ayant pour but de vérifier l'équivalence de deux types abstraits.

Cette troisième fonction n'est pas actuellement implantée du fait de sa complexité et de notre relative ignorance en la matière. Dans ce qui suit, l'on considérera donc que deux types sont équivalents s'ils portent le même identificateur. Par conséquent, seules les deux premières fonctions seront abordées ici, la troisième ayant une implementation trop rudimentaire pour que l'on s'y attache.

La découpe choisie, séparant évaluation du type d'une expression et vérification de la concordance, fut sélectionnée car le processus d'évaluation de type est utilisable à d'autres fins que les vérifications abordées ici. Signalons qu'il intervient notamment dans le processus de génération des structures d'objet (cf1 point 3.4), où, par analyse du dossier, on infère la structure d'un objet.

4.1.3.1 L'évaluateur de type

L'évaluateur de type a pour objectif d'évaluer le type d'une expression de complexité indifférente dans un contexte donné. Un tel processus détectera donc certaines erreurs, et les signalera, mais avant sa réalisation, il a été décidé que sa fonction était uniquement l'évaluation du type d'une expression, et que les erreurs qu'ils signalent sont uniquement celles l'ayant empêché de réaliser son objectif. Ainsi, l'analyse de l'expression "a [i] [j]" restera si l'objet "a" peut être indicé deux fois, et si oui évaluera le type de cette expression. Le processus ne vérifiera pas que les expressions i et j ont un résultat entier, car le fait qu'ils le soient ou non n'a pas d'influence sur le type du résultat de l'expression. C'est donc un processus de niveau supérieur qui devra guider l'évaluateur sur les expressions i et j, et en tirer les conclusions quant aux résultats fournis.

Il est souhaitable qu'un évaluateur de type analyse la totalité de l'expression analysée et y détecte le maximum d'erreurs, et qu'il fournisse un maximum de renseignements sur la manière dont s'est déroulée l'évaluation.

Afin de détecter le maximum d'erreurs et d'analyser la totalité de l'expression, il sera fait appel à certaines constructions du langage SPES. Il existe en effet en SPES des opérateurs qui ne peuvent être surchargés, et dont on connaît domaine et co-domaine. C'est le cas des opérateurs suivants : ==> , <==> , <u>et</u>, <u>ou</u>, <u>non</u> ; les paramètres et résultats de ces opérateurs sont tous de type booléen ($B).

Ainsi, l'évaluation de l'expression non(a) donnera toujours comme résultat $B, que a soit ou non booléen.

Le premier intérêt de ce système est qu'il permet d'analyser au mieux l'expression en entrée. Supposons

./...

par exemple l'expression "non(a)+b" à évaluer, où b est de type $M. Avec ce mécanisme, il est possible de vérifier s'il existe une définition de l'opérateur "+" avec comme domaine $B,$M, et par là soit de détecter une erreur si la définition n'existe pas, soit de continuer l'analyse de l'expression englobante.

Le second intérêt est de fournir un résultat pour ces opérateurs quels que soient leurs arguments, et donc éventuellement de permettre la vérification d'équivalence sur une expression contenant l'un de ces opérateurs avec un argument dont le type ne correspond pas à l'attente.

Dans le texte qui suit, on appellera opérateur-bin-bool les opérateurs suivants : ==> , <==> , "et", "ou", qui sont des opérateurs infixés ayant des arguments de type $B et un résultat de type $B. L'opérateur "non" sera appelé opérateur-un-bool, et est lui préfixé et possède un paramètre de type $B et un résultat de type $B.

a) Spécification

L'évaluateur de type est un programme recevant comme argument une expression et l'énoncé englobant celle-ci, donnés sous forme arborescente. Il est destiné à déterminer le type du résultat de l'expression dans l'énoncé considéré. Compte tenu de son objectif, certains éléments inclus dans l'expression ne seront pas vérifiés, tels le type du résultat d'une expression d'indiçage, le type et le nombre d'arguments présents dans une référence d'énoncé.

Afin d'analyser au mieux l'expression, il sera tenu compte d'opérateurs prédéfinis du langage, dont le type des arguments et résultats sont connus . Les opérateurs prédéfinis sont les opérateurs d'arguments et de résultat de type booléen (non(...),...et...,
ex i:$Q(...),...==> ...,...).

Le processus renverra deux résultats. Le premier indiquera la manière dont s'est déroulée l'évaluation, qui peut soit, s'être déroulée parfaitement, soit s'être arrêtée sans avoir pu remplir l'objectif, soit enfin fournir un résultat correct, mais après avoir remarqué certaines erreurs (p.ex. des arguments d'opérateurs prédéfinis dont le type n'est pas booléen). Le second argument est une liste d'identificateurs de type représentant le type du résultat de l'expression fournie pour autant que l'objectif ait pu être rempli.

b) Procédé de construction

Le processus d'évaluation proposé ici est le processus classique ascendant, qui pour déterminer le type du résultat d'un opérateur détermine le type du résultat de ses arguments, arguments auxquels le même principe est appliqué par récurrence. Ce processus se termine lorsque l'argument est une constante, un objet, un objet indexé, une référence de fonction ou une référence d'énoncé.

Si l'argument est une constante, le nom du noeud indique le type de la constante.

Le type d'un objet peut être déterminé par le recherche de sa définition de type dans l'énoncé englobant. Si celle-ci n'existe pas, la procédure le signalera au processus appelant par le positionnement d'un indicateur, et à l'utilisateur pour l'affichage d'un message.

Pour obtenir le type d'un objet indexé, la première étape consiste à déterminer le type de l'objet par la recherche dans l'énoncé de sa définition de type. Si celle-ci n'existe pas, la procédure se termine de façon identique au cas précédent. En possession de la définition de type, il y a recherche dans la bibliothèque de la description d'expression associée à ce type. Si celle-ci ne s'y

trouve pas ou si la description trouvée n'utilise pas le constructeur "suite", la procédure le signale et renvoit l'indicateur d'erreur. Cette phase de recherche en bibliothèque est appliquée successivement pour chaque expression d'indiçage, la recherche s'effectuant chaque fois sur le paramètre de "suite"présent dans la description précédente. Après n recherches en bibliothèque, où n est le nombre d'indice de l'objet pour autant qu'il n'y ait pas eu d'arrêt prématuré du processus, le type de l'objet indicé est l'identificateur de type utilisé comme paramètre du constructeur "suite" dans la dernière description cherchée en bibliothèque.

Le type d'une fonction sera établi en déterminant le type du résultat de chacun de ses arguments. Si le type de tous les arguments peut être évalué, une recherche est effectuée en bibliothèque d'une fonction ayant même nom et même domaine (type des arguments).

Si la recherche est concluante, le type du résultat de la fonction est obtenue par copie de l'identificateur indiquant la co-domaine de la fonction trouvée dans la bibliothèque.

Si la recherche n'est pas concluante, un message indique à l'utilisateur qu'il s'agit d'une fonction non encore décrite, et un indicateur est positionné afin de le signaler au processus appelant.

Enfin, si le type de tous les arguments ne peut être évalué, un indicateur est positionné afin d'indiquer au processus appelant que la détermination du type n'a pas réussi.

Dans le cas d'une référence d'énoncé, une recherche est effectuée dans le dossier afin d'y trouver l'énoncé. Si celui-ci est absent du dossier, un message indique à l'utilisateur cette absence, et signale l'échec au processus appelant par le positionnement d'un indicateur. Si l'énoncé est trouvé, le type de chacun de ses

./...

résultats est déterminé par recherche de leurs définitions de type dans l'énoncé référencé. Si la recherche est concluante pour tous les résultats, ceux-ci sont renvoyés au processus appelant dans une liste. Si le type de tous les résultats ne peut être évalué, un indicateur est positionné afin de signaler au processus appelant l'échec subi.

Lorsque les types des deux arguments d'un opérateur ont été trouvés, le type du résultat est obtenu, comme dans le cas des des références de fonctions, par recherche dans le bibliothèque du même opérateur avec le même domaine.

Dans le cas des opérateurs prédéfinis, il est vérifié que le type de chacun des arguments est bien ØB, à savoir booléen. Dans le cas contraire, un message est affiché signalant que l'expression correspondante doit avoir un résultat booléen, il est signalé au processus appelant qu'une erreur a été détectée, mais récupérée. Dans tous les cas, le type du résultat est ØB et est renvoyé au niveau supérieur.

Il convient d'évoquer un traitement particulier conçu pour les expressions contenant un quantificateur (ex,qq), dont la forme est "ex objet : ØTYPE (predicat)". Pour permettre l'analyse du prédicat associé, sans avoir à se soucier de l'objet lié au quantificateur, la stratégie suivante est appliquée : création dans le corps de l'énoncé d'une définition de type reprenant l'objet lié et son type déclaré; analyse du prédicat associé; destruction de la définition de type précédemment créée.

Aucune opération de base n'étant présente sur les types prédéfinis, une arborescence a été initialement créée contenant pour les types prédéfinis (ØN, ØR, ØC et ØB) les opérations standards sur des objets de ces types (comparaison, addition, soustraction...). Les profils de ces différentes opérations sont archivés, et seront consultés durant l'évaluation du type du résultat d'une expression. Cependant, ces profils sont moins prioritaires que ceux créés par

./...

l'utilisateur. Ainsi, si le profil "+:ØN,ØN→ØB" est défini par l'utilisateur, et si le profil "+:ØN,ØN →ØN" est défini de façon standard, le type d'une expression "a+b" où a et b sont entiers est ØB. L'arborescence standard prédéfinie n'est dès lors consultée qu'en cas d'échec dans la recherche des profils fournis par l'utilisateur.

c) Conception

La procédure globale, évaluation, possède quatre paramètres.

- exp : l'expression à analyser
- eno : l'énoncé englobant exp.
- des : résultat indiquant comment s'est déroulée l'évaluation
  der vaut 1.- si l'évaluation s'est déroulée sans détection derreurs;
  2.- si l'évaluation s'est déroulée avec détection d'erreurs que l'on a récupérée;
  3.- si l'évaluation n'a pu être effectuée.
- typ : résultat indiquant, si der vaut 1 ou 2, le type du résultat de l'expression exp dans l'énoncé eno.

L'algorithme de ce processus est présent dans l'annexe B de ce rapport.

4.1.3.2 Processus de vérification de type
(Implémentation réalisée par P. Vanhemelryck)

Dans la grammaire du langage, certaines contraintes existent quant au type du résultat d'une expression, contraintes exprimées en fonction de la position dans l'arbre syntaxique de cette expression. Ainsi, par exemple, dans une définition formelle implicite, le membre droit de la définition doit avoir un résultat de type booléen.

Le processus de vérification de type doit, à l'aide de l'évaluateur, contrôler que toutes les contraintes portant sur le type du résultat d'une expression sont vérifiées dans un énoncé.

./...

Il s'agit donc d'un processus qui va analyser l'arborescence, détecter dans celle-ci les expressions sur lesquelles une contrainte porte, demander ensuite d'évaluer le type de cette expression et enfin comparer celui-ci avec le type attendu, en signalant les violations de contraintes lorsque celles-ci surviennent.

La première partie de ce travail consistera à détecter dans la grammaire du langage les contraintes portant sur les types des expressions, et la seconde phase sera constituée par l'implémentation du contrôle de ces différentes règles.

a) <u>Spécification</u>

Il s'agit de construire un programme qui par analyse de l'arborescence d'un énoncé reçu , signalera à l'utilisateur la violation des contraintes du langage portant sur le type des expressions dans un énoncé. Pour parvenir à ces fins, le programme utilisera l'évaluateur de type présenté en 4.1.3.1

b) <u>Relevé des contraintes portant sur le type des expressions</u>

La première contrainte porte sur les définitions implicites, qui sont la forme "objets 1,...,objet n tq prédicat". Il faut que dans ce cas, le type de "prédicat" soit ØB (c.à.d. booléen). Dans les définitions formelles explicites, il doit y avoir une correspondance par paires entre type de l'objet défini et type de l'expression le définissant
Ex. r,s,t, = a+b, c ou d , q

La présence du mot réservé "donnée" en tant que définition formelle d'un objet est réputée valide pour tout objet, et aucune vérification ne doit dans ce cas être effectuée.

./...

L'expression située dans une définition conditionnelle après un "si" ou située dans une définition de boucle de forme "jusqu'à" après le "jqa" doit avoir un résultat de type ØB(booléen).

Dans les définitions de suite de forme "pour", les expressions délimitant les bornes de l'intervalle de définition de la suite doivent être de type ØN (entier),de même que toutes les expressions d'indiçage d'un objet.

Dans les références d'énoncé, il doit y avoir correspondance entre type du paramètre effectif et type du paramètre formel.

c) Procédé de construction

La programme consiste en un balayage d'un énoncé afin d'y sélectionner les définitions formelles.

Pour chaque référence d'énoncé présente dans la définition sélectionnée, il est vérifié qu'un énoncé de même nom est archivé. Si c'est le cas, le type de chaque paramètre effectif est comparé au type du paramètre formel correspondant, et toute non-concordance est signalée à l'utilisateur, de même que l'éventuelle différence entre le nombre de paramètres affectifs et formels.

Si un objet indicé est situé dans la définition à analyser, chacune de ses expressions d'indice voit son type évaluer, et ensuite comparé avec le type attendu (ØN). En cas de différence, l'indice est affiché ainsi qu'un message explicatif.

Si la définition à analyser est une définition implicite, la partie droite de la définition voit son type évaluer.

./...

Si celui-ci n'est pas ØB ou si l'évaluation n'a pu se terminer correctement, un message est affiché signalant que la définition ne vérifie pas la contrainte.

Dans le cas de définition explicite, la première étape consiste en la détermination du type des objets y définis, et ensuite un traitement est appliqué selon le type de la définition.

Si la définition à analyser est une conditionnelle, alors chacune des alternatives voit le type de la condition associée évaluer. Si celui-ci est non évaluable ou différent de ØB, un message le signale à l'utilisateur. Pour chacune des alternatives est alors appliqué le traitement des définitions simples, expliqué par la suite.

Si la définition courante est une définition de suite de la forme "pour", il est d'abord procédé à l'évaluation des expressions définissant les bornes de l'intervalle de définition de la suite, évaluation devant produire le résultat ØN. Si tel n'est pas le cas, un message est affiché, le signalant à l'utilisateur. Les parties "définition du corps" et "initialisation" sont ensuite analysées comme si elles étaient des définitions formelles, car elles sont aussi composées d'une liste d'éléments de suite à définir, et d'une liste de définitions correspondantes. Avant de procéder à cette évaluation, une définition de type est introduite dans l'énoncé, l'objet défini étant l'indice local de la définition, et son type ØN (entier), définition détruite à la fin de l'analyse de la définition.

Dans le cas de définition de suite de la forme "jusqu'à", l'expression reprenant la condition d'arrêt voit son type évaluer. Si celui-ci est distinct de ØB ou non évaluable, alors un message le signale à l'utilisateur. Les parties "définition du corps" et "initialisation" subissent ensuite un traitement identique

./...

à celui présenté par leurs "homologues" des définitions "pour", sans cependant créer une définition de type, aucun objet local n'étant défini.

Le traitement effectué pour les définitions formelles explicites "simples" consiste en une évaluation de chaque définition distincte du mot réservé "donnée", et le résultat est comparé avec l'élément correspondant de la liste des types des objets définis dans la définition courante. Si les types du défini et de l'expression définissante sont distincts, un message le signale à l'utilisateur.

L'algorithme du vérificateur de type peut être consulté dans l'annexe C du présent rapport.

4.2 Vérifications de cohérence effectuées au sein du dossier

L'objectif poursuivi dans cette section est la détection d'incohérences d'un dossier eu égard aux règles de référence des énoncés. Trois règles sont vérifiées : l'existence et l'unicité d'un énoncé racine, l'absence de cycles de références entre énoncés, et enfin la présence dans le dossier de tous les énoncés référencés et l'absence d'énoncés non-racines non référencés.

Un énoncé est dit racine, s'il n'est référencé par aucun autre. Un dossier doit comporter un et un seul énoncé de ce type.

Des références sont qualifiées cycliques si la liste des descendants d'un énoncé contient l'énoncé considéré. La liste des descendants d'un énoncé, composée initialement de tous les identificateurs des énoncés référencés dans l'énoncé considéré, contient pour chaque identificateur d'énoncé X présent dans cette liste, tous les identificateurs des énoncés référencés par X.

Enfin, tout énoncé sur lequel porte une référence dans un énoncé du dossier doit faire partie du dossier, et tout énoncé du

./...

dossier doit soit être l'énoncé racine, soit être un énoncé référencé dans le dossier.

Pour arriver à réaliser ces différents objectifs, la table des pseudo-commentaires reprenant sous forme condensée les références d'énoncé (cf. section 3.1) sera avantageusement utilisée.

4.2.1 Présence d'un et un seul énoncé racine

Un énoncé est dit racine s'il est défini, mais n'est référencé par aucun autre énoncé. Notons en particulier que l'énoncé racine ne peut pas avoir d'arguments, car leur présence implique une référence par un autre énoncé.

4.2.1.1 Spécification

Ce module doit vérifier qu'il existe bien une et une seule racine dans le dossier, ensemble des énoncés archivés.

L'entrée initiale du module est un dossier, et l'effet de son exécution est l'affichage de messages indiquant les éventuelles erreurs trouvées ou l'identificateur de l'énoncé racine.

Les erreurs possibles sont l'absence de racine ou la présence de plusieurs racines. L'absence de racine est dûe au fait que soit tous les énoncés du dossier possèdent au moins une référence, soit que les énoncés pour lesquels aucune référence n'existe possèdent un ou plusieurs arguments.

4.2.1.2 Procédé de construction

La spécification peut être réalisée en résolvant les sous-problèmes suivants :

- recherche dans la table des pseudo-commentaires de tous les énoncés référencés par les énoncés archivés et mémorisation de leurs identificateurs;

- recherche dans le dossier de tous les identificateurs d'énoncés archivés et mémorisation des identificateurs;

- mémorisation des identificateurs trouvés à la deuxième étape et non trouvés à la première; si un seul de ces identificateurs correspond à un énoncé sans argument, il s'agit de l'énoncé racine, si plusieurs ne possèdent pas d'arguments, il y a présence de plusieurs racines; si aucun identificateur n'est mémorisé ou si tous les identificateurs correspondent à des énoncés possédant des paramètres, alors il n'y a pas de racine .

4.2.1.3 Construction

Dans les algorithmes qui suivent, la fonction mise en file XX de l'objet Y, notée XX ⟵ XX+Y, insère l'objet Y dans la file XX pour autant que ceux-ci ne s'y trouvaient déjà; elle est sans effet dans le cas contraire.

```
*Algorithme principal
   rech-référencés(file-référencés);
   rech-définis(file-définis);
   pour chaque identificateur id de file-définis
      faire si id n'est pas présent dans file-référencés
              alors file-racine ⟵ file-racine+id;
           fin-si;
      fin-faire;
   pour chaque identificateur id de file-racine
      faire si l'énoncé id ne possède pas de paramètres
              alors f-rac ⟵ f-rac+id;
           fin-si;
      fin-faire;
   si file-racine est vide ou f-rac est vide
      alors afficher "pas de racine dans le dossier";
      sinon si f-rac contient un seul objet
              alors afficher "une seule racine:",f-rac;
              sinon afficher "plusieurs racines:",f-rac;
           fin-si;
      fin-si;.
```

./...

*Rech-référencés(file)
(dans table des pseudo-commentaires, le premier élément représente dans tous les cas l'énoncé référencé).
pour chaque pseudo-commentaire ps de la table
    faire file ⟵ file + premier objet de ps;
    fin-faire;.

*Rech-définis(file)
pour chaque énoncé archivé eno du dossier
    faire file ⟵ file + identificateur de eno;
    fin-faire;.

4.2.2 Détection des cycles de référence

Ce module doit analyser le dossier et y détecter les cycles de références d'énoncés. Il y a cycle de référence lorsque la liste des descendants d'un énoncé contient l'énoncé considéré. La liste des descendants d'un énoncé est obtenue par l'ajout à la liste des énoncés référencés dans l'énoncé courant de la liste des descendants de chacun de ceux-ci.

Aucune hypothèse n'est émise sur le dossier : il peut posséder 0, 1 ou plusieurs racines. Ce choix a été fait pour obtenir des outils autonomes et pour permettre de signaler l'ensemble des incohérences en une fois plutôt que d'attendre pour la détection de cycle qu'il y ait une et une seule racine.

4.2.2.1 Réalisation

L'ensemble des références d'énoncé à analyser est présent dans la table des pseudo-commentaires. L'analyse sera donc basée sur celle-ci. La structure des éléments présents dans cette table permet une automatisation facile : le premier élément d'un composant de la table est toujours l'identificateur de l'énoncé référencé, et le dernier est toujours l'identificateur de l'énoncé référençant.

Aucune hypothèse n'étant fait sur le groupe, le point de départ de l'analyse est quelconque. On décidera par exemple de commencer l'analyse par le premier composant de la table.

./...

4.2.2.2 Procédé de construction

La première phase de l'algorithme consiste en la sélection d'un pseudo-commentaire dont le dernier élement (l'énoncé référençant) est un identificateur sur lequel aucune analyse n'a été réalisée. Une file contiendra dès lors tous les énoncés ayant déjà été analysés.

Un identificateur d'énoncé étant sélectionné et mémorisé, on va sélectionner successivement tous les énoncés qu'il référencie, par balayage de la table des pseudo-commentaires. Pour chacun de ces énoncés, on vérifie qu'il n'est pas mémorisé. Afin d'effectuer cette vérification, la mémorisation consistera en fait en la mise en pile de l'identificateur rencontré. Si l'identificateur est déjà présent dans cette pile, alors il y a cycle de référence entre la première et la seconde occurence de l'identificateur. Si l'identificateur n'est pas dans la pile, il doit y être inséré, et il devient l'identificateur sélectionné, auquel le processus est appliqué.

Lorsque tous les pseudo-commentaires dans lesquels l'énoncé analysé est situé en dernière position ont été traités, l'identificateur de cet énoncé est retiré de la pile.

Ce processus se termine toujours car il n'y a qu'un nombre fini d'énoncés, et dès lors après un nombre fini de récurrence, on trouvera un énoncé qui soit ne contient aucune référence à d'autres énoncés, soit ne contient que des références à des énoncés déjà présents dans la pile, auquel cas on ne procède qu'à l'affichage des cycles détectés.

Tel qu'il est présenté, l'algorithme effectue beaucoup de travail inutile. Si un identificateur est présent en dernière partie de plusieurs pseudo-commentaires, son insertion dans la file des énoncés déjà analysés fait qu'il ne sera plus sélectionné

./...

par la première phase. Cependant, tous les derniers éléments ne doivent pas être analysés. La sélection initiale doit en effet porter non seulement sur des identificateurs n'ayant pas été sélectionnés mais aussi sur des identificateurs d'énoncé n'ayant pas été traités durant la phase de récurrence. Supposons en effet, la présence d'un énoncé B ayant été sélectionné par le principe de récurrence appliqué au départ de l'énoncé A. Dès lors, l'analyse de A a consisté partiellement en l'analyse complète de B. Si aucun cycle n'a été détecté à partir de A, à fortiori il est impossible d'en détecter à partir de B, et tout cycle détectable à partir de B l'a déjà été par l'analyse ayant comme point de départ A. Il est dès lors inutile de sélectionner B dans la phase initiale (B reste bien évidemment sélectionnable par récurrence).

Pour éviter ce travail inutile, chaque énoncé qu'il soit sélectionné par la première phase ou sélectionné par application de la règle de récurrence, verra son identificateur placé dans une file. La première étape consistera dès lors en la sélection d'un pseudo-commentaire dont le dernier élément n'est pas dans cette file.

4.2.2.3 Conception

Algorithme principal

```
*Détection-cycle
 créer une file énoncé-traité;
 pour chaque composant comp de la table des pseudo-commentaires
      faire si le dernier élément E de comp n'est pas dans énoncé-
                                                          traité
      alors créer un file ascendent;
            ascendant ⟵ E;
            analyse-cycle(E; ascendant, énoncé-traité);
      fin-si
 fin-faire;.
```

```
*Analyse-cycle(eno,ascend,eno-traité)
 eno-traité ←— eno-traité+eno;
 pour chaque composant comp de la table des pseudo-commentaires
      faire si eno est le dernier élément de comp
               alors nouv-eno ←— premier élément de comp;
                     insérer nouv-eno dans ascend;
                     si erreur (*si nouv-eno est dans ascend*)
                        alors afficher "références cycliques pour
                                       les énoncés suivants:"
                                       sous-liste commençant à la
                                       première occurence de nouv-eno
                                       nouv-eno;
                        sinon analyse-cycle (nouv-eno,ascend,
                                             eno-traité);
                              retirer nouv-eno de ascend;
                        fin-si;
               fin-si;
      fin-faire;.
```

4.2.2 Vérification de présence des énoncés référencés

Ce module a le double but - comme le titre ne l'indique pas - de vérifier que tout énoncé référencé par un énoncé du dossier fait lui-même partie du dossier, et qu'il n'existe pas dans le dossier d'énoncés non référencés autres que l'énoncé racine.

Procédé de construction

Afin de déterminer s'il n'existe pas dans le dossier d'énoncés non référencés autres que l'énoncé racine, les résultats du processus vérifiant l'unicité et l'existence de l'énoncé racine seront largement exploités. En disposant d'une liste reprenant les énoncés archivés mais non référencés, il suffit d'y éliminer les énoncés susceptibles d'être racine, à savoir ceux n'ayant pas d'arguments, et la liste contient dès lors les identificateurs des énoncés archivés non racines et non référencés.

Le second objectif, visant à vérifier que tous les énoncés référencés sont archivés emploiera également les résultats créés par ce processus. En disposant de la liste des énoncés référencés et de

la liste des énoncés archivés, il suffit de sélectionner les identificateurs d'énoncés présents dans la première mais non dans la seconde pour obtenir la liste désirée.

Les manipulations effectuées étant élémentaires, aucune information supplémentaire ne sera donnée à leurs sujets.

4.3 Vérifications effectuées au sein de la bibliothèque

Pour rappel, une spécification SPES est composée d'un dossier, liste d'énoncés, et d'une bibliothèque, liste de descriptions de type. La bibliothèque contiendra donc une description de chaque type utile au dossier. Ces descriptions peuvent être des descriptions d'expressions, des descriptions strictes ou des descriptions d'adjonctions.

La bibliothèque doit également obéir à certaines règles qui ne peuvent être vérifiées par l'éditeur syntaxique, car il s'agit de règles de cohérence et non de règles syntaxiques. Des procédures auxiliaires devraient par conséquent pouvoir vérifier des règles de cohérence parmi les descriptions de type. Ainsi, par exemple, les règles suivantes doivent être vérifiées :

- Il ne peut y avoir de cycle de description de type.
- Tout type non prédéfini (les types prédéfinis étant ØN, ØC, ØR ØR, ØB) dont l'identificateur apparaît dans une description de type doit posséder une description dans la bibliothèque.
- Chaque couple formé d'un identificateur de fonction et du profil de ses paramètres doit être unique au sein de la bibliothèque.

Des procédures de contrôle ont été implémentées pour les deux premières règles énoncées, et sont présentées dans les deux sections suivantes .

4.3.1 Vérification de l'existence d'une description pour tout type

4.3.1.1 Spécification

Dans la plupart des descriptions de type, se trouvent des identificateurs de type. Une description - et dès lors un type - n'est donc complète qu'à la condition que chaque type y inter-

venant, via son identificateur, possède lui-même une description, à l'exception des types prédéfinis (à savoir ØN, ØB, ØC, ØR).

L'objectif de ce processus est de signaler à l'utilisateur tous les types utilisés dans la bibliothèque n'ayant pas eux-mêmes de descriptions.

4.3.1.2 Procédé de construction

Ce processus consiste en une analyse de la bibliothèque description par description. Pour chaque description, il doit être vérifié que tous les identificateurs présents correspondent bien à un type possédant une description. La solution la plus simple consiste à rechercher pour chaque type non prédéfini, la description de ce type dans la bibliothèque, et en cas d'échec signaler à l'utilisateur l'absence de description pour ce type. Cette procédure a deux désavantages : sa lenteur et la répétition éventuelle du même message pour le même type. La lenteur est due au fait que pour chaque identificateur, il y a parcours de la bibliothèque, afin de déterminer l'absence ou la présence de description, et donc si un identificateur intervient n fois, n recherches sont effectuées. De plus, dans le cas d'absence de description, n messages sont affichés noyant dès lors l'information nouvelle parmi l'information connue.

Dès lors, chaque identificateur pour lequel une recherche a été effectuée est placé dans une file, et il n'y a parcours de la bibliothèque que si l'identificateur n'est pas dans la file. Afin d'optimiser quelque peu, on insérera également dans la file l'identificateur de type dont la description est analysée. Initialement, cette file contiendra les identificateurs des types prédéfinis afin d'homogénéiser le traitement effectué.

4.3.1.3 Construction

*Existence

```
cas-analyse ←—— ∅N,∅B,∅C,∅R;
pour chaque description desc de la bibliothèque
     faire cas-analyse ←— cas -analyse + identificateur
                                            du type défini
                                            par desc;
           pour chaque identificateur de type
                faire si id n'est pas dans cas-analysé
                      alors recherche en bibliothèque d'une
                            description pour le type id;
                            si recherche non concluante
                               alors afficher "description
                                      manquante pour ce type",id
                               fin-si;
                            cas-analyse ←—— cas-analyse + id;
                      fin-si;
                fin-faire;
     fin-faire;.
```

4.3.2 Détection de cycles de descriptions de type

4.3.2.1 Spécification

Il est possible de définir un type à partir d'autres types, soit par leur emploi en tant que paramètre d'un constructeur dans la description d'expression du type considéré, soit par leur utilisation en tant que type de référence dans les descriptions d'adjonction.

Il importe, pour que les types soient correctement définis, qu'il n'y ait pas de cycles lors de l'utilisation de telles constructions. Il y a cycle de descriptions lorsque l'ensemble des descendants d'un type contient l'identificateur de ce type. L'ensemble des descendants d'un type considéré est constitué :

- des identificateurs de type utilisés dans la description d'expression ou d'adjonction du type considéré;
- de l'ensemble des descendants de chacun des types utilisés dans la description, ensemble trouvé par application de la récurrence.

./...

L'objectif de ce module est de signaler à l'utilisateur tous les cycles de description existant dans la bibliothèque.

4.3.2.2 Procédé de construction

Le procédé de construction est fort proche de celui utilisé pour la détection des cycles de références d'énoncés, aussi nous contenterons-nous d'en rappeler les points principaux. La première étape consiste à choisir dans la bibliothèque une description qui va servir à l'analyse; celle-ci est choisie de façon à correspondre à un type n'ayant jamais été analysé, choix correspondant à son absence d'une file. L'identificateur du type choisi est placé dans une file des cas analysés et au sommet de la pile des descendants.

Pour chacun des identificateurs de type du corps de la description; cet identificateur est placé dans la file des cas analysés; s'il est présent dans la pile des descendants, alors il y a cycle de description, signalé à l'utilisateur. Sinon, si ce type possède une description d'adjonction ou d'expression, il est placé dans la pile des descendants, et sa description subit la même analyse par récurrence à la fin de laquelle l'identificateur est ôté de la pile des descendants.

4.4 Contrôle effectué dans la spécification

L'objet de cette section est la présentation non d'un outil implémenté, mais de concepts pouvant servir de base à l'implémentation d'un analyseur de la cohérence des prédicats associés aux références entre énoncés.

4.4.1 Introduction et concepts

Comme nous l'avons déjà vu, il est possible en SPES

d'associer à un ensemble de définitions un nom d'énoncé, et de faire référence à ce nom au sein d'autres énoncés ce qui permet d'éviter de devoir y introduire l'ensemble des définitions correspondantes. Ces références d'énoncés peuvent être représentées sous forme d'arborescence sans circuit, le langage interdisant les cycles de référence. Un arc de l'arborescence représente une référence d'énoncé, l'origine de l'arc étant l'énoncé référençant et le destinataire l'énoncé référencé. De plus, suivant la définition du langage, on sait que l'arborescence a une seule racine. L'arborescence n'ayant pas de circuit, nous décidons que chaque référence entrainera la création d'un nouveau noeud destinataire, ou encore que chaque noeud à l'exception de la racine, est destinataire d'un et un seul noeud. Il y aura donc au moins autant de noeuds de même nom qu'il y a de références à l'énoncé correspondant.

A chaque arc va être associé un prédicat reprenant la condition de référence. En effet, les références peuvent être présentes dans une expression sur laquelle porte une condition (p.ex. : r= <u>si</u> cond <u>alors</u> A). La condition de référence représente donc la condition devant être vérifiée dans l'énoncé référençant pour que la référence soit utilisée.

Nous appelerons prédicat-chemin d'un noeud, la conjonction des prédicats arcs de la chaîne d'arcs liant la racine au noeud considéré. Un prédicat chemin reprend donc l'ensemble des prédicats à vérifier pour qu'un énoncé soit référencé sur un chemin donné.

Dans ce qui suit, nous appellerons conditions redondantes des conditions apparaissant plusieurs fois dans un prédicat. Des conditions seront réputées contradictoires si leur conjonction, produisant la valeur "faux", interdit la production de la valeur "vrai" pour le prédicat dans lequel elles interviennent.

La finale de ce travail est la détection dans les prédicats chemins de conditions redondantes ou incompatibles.

./...

4.4.2 Procédé de construction

Afin de parvenir à remplir les objectifs, trois sous-problèmes semblent se dégager :

- la création de l'arborescence des références d'énonce, y compris la production des prédicats arcs associés à ces références
- la création pour chaque prédicat-arc d'une table de vérité, reprenant toutes les combinaisons des conditions du prédicat arc permettant l'attribution de la valeur "vrai" au prédicat-arc;
- détection dans les prédicats chemins des conditions contradictoires ou redondantes.

4.4.3 Création de l'arborescence et des prédicats-arcs.

4.4.3.1 Précondition

Le dossier à analyser doit vérifier les conditions suivantes :
- absence de cycles dans les références d'énoncé;
- un seul énoncé n'est référencé par aucun autre; il s'agit de l'énoncé racine;
- les objets mentionnés dans chaque énoncé sont tous correctement définis par rapport à tous les contrôles effectués dans ce travail;
- la définition formelle de chaque objet d'un énoncé doit être de forme explicite, c.à.d. en particulier qu'elle ne peut contenir de quantificateurs, difficilement manipulables.

4.4.3.2 Spécification

Le but de ce module est la construction d'une arborescence représentant les relations de référence entre énoncés, construction réalisée par analyse du dossier.

L'arborescence résultat de cette procédure répondra aux exigences suivantes :

- chaque noeud représente un énoncé;
- un arc représente une référence en provenance de l'énoncé; représenté par le noeud origine de l'arc vers l'énoncé destinataire de l'arc;
- chaque noeud est origine d'autant d'arcs que le corps de l'énoncé correspondant contient de références d'énoncés;
- chaque nœud est destinataire d'un arc au plus ; le seul énoncé n'étant destinataire d'aucun arc est l'énoncé racine; si un énoncé est référencé plus d'une fois dans le dossier, plusieurs noeuds porteront le nom de cet énoncé;
- chaque arc sera étiqueté d'un prédicat-arc reprenant la condition de référence de l'énoncé destinataire de l'arc dans l'énoncé origine; l'absence de condition de référence correspondra à la présence d'un prédicat dont l'expression est "vrai";
- le prédicat-arc devra donc reprendre la correspondance entre l'objet défini et son identificateur; si deux objets distincts ont des noms identiques dans des énoncés différents, les noms présents dans les prédicats-arcs seront distincts; parallèlement, si un objet possède deux noms distincts dans deux énoncés (phénomène observé lors du passage d'agument dans une référence d'énoncé), les prédicats-arcs reprendront cette identicité de l'objet par l'attribution du même nom.

4.4.3.3 Rappels

Avant toute chose, il convient de rappeler la structure des définitions sur lesquelles peut porter une condition dans le langage SPES. Ces constructions sont au nombre de deux : la définition de suite de la forme "jusqu'à" et la définition conditionnelle.

./...

La définition de suite de la forme "jusqu'à" a comme représentation "r=jqa cond rep objet = définition init objet = définition". La définition présente derrière init est appliquée, puis ensuite la définition présente derrière rep appliquée,elle, jusqu'à ce que la condition cond soit établie. Toute référence d'énoncé présente dans cette dernière définition aura donc comme condition de référence non(cond).

La définition conditionelle prend la forme suivante :

r = si cond1 alors def1,
si cond2 alors def2
sinon def3.

Cette formulation est équivalente à la suivante :

r = si cond1 alors def1,
si non(cond1) et cond2 alors def2
si non(cond1) et non(cond2) alors def3.

La condition de référence associée, par exemple,à toute référence présente dans def2 est non(cond1) et cond2 .

4.4.3.4 Procédé de construction

Deux sous-problèmes peuvent être séparés dans l'élaboration de l'arborescence : la création d'une arborescence "nue", et l'adjonction à celle-ci des prédicats-arcs.

La première phase peut être réalisée à l'aide de la table des pseudo-commentaires reprenant sous forme compacte l'ensemble des références d'énoncés du dossier.

L'adjonction des prédicats-arc ne peut être réalisée que par analyse du dossier si du moins l'on respecte la dernière règle de la spécification concernant la correspondance

entre l'objet et son identificateur, correspondance existant dans chaque énoncé mais qu'il faut rétablir au niveau du dossier.

- Création de l'arborescence nue

L'algorithme suivant peut être appliqué afin de construire l'arborescence nue :

- détermination de l'énoncé racine;
- création d'un noeud dont le nom est celui de l'énoncé racine;
- l'énoncé racine devient l'énoncé considéré;
- son noeud devient le noeud considéré;
- analyse de la table des pseudo-commentaires pour chaque énoncé référencé par l'énoncé considéré, création d'un noeud dont le nom est celui de l'énoncé référencé, et d'un arc dont le noeud considéré est origine, et le noeud créé destinataire, le noeud créé devient le noeud considéré et l'énoncé référencé devient l'énoncé considéré, auquel le principe est appliqué par récurrence.

Comme il n'y a pas de cycle de référence, il est certain que cet algorithme se termine.

- Création des prédicats-arcs

Le prédicat-arc entre deux noeuds est soit la condition de référence du destinataire de l'arc dans le corps de l'énoncé correspondant à l'origine de l'arc, soit la constante "vrai" dans le cas où aucune condition ne pèse sur la référence.

Nous ne nous préoccuperons à ce niveau que des références sur lesquelles portent une condition.

Dans ce cadre, deux sous-problèmes interviennent : la détermination de la condition de référence telle que présente dans l'énoncé, et ensuite la transformation de celle-ci de manière à vérifier la correspondance dans tous les prédicats-arcs entre l'objet et son identificateur.

./...

-- Détermination de la condition de référence

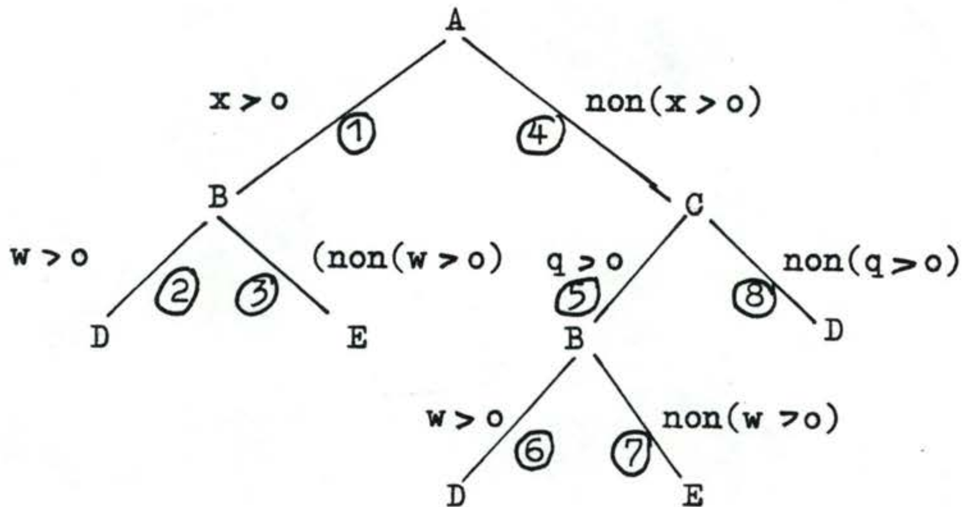
La détermination de la condition de référence peut s'envisager de deux façons : soit détermination précise par transformation d'énoncé (cf. 4.4.3.3 : transformation de la première forme de conditionnelle vers la seconde) soit beaucoup plus simplement par récupération dans la table des pseudo-commentaires de cette information.

La seconde méthode ne nécessitant aucune manipulation sera choisie. On se basera dès lors sur les pseudo-commentaires qui, dans le cas où existe une condition de référence, ont la forme suivante : "A est garde par B dans C", où B est la condition de référence.

Avant d'entamer la seconde phase, précisons quelque peu l'état actuel de notre arborescence par un exemple. Soit la suite de fragments d'énoncés suivante, à la signification desquels nous n'attacherons pas d'importance :

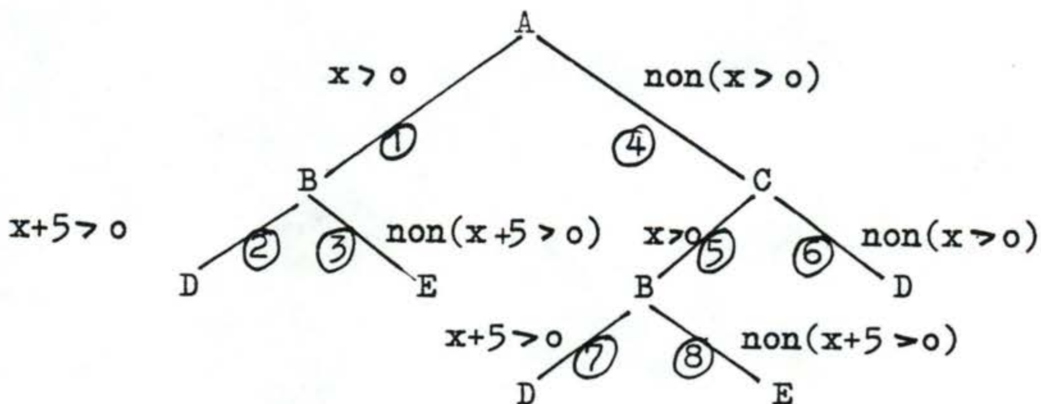
```
r  A
r = si x >0 alors B(x)
    sinon C(x)
.
.
.
r B(y)
.
.
.
r = si  w > 0 alors (w)
    sinon E
w = y+5
.
.
.
r D(a)
.
.
.
r E(a)
.
.
.
r C(q)
r = si q >0 alors B(q)

    sinon D (q)
```

L'arborescence construite à partir de ce dossier à laquelle nous ajoutons les conditions de référence est la suivante :



La transformation à effectuer consiste à rétablir la correspondance entre objet et identificateur. Ainsi, sur les arcs 5 et 8, l'objet identifié par q est en fait l'objet identifié par x dans l'énoncé A. De même, l'objet w des arcs 2 et 3 représente en réalité l'expression x+5, du fait de la définition de w et du passage d'argument lors de la référence de B dans A.

La seconde partie transformera cette première arborescence en la suivante :



./...

C'est à partir de cette nouvelle arborescence que seront vérifiées non-redondance et compatibilité des conditions d'un prédicat-chemin. Un des résultats sera de signaler que le chemin 1, 2, dont le prédicat-chemin est ($x>o$ et $x+5>o$) contient une condition redondante ($x+5 > o$), ou encore que le chemin 1,3 de prédicat-chemin ($x > o$ et non($x+5 > o$)) contient des conditions incompatibles, empêchant d'attribuer la valeur "vrai" à ce prédicat-chemin.

-- <u>Transformation de la condition de référence</u>

En SPES, comme dans la plupart des langages (qu'ils soient de spécification ou de programmation), on peut classer les objets en deux classes : les objets externes, en provenance de l'extérieur, et les objets internes dont la valeur est obtenue en utilisant la valeur des premiers, des constantes et en reliant ceux-ci par des opérateurs et/ou par des fonctions.

Ainsi, dans les prodécures PASCAL, d'utilisation comparable à un énoncé SPES, les objets externes sont les paramètres de la procédure, les variables lues en cours d'exécution de la procédure, et les variables globales définies dans l'environnement de l'appel. Les objets internes sont les variables locales à la procédure, son identificateur, et les variables destinées à recevoir le ou les résultats.

Dans un énoncé SPES, les objets externes sont les paramètres de cet énoncé, et les objets dont la définition formelle indique que ce sont des données pour l'énoncé considéré. Les autres objets (résultats de l'énoncé et objets intermédiaires) sont définis en liant les objets externes et des constantes par des opérateurs ou appels de fonction.

De plus, en SPES, les classes d'objets (externe et interne) sont distinctes, l'appartenance d'un objet à l'une entraînant

./...

son exclusion de l'autre, et tout objet appartient à l'une des classes.

Cette dernière propriété permet de réécrire chaque objet interne présent dans un prédicat-arc par une expression ne contenant que des objets définis au niveau supérieur, des constantes et des objets donnés, où un objet-donné est un objet dont la définition explicite est le mot réservé "donnée".
En itérant ce processus de transformation aux objets définis au niveau supérieur dans l'énoncé duquel ils proviennent, chaque arc possèdera un prédicat-arc exprimé en fonction des objets-donnés; dès lors dans le prédicat-chemin, les objets ayant même signification auront même identificateur (pour autant que, par exemple, l'on suffixe le nom de l'objet par l'énoncé dans lequel il est défini).

Le procédé de construction associé à cette transformation est le suivant :pour chaque objet de l'expression, si l'objet est un objet-donné de l'énoncé, concaténer à son identificateur l'identificateur d'énoncé. Si l'objet est un paramètre de l'énoncé analysé, alors remplacer l'objet par le paramètre effectif, et appliquer à celui-ci la transformation dans le contexte de l'énoncé référençant. Si l'objet n'est ni un objet-donné, ni un paramètre formel, remplacer cet objet par sa définition formelle, à laquelle est à nouveau appliquée la transformation.

4.4.3.5 Construction

*Algorithme principal

```
arborescence-ref ⟵ vide;
chercher l'énoncé racine ENO;
créer un noeud de nom ENO dans arborescence-ref;
arborescence (ENO);.
```

./...

```
*Arborescence (ident)
   pour chaque composant comp de la table des pseudo-commentaires
        faire si le dernier élément de comp est ENO
                  alors créer noeud dont le nom est le premier élément
                                de comp;
                        créer un arc reliant le noeud créé et ident;
                        créer (prédicat-arc,comp)
                        attacher prédicat-arc à l'arc créé;.
                        arborescence (premier élément de comp);
               fin-si;
        fin-faire;.

*Créer (prédicat,pseudo)
   si pseudo indique une référence sans condition
        alors prédicat ⟵ vrai;
        sinon condition (2ème élément de pseudo, cond-resul,
                               3ème élément de pseudo);
              prédicat ⟵ cond-résul;
   fin-si;.

*Condition (expression,cond-résul,eno-référençant)
   cond-résul ⟵ expression;
   pour chaque objet ob de expression
        faire transformation(ob, résul,eno-référençant);
              remplacer dans cond-résul chaque occurence de ob par
                        résul;
        fin-faire;.
*Transformation(ob,res,référençant)
   si la définition formelle de ob dans référençant est "donnée"
      alors res ⟵ concotenation(référençant,ob);
   ou si ob est un paramètre formel de référençant
         alors déterminer l'énoncé ENO qui référencie référençant
                         dans la chaîne courante;
               res ⟵ paramètre effectif correspondant dans ENO à
                    ob;
               transformation(res,ENO,résultat);
               res ⟵ résultat;
   ou si ob est une référence de l'énoncé E
         alors résultat ⟵ définition formelle du résultat de E;
               transformation(résultat,res,E);
   sinon res ⟵ définition formelle de ob dans référençant;
         transformation(res,résultat, référençant);
         res ⟵ résultat;
fin-si;.
```

4.4.4 Construction de la table de vérité associée aux prédicats-arcs.

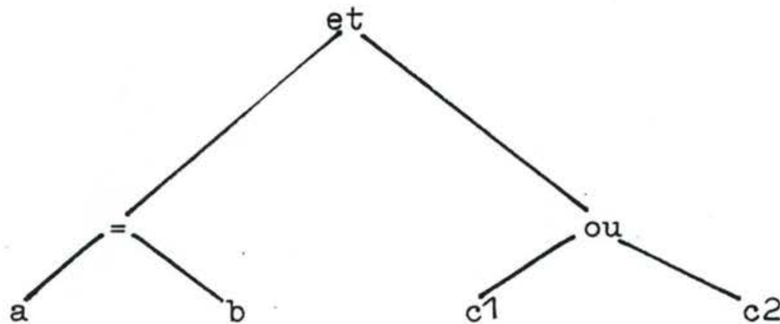
Un prédicat-arc consiste donc en une transformation

./...

de condition de référence, et représente donc une condition devant être vérifiée pour que l'énoncé destinataire de l'arc soit référencé dans l'énoncé origine. A tout prédicat-arc va être associé une table de vérité. Toute ligne dans cette table contient une expression à résultat booléen et une suite de valeurs booléenes. Toutes les lignes de la table de vérité ont même longueur. Les expressions à résultats booléens sont les différentes conditions élémentaires où une condition élémentaire est un terme du prédicat ayant comme caractéristique le fait d'avoir un résultat booléen et une représentation arborescente dont le sommet ne corespond pas à une fonction ou opérateur à opérandes de type booléen.

<u>Exemple</u> :

Soit le prédicat-arc "a=b et (c1 ou c2)", où c1 et c2 sont des objets de type booléen, et a et b des entiers que l'on compare. La représentation arborescente de ce prédicat-arc est :



Les conditions élémentaires de ce prédicat-chemin sont a=b;c1;c2.

Une colonne de cette table de vérité représente une conjonction logique des différentes conditions élémentaires, conjonction permettant l'attribution de la valeur "vrai" au prédicat-arcs.

La ième condition sera affirmée dans la conjonction logique si le ième booléen de la colonne analysée vaut 1 , et niée s'il vaut o.

./...

<u>Exemple</u> :

Le prédicat arc "a=b <u>et</u> (c1 <u>ou</u> c2)" se verra associé à la table vérité suivante

a = b	1	1	1
c1	1	1	0
c2	1	0	1

Le prédicat-arc pourra dont être validé par l'une des trois combinaisons suivantes : (a=b) <u>et</u> c1 <u>et</u> c2,(a=b) et c1 <u>et</u> <u>non</u>(c2),ou (a=b) <u>et</u> <u>non</u>(c1) <u>et</u> c2.
Chaque occurence d'une condition élementaire dans le prédicat-arc se voit attribuer une ligne de décision; il peut donc y avoir dans la table plusieurs lignes ayant la même condition élémentaire.

<u>Spécification</u>

L'objectif de ce module est la production d'une table de vérité associée à un prédicat donné. La table de vérité aura autant de lignes qu'il y a de conditions élémentaires dans le prédicat-arc. Une colonne de cette table représentera une combinaison des différentes conditions élémentaires telle que leur conjonction permet d'attribuer la valeur "vrai" au prédicat-arc.

Le procédé de construction peut être consulté dans l'annexe D du présent rapport.

4.4.5. Détection des redondances et des incompatibilités

L'objectif de cette troisième phase est la détection dans les prédicats-chemins de conditions redondantes ou incompatibles, en s'aidant des tables de vérité associées aux prédicats-arcs.

Une condition est réputée redondante si son évaluation dans le prédicat-chemin n'apporte aucune information nouvelle, c.à.d. si l'espace de variation des objets du prédicat-chemin permettant la satisfaction du prédicat-chemin ne varie pas, si la condition considérée est enlevée du prédicat-chemin.

Une condition est réputée incompatible avec le prédicat-chemin si, compte tenu de la ligne de vérité qui lui est associée, aucune valeur des objets du prédicat-chemin ne permet d'attribuer à celui-ci la valeur "vrai".

L'analyse du prédicat-chemin sera réalisée de la manière suivante, le noeud initial étant le noeud de l'énoncé racine, et le prédicat-chemin initial vide.

Pour chaque arc partant du noeud considéré, il y a analyse du prédicat-arc, en tenant compte du prédicat-chemin existant. Avant l'insertion de chaque ligne du prédicat-arc, dans le prédicat-chemin, il est vérifié que la ligne de vérité n'est ni redondante ni incompatible avec les lignes déjà présentes dans le prédicat-chemin. Si la ligne à insérer est inutile, elle ne sera pas insérée dans le prédicat-chemin, et un message le signalera à l'utilisateur. Si elle est incompatible avec le prédicat-chemin, un message le signalera à l'utilisateur, et l'analyse se terminera pour l'arc courant, ainsi que tous ceux partant du noeud destinataire de l'arc. Si la condition n'est ni redondante ni incompatible,

./...

elle sera insérée dans le prédicat-chemin, moyennant une éventuelle suppression des colonnes dont l'analyse de la ligne de vérité a indiqué qu'elle serait source d'incompatibilité, suppression réalisée dans le prédicat-chemin et dans le prédicat-arc.

Exemple

Soit le prédicat-chemin déjà analysé suivant :

a > 0	1	1	0
b > 0	1	0	1

Soit le prédicat-arc restant :

a > 0	0	0	0
x+y > 20	0	0	1

S'il a été déterminé (nous verrons comment par la suite) que les deux premières colonnes de la première ligne du prédicat-arc rendent incompatibles les colonnes correspondantes au prédicat-arc, l'analyse se poursuivra avec le prédicat-chemin suivant :

a > 0	0
b > 0	1
a > 0	0

Le prédicat-arc devenant

x+Y > 20	1

Cette analyse est réalisée pour chaque ligne du prédicat-arc, pour autant qu'aucun ne soit source d'incohérence. Lorsque toutes les lignes ont été analysées, il faut appliquer le même principe par récurrence aux arcs originaires du noeud destinataire de

l'arc courant, mais après avoir appliqué une modification au prédicat-chemin et au prédicat-arc à analyser. Le prédicat-chemin voit sa partie, table de vérité, reproduite en autant d'exemplaires qu'il y a d'éléments dans la suite de booléens du prédicat-arc suivant, et le prédicat-arc suivant, et le prédicat-arc voit chaque colonne de sa table de vérité reproduite en autant d'exemplaires qu'il y avait d'éléments dans la suite du prédicat-chemin, afin de pouvoir envisager toutes les combinaisons des différentes conditions.

Exemple

Soit les tables de vérité du prédicat-chemin et du prédicat-arc suivantes :

$a > 0$	1	0	1
$x+y < 23$	0	1	1

Table prédicat-chemin

$q+y > 20$	1	0
$y < 30$	0	1

Table prédicat-arc

Après application des reproductions, ces tables sont transformées en les suivantes :

$a > 0$	1	0	1	1	0	1
$x+y < 23$	0	1	1	0	1	1

Table prédicat-chemin

$q+y > 20$	1	1	1	0	0	0
$y < 30$	0	0	0	1	1	1

Table prédicat-arc

Le dernier concept a été introduit seulement à ce moment pour des facilités de compréhension, mais doit, en fait, être pris en considération dès le début, la règle peut être traduite de la façon suivante :

./...

```
*Redondance-incompatibilité (noeud, prédicat-chemin)
   pour chaque arc  partant de noeud
        faire si la longueur d'une ligne du prédicat-chemin est
                                     supérieur à 0
              alors reproduire la table de vérité du prédicat-chemin
                    en autant d'exemplaires qu'il y a de colonnes
                    dans le prédicat-arc;
                    reproduire les colonnes du prédicat-arc en
                    autant d'exemplaires qu'il y a de colonnes dans
                    le prédicat-chemin initial;
              fin-si;
              pour chaque ligne du prédicat-arc
                   faire vérifier (ligne, prédicat-chemin);
                         si ligne incompatible
                            alors l'afficher;
                                  aller en sortie
                            sinon détruire les lignes incompatibles;
                                  si ligne redondante
                                     alors l'afficher;
                                     sinon insérer la ligne
                                       dans le prédicat-chemin;
                                     fin-si;
                            fin-si;
                         fin-faire;
sortie : fin-faire;.
```

L'objet de la suite de cette section est la détermination de la fonction"vérifier"(ligne, prédicat - chemin),qui permet de détecter si la condition est redondante ou incompatible avec le prédicat-chemin, et d'indiquer également les éventuelles colonnes représentant des combinaisons incompatibles.

Afin d'y arriver, nous procéderons en trois phases :

- La première concerne la cohérence interne d'une ligne de prédicat-arc. Nous regarderons dans ce cas s'il n'y a pas d'incompatibilité ou de redondance entre la condition et sa ligne de vérité;
- La seconde phase concerne les interactions pouvant surgir entre la ligne de vérité et chaque ligne de la table du prédicat-chemin;
- La troisième phase concerne les interactions pouvant survenir entre la ligne de vérité et plusieurs lignes de la table du prédicat-chemin, cas que nous ramènerons en fait à l'analyse de la seconde phase.

./...

4.4.5.1 Cohérence d'une ligne de prédicat-arc

Si la partie condition d'une ligne de prédicat-arc ne comporte que des constantes, des traitements particuliers peuvent être effectués.

La présence de constantes uniquement pourra ne pas relever de l'exception, du fait de la substitution des objets par leurs définitions formelles.

Nous supposerons à ce niveau l'existence d'un outil capable d'interpréter une expression booléene afin de déterminer si celle-ci est vraie ou fausse. Dès à présent, il est certain qu'une condition ne contenant que des constantes est une condition inutile.

La ligne de vérité associée à la condition peut, soit contenir des éléments tous identiques (tous vrais ou tous faux), soit contenir des éléments des deux valeurs.

Si l'évaluation de l'expression booléene fournit pour la condition formée de constantes la valeur "vrai" (resp "faux"), les cas suivants sont à envisager :

- si tous les éléments de la suite des valeurs sont "vrai" (resp"faux"); la ligne est inutile sans plus;
- si tous les éléments de la suite des valeurs sont "faux" (resp "vrai") la ligne est incompatible , la condition devant être vraie (fausse) pour le prédicat-chemin, alors qu'il est établi qu'elle est fausse (vraie);
- si la ligne contient des éléments des deux valeurs, il faut marquer les colonnes contenant la valeur "faux" (resp "vrai"), afin de détruire celle-ci par la suite et indiquer de plus que la condition est inutile;

./...

4.4.5.2 Interaction entre une ligne de vérité et une ligne du prédicat-chemin.

La fonction "vérifier", établissant redondance et incompatibilité, doit vérifier que la ligne à insérer n'interagit pas avec chaque ligne prise individuellement dans la table du prédicat-chemin.

Nous allons ici analyser les interactions possibles entre deux conditions.

Dans un premier temps, nous allons introduire les relations pouvant exister entre deux conditions, et les renseignements que l'on peut en tirer, quant à la redondance et l'incompatibilité. Nous verrons ensuite comment détecter l'éventuelle relation existant entre deux conditions, et enfin, comment déterminer les redondances et les incompatibilités.

a) Détermination des relations entre conditions

Afin d'envisager tous les cas nous intéressant de relations entre deux conditions, nous nous baserons sur la recherche de couples de conditions dont l'évaluation des résultats n'est pas indépendante.

En définissant la notation c comme "la condition c est vraie", et non (c) comme "la condition c est fausse", nous avons dès lors analysé les inférences suivantes :

$$c1 \longrightarrow c2$$
$$c1 \longrightarrow non(c2)$$
$$non(c1) \longrightarrow c2$$
$$non(c1) \longrightarrow non(c2)$$

. $c1 \longrightarrow c2$

Nous distinguons deux cas dans lesquels le fait que c1 soit vrai entraîne que c2 doit l'être : l'équivalence et la plus grande contrainte.

* La relation d'équivalence est prise au sens du calcul des prédicats.
ex : x > =20 et 20=< x

Pour établir des règles quant aux interactions, nous procéderons dans ce cas comme dans les suivants, en se basant sur l'idée suivante : soient c1 et c2 des conditions ayant entre-elles la relation étudiée (ici l'équivalence). Nous regarderons les quatre combinaisons des deux valeurs de chacune de ces conditions et en tirerons des conclusions.
Soient c1 et c2 deux conditions ayant entre-elles une relation d'équivalence. Les quatre combinaisons possibles de leurs valeurs sont :

c1	1	1	0	0
c2	1	0	1	0
	(1)	(2)	(3)	(4)

(1) La première combinaison "c1 vrai et c2 vrai" : c1 et c2 étant équivalents, une telle combinaison de valeurs est redondante.

(2) "c1 vrai et c2 faux" : combinaison de valeurs incompatible étant donnéela relation d'équivalence existant entre c1 et c2.

(3) "c1 faux et c2 vrai" : remarque identique.

(4) "c1 faux et c2 faux" : combinaison redondante, étant donnée l'équivalence de c1 et c2.

Dès lors, nous pouvons affirmer que dans le cas où une relation d'équivalence existe entre c1 et c2, la présence dans un prédicat-chemin de c2 est soit redondante, soit incompatible.
Si les lignes de vérité associées à c1 et c2 sont identiques, alors c2 est inutile (cf. cas (1) et (4)).
Si les lignes de vérité associées à c1 et c2 sont différentes, mais possèdent au moins un couple de valeurs identiques, alors les éléments de la ligne de vérité de c2 diffèrent de ceux de c1 constituent des

./...

combinaisons incompatibles, et doivent être marquées afin de permettre la destruction des colonnes correspondantes, c2 étant de plus inutile (cf. cas (2) et (3)).
Si la ligne de vérité associée à c2 est la négation logique de celle de c1, alors c2 est incompatible.

* La relation de plus grande contrainte est celle de l'implication au sens du calcul des prédicats
  ex $x > 30$ et $x > 0$

Soient c1 et c2 deux conditions ayant entre-elles une relation de plus grande contrainte. Les quatre combinaisons possibles des valeurs de 2 booléens sont :

c1	1	1	0	0
c2	1	0	1	0
	(1)	(2)	(3)	(4)

(1) "c1 vrai et c2 vrai" : combinaison de valeur redondante, les conditions étant liées par la relation de plus grande contrainte.
(2) "c1 vrai et c2 faux" : combinaison de valeurs incompatible, au vu de la relation entre c1 et c2.
(3) "c1 faux et c2 vrai" : combinaison réaliste, la négation de la condition la plus contraignante ne permettant pas d'inférer la valeur de l'autre condition.
(4) "c1 faux et c2 faux" : combinaison également réaliste.

De ce qui précède nous pouvons tirer les règles suivantes dans le cas de conditions ayant une relation de plus grande contrainte :

- si la ligne de vérité de c1 ne contient que des éléments "vrai"
    - alors si celle de c2 ne contient que des éléments "vrai"
        - alors c2 est une condition redondante

        si celle de c2 contient des éléments"vrai" et des éléments"faux"

./...

- alors il faut marquer les colonnes où un élément de c2 a la valeur "faux", qui correspondent à des colonnes incompatibles à éliminer, et c2 est redondante.

si la ligne de vérité de c2 ne contient que des éléments "faux"

- alors, la condition c2 est incompatible

- si la ligne de vérité de c1 contient des éléments "vrai" et des éléments "faux"
  - alors chaque élément de la ligne de vérité de c2 valant "faux" et correspondant à une valeur "vrai" pour c1 doit voir sa colonne marquée en vue d'une destruction ultérieure.

. c1 ⟶ non(c2)
================

Nous distinguons deux cas dans lesquels le fait que c1 soit vrai entraîne la négation de c2 : la contradiction et l'implication négative.

* L'implication négative. Deux conditions sont réputées avoir une relation d'implication négative, si la validation de la première implique la certitude que l'autre est logiquement fausse, sans que la négation de la première permette d'inférer la valeur logique de la seconde.
  ex $x = 0$ et $x > 20$

Soit c1 une condition ayant une relation d'implication négative avec c2. Les quatre combinaisons possibles des valeurs de 2 booléens sont

c1	1	1	0	0
c2	1	0	1	0
	(1)	(2)	(3)	(4)

(1) "c1 vrai et c2 vrai" : combinaison de valeurs incompatible, c1 ayant une relation d'implication négative avec c2

(2) "c1 vrai et c2 faux" : combinaison de valeurs redondante, vue la relation entre c1 et c2

(3) "c1 faux et c2 vrai" : combinaison réaliste, la négation de c1 ne permettant pas d'inférer

(4) "c1 faux et c2 vrai" : combinaison réaliste eu égard à la relation entre c1 et c2.

De ce qui précède, nous pouvons tirer les règles suivantes, dans le cas de conditions, ayant entre-elles une relation d'implication négative :

- si la ligne de vérité de c1 ne contient que des éléments de valeur "vrai"
  - alors - si la ligne de vérité de c2 ne contient que des éléments de valeur"vrai"
    alors c2 est incompatible
  - si la ligne de vérité de c2 ne contient que des éléments de valeur "faux"
    alors c2 est redondante
  - si la ligne de vérité de c2 contient des éléments de valeur "faux" et des éléments de valeur "vrai"
    alors il faut marquer les éléments de c2 ayant la valeur "vrai", qui correspondent à des colonnes incompatibles, à éliminer ultérieurement,
    et c2 est redondante
- si la ligne de vérité de c1 contient des éléments des deux valeurs
  alors chaque élément de la ligne de vérité de c2 ayant la valeur "vrai" correspondant à une valeur "vrai" pour c1 doit être marqué en vue d'une destruction ultérieure.

* La contradiction entre deux conditions se traduit par le fait que la validation de l'une entraîne la négation de l'autre et v.v.

./...

ex : x = 0 et x<>0

Soient c1 et c2, deux conditions contradictoires. Les quatre combinaisons possibles des valeurs de 2 booléens sont

c1	1	1	0	0
c2	1	0	1	0
	(1)	(2)	(3)	(4)

(1) "c1 vrai et c2 vrai" : combinaison de valeurs incompatible, c1 étant contradictoire avec c2

(2) "c1 vrai et c2 faux" : combinaison de valeurs redondante, la validation d'une condition entraînant la négation de l'autre

(3) "c1 faux et c2 vrai : combinaison de valeurs redondante

(4) "c1 faux et c2 faux" : combinaison de valeurs incompatible, c1 étant contradictoire avec c2.

De ce qui précède, nous pouvons tirer les règles suivantes, dans le cas de conditions contradictoires :

- si la ligne de vérité associée à c1 est identique à celle de c2, alors c2 est incompatible;
- si la ligne de vérité associée à c1 est la négation logique de celle de c2, alors c2 est redondante;
- si les deux lignes possèdent des éléments semblables et des éléments différents,
  alors il faut marquer les colonnes correspondant à des valeurs semblables des deux lignes, et c2 est redondante.

. non(c1) ⟶ c2
================

Nous distinguons deux cas dans lesquels la négation de c1 entraîne la validation de c2 : la contradiction et le recouvrement.

* La contradiction : cf c1 $\longrightarrow$ non(c2)
* Le recouvrement. Nous appellerons conditions à recouvrement d'espace de validation, des conditions dont l'une au moins est vraie, les deux pouvant l'être.
  ex : x < 20 et x > 10

Soient c1 et c2 deux conditions à recouvrement de valeurs. Les quatre combinaisons possibles de valeurs de deux booléens sont

c1	1	1	0	0
c2	1	0	1	0
	(1)	(2)	(3)	(4)

(1) "c1 vrai et c2 vrai" : situation réaliste, les deux conditions pouvant être simultanément vraies

(2) "c1 vrai et c2 faux" : situation également plausible

(3) "c1 faux et c2 vrai" : idem

(4) "c1 faux et c2 faux" : combinaison entraînant l'incompatibilité, l'une au moins des conditions devant être vraie.

Des quatre points qui précèdent, l'on peut en déduire les règles suivantes, valables pour les conditions en relation de recouvrement :

- si la ligne de vérité de c1 n'est composée que d'éléments "faux"
  alors si la ligne de c2 n'est composée que d'éléments "faux"
  alors c2 est incompatible
- si la ligne de vérité de c1 est composée d'éléments de valeur "vrai" et d'éléments de valeur "faux"
  alors chaque colonne de la ligne de c2 ayant la valeur "faux" correspondant à un élément de c1 ayant la valeur "faux" est marquée, afin d'être détruite ultérieurement.

. non(c1) $\longrightarrow$ non(c2)
=====================

Nous distinguons également deux relations dans lesquelles la négation de c1 entraîne la négation de c2 : l'équivalence et la moindre contrainte.

./...

* L'équivalence : cf. c1 ⟶ c2
* La moindre contrainte; Une condition c1 est en relation de moindre contrainte avec une condition c2 si la négation de c1 entraîne la négation de c2 sans que la validation de c1 permette la négation ou l'affirmation de c2.
  ex : $x > 0$ et $x > 30$

Soit c1 une condition en relation de moindre contrainte avec c2. Les quatre combinaisons possibles des valeurs de deux booléens sont :

c1	1	1	0	0
c2	1	0	1	0
	(1)	(2)	(3)	(4)

(1) "c1 vrai et c2 vrai" : combinaison plausible et utile des valeurs de deux conditions liées par une moindre contrainte, la validation de c1 ne permettant pas d'inférer c2

(2) "c1 vrai et c2 faux" : combinaison plausible et utile pour les mêmes raisons.

(3) "c1 faux et c2 vrai" : combinaison incompatible de valeurs, c1 étant faux, c2 doit l'être aussi.

(4) "c1 faux et c2 faux" : combinaison redondante de valeurs, c1 étant faux, on sait que c2 l'est aussi.

Des quatre points qui précèdent, on peut déduire les règles suivantes, valables pour des conditions c1 et c2, c1 étant moins contraignante que c2 :

- si la ligne de vérité de c1 n'est composée que d'éléments de valeur "faux"
  - alors -si la ligne de vérité de c2 n'est composée que d'éléments de valeur "faux"
    - alors c2 est redondante
  - -si la ligne de vérité de c2 n'est composée que d'éléments de valeur "vrai"
    - alors c2 est incompatible

./...

- si la ligne de vérité de c2 est composée d'éléments des deux valeurs
  alors marquer les colonnes dont les éléments de c2 ont la valeur "vrai" en vue d'une destruction ultérieure
  c2 est redondante
- si la ligne de vérité de c1 est composée d'éléments des deux valeurs
  alors marquer chaque colonne dont les éléments de c2 ont la valeur "vrai" et les éléments de c1 la valeur "faux", en vue d'une destruction ultérieure.

Les quatre inférences initiales (pour rappel : c1 ⟶ c2, c1 ⟶ non(c2), non(c1) ⟶ c2, non(c1) ⟶ non(c2)) sont complètement couvertes par les 6 relations définies entre deux conditions. Nous allons le montrer pour l'une des inférences.

. c1 ⟶ c2
=============

Quatre cas peuvent être envisagés :

c1 ⟶ c2 et c1 ⟶ non(c2) (1)
c1 ⟶ c2 et non(c1) ⟶ c2 (2)
c1 ⟶ c2 et non(c1) ⟶ non(c2) (3)
c1 ⟶ c2 et non(non(c1) ⟶ non(c2)) (4)

(1) : situation impossible
(2) : idem
(3) : relation d'équivalence
(4) : relation de plus grande contrainte.

b) <u>Détection de la relation existant entre deux relations</u>

L'objectif de ce module est de déterminer l'éventuelle relation existant entre deux conditions.

La première hypothèse à poser pour détecter la relation entre deux conditions est que le membre contenant des objets soit identique dans les deux conditions.

Il n'est pas question pour l'instant de chercher une relation entre, par exemple, $a > 0$ et $a+b < 0$. Les détections

porteront sur des couples du type "a > 0 et a < 0" ou "a+4y > =23 et a+4y <> 30".

Pour déterminer l'éventuelle relation existant entre deux relations, il sera fait référence à une base de connaissance reprenant pour les formes canoniques des conditions, les relations existant entre celles-ci.

Cette base de connaissance pourrait prendre la forme d'une liste de règles de types abstraits, le premier membre étant composé de deux conditions, le second indiquant le type de relations avec une éventuelle condition sur les arguments.

```
ex  a>b  et  a>b  == equivalence si  b  =  c;
    a>b  et  a>c  == moindre contrainte si  b<c;
    a>b  et  a>c  == plus grande contrainte si b>c;
    a=b  et  a>c  == implication négative si  b=<c;
    a=b  et  a>c  == plus grande contrainte si  b>c;
    a=b  et  a<>b ==  contradiction;
    a=b  et  a<>c == équivalence si b<>c;
    a<b  et  a>c  == recouvrement si b>c;
    a<b  et  a>c  == implication négative si b = c;
    a<b  et  a>c  == implication négative si b< c.
    ...
```

La détermination de la relation existante entre deux relations comportera typiquement deux phases : l'évaluation des expressions formées de constantes, et une recherche dans la base de connaissance afin de déterminer le cas à envisager.

L'évaluation des expressions doit être effectuée afin de permettre la comparaison de celle-ci, et dès lors d'évaluer la partie condition d'une règle de type abstrait.

./...

La recherche dans la base de connaissance comprendra une recherche d'une règle dont le membre droit est la forme canonique des conditions à analyser, et ensuite validation de l'éventuelle condition présente dans la règle.

c) Détermination des redondances et incompatibilités

L'objectif est de déterminer la redondance ou l'incompatibilité d'une condition face au prédicat-chemin existant. La condition sera comparée successivment à toutes les conditions du prédicat-chemin. L'itération s'arrêtera soit lorsque toutes les conditions ont été utilisées, soit lorsque la condition est incohérente avec une condition, soit enfin lorsque toutes les colonnes de la ligne de vérité de la condition à analyser ont été marquées en vue d'une destruction ultérieure.

4.4.5.3 Interactions entre plusieurs conditions

Nous avons, dans un premier temps, envisagé la cohérence interne d'un couple (condition, ligne de vérité), cohérence interne vérifiable pour des conditions uniquement composées de constantes.

La seconde étape concernait les interactions survenant entre deux conditions, pour autant que celles-ci aient un nombre identique.

Nous allons envisager ici le cas où une condition interagit avec plusieurs conditions simultanément.

Ex : soit les conditions " $a > 0$, $b > 0$ et $a+b > 0$".

La condition "$a+b > 0$" interagit simultanément avec les deux premières.

./...

Pour arriver à traiter facilement ces cas, de nouvelles conditions seront introduites dans le prédicat-chemin, conditions construites à partir de celles déjà présentes dans le prédicat-chemin.

Cette construction sera basée sur les propriétés formelles des opérateurs utilisés.

L'intérêt de cette insertion est de permettre d'analyser pour toute condition uniquement les interactions entre cette condition et chaque condition du prédicat-chemin.

Le mécanisme utilisé sera le suivant, modificant quelque peu l'algorithme redondance-incompatibilité présenté auparavant

Après l'insertion de la ligne de vérité dans le prédicat-chemin, il va être fait appel à une fonction ayant pour but de générer de nouvelles conditions, génération effectuée par application des définitions formelles des opérateurs. Nous nous interrogerons pour chaque condition présente dans le prédicat-chemin, de la possibilité de coupler celle-ci avec la nouvelle condition insérée, afin de déterminer une nouvelle ligne dans la table de vérité.

Une définition formelle des opérateurs consiste en fait en la présentation d'un certain nombre de règles permettant la manipulation de certaines expressions. Un exemple de règle formelle est : "a <b et c <d == a+c <b+d".

Aux conditions ainsi créées seraient adjointes des lignes de vérité pouvant contenir outre les valeurs "vrai" et "faux", la valeur "indécidable" (notée 2 par la suite).

./...

Ex : soit le prédicat chemin :

a<o	1	1	0
bool	1	0	1

le prédicat-arc devant encore être analysé étant :

b <10	1	0	0
a+b < 20	1	1	1

après la vérification de la ligne "b <10   110", celle-ci sera insérée dans le prédicat-chemin qui deviendrait :

a <0	1	1	0
bool	1	0	1
b<10	1	0	0

Cette insertion réalisée, une recherche serait effectuée en vue de combiner les lignes précédentes et la nouvelle, phase qui modifierait la table du prédicat-chemin en :

a<0	1	1	0
bool	1	0	1
b<10	1	0	0
a+b<10	1	2	0

L'analyse de la condition suivante du prédicat-arc, "a+b < 20   111" indiquant que la 3ème colonne est source d'incompatibilité, et ce uniquement par analyse de couples de conditions.

Ce traitement basé sur les expressions formelles des opérateurs, permettrait donc une analyse des cas d'inférence de plus de deux conditions de manière identique à l'analyse de l'inférence de deux conditions.

./...

5. PRODUCTION DE RECAPITULATIFS

Dans ce chapitre, nous aborderons la présentation de quelques récapitulatifs utiles à l'utilisateur.

Nous envisagerons successivement le récapitulatif des profils d'énoncés qui indique pour chaque énoncé le type des arguments et des résultats y associés, et le récapitulatif d'une spécification, qui comprend à la fois le dossier, le récapitulatif des profils d'énoncés et la bibliothèque des types.

5.1 Production de récapitulatif de profils d'énoncés

5.1.1 Opportunité

Lors de l'écriture de spécifications, le spécifieur doit pouvoir disposer rapidement d'informations sur ce qui a déjà été écrit sans qu'il soit obligé de faire de longues consultations.

A cet égard, les profils d'énoncés sont des informations intéressantes. Un profil d'énoncé est un objet structuré qui comprend un identificateur d'énoncé, la liste des types de ses arguments et celle de ses résultats.

Lorsque le spécifieur SPES utilise une référence dans une définition formelle, il doit savoir de quel type sont les arguments et les résultats de l'énoncé auquel il est fait référence, ceci afin d'attribuer les bons types aux objets correspondants dans l'énoncé qu'il traite. Un récapitulatif des profils d'énoncés lui permettra dès lors de trouver rapidement cette information.

La non-existence d'un tel récapitulatif l'aurait contraint à rechercher l'information dans le dossier, ce qui occasionne une perte de temps inutile.

Ce récapitulatif doit pouvoir être consulté à l'écran et se trouver dans l'état imprimé de la spécification (cf. paragraphe 5.2). Il peut également être exploité lors de l'établissement du graphe des références entre énoncés. Il serait en effet intéressant pour le spécifieur de visualiser en même temps dans le graphe, les références entre énoncés, les conditions associées à ces références, et le type des résultats et des arguments de chacun des énoncés représentés dans ce graphe. Ces informations constituent généralement ce que le spécifieur cherche à savoir lorsqu'il désire introduire de nouveaux énoncés dans le dossier.

Signalons que la procédure de sortie graphique du graphe des références entre énoncés n'a pas été implémentée, nous avions cependant prévu l'utilisation du récapitulatif des profils d'énoncés dans ce contexte.

5.1.2 Spécification de la procédure

La procédure doit générer un récapitulatif de profils d'énoncés à partir d'un dossier en mémoire. Pour chaque énoncé du dossier, on produira l'identificateur de l'énoncé et la liste des types des arguments et des résultats.

Le type des arguments et des résultats dans la liste des types d'un profil doit correspondre, pour autant qu'il existe, au type des arguments et des résultats contenus dans le corps de l'énoncé. L'ordre dans lequel sont décrits les types dans les listes d'un profil dépendra de l'ordre d'apparition des objets dans les listes de l'en-tête de l'énoncé. Si pour un objet (résultat ou argument) il n'existe pas, dans le corps de l'énoncé, une définition de type, on indiquera dans la liste des types, à l'endroit correspondant à l'objet, l'absence de cette définition.

Nous renvoyons le lecteur au paragraphe 3.2.2 pour la description de la structure des énoncés SPES.

Chaque profil dans le récapitulatif aura la structure suivante :

Identificateur d'énoncé	(:)	Liste des types des arguments	(-->)	Liste des types des résultats

(Les symboles entourés sont des mots-réservés du langage)

Les listes de types des arguments et des résultats comprennent un nombre de type égal au nombre d'arguments et de

./...

résultats définis dans les listes de l'en-tête d'énoncé. Il existe un type particulier qui indique l'absence de définition de type pour un objet décrit dans l'en-tête d'énoncé. L'identificateur d'énoncé existe toujours.

Pour illustrer ce qui précède, prenons l'exemple suivant :
Supposons le dossier suivant en mémoire :

Commande; validité-commande CONSTRUCTION-COMMANDE-INTERNE
Commande : ∅BC
Validité-commande : ∅B
.
.
.
Commande-valide CONSTRUCTION-COMMANDE-VALIDE (bon-commande)
Commande-valide : ∅BC
.
.
.

L'exécution de la procédure sur cette liste d'énoncés donnera le récapitulatif suivant :

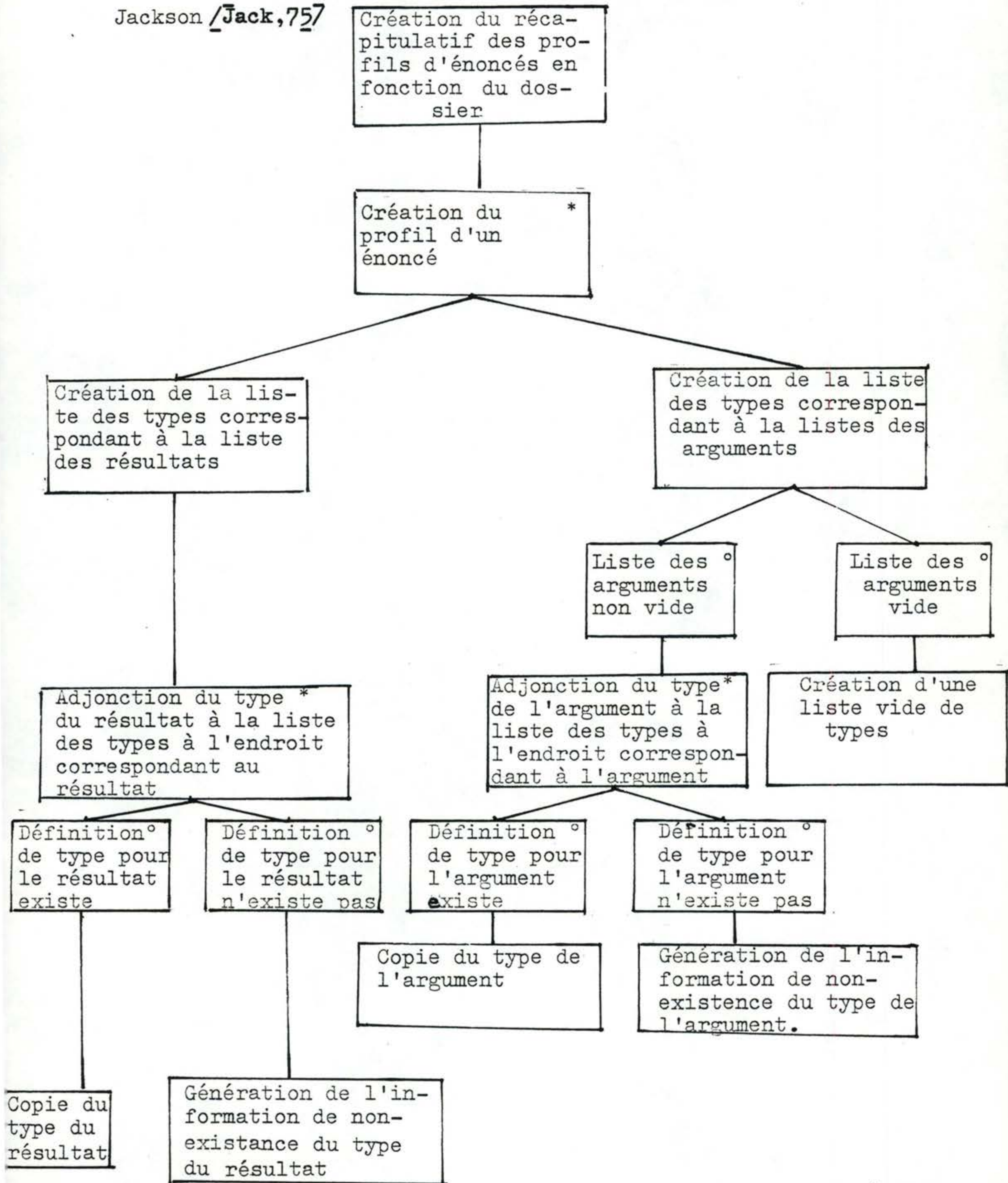
CONSTRUCTION-COMMANDE-INTERNE : --> ∅BC,∅B
CONSTRUCTION-COMMANDE-VALIDE : ∅NON-SPECIFIE --> ∅BC

Nous observons que l'énoncé"CONSTRUCTION-COMMANDE-INTERNE" n'a pas d'arguments. Par conséquent la liste des types des arguments est inexistante. La liste des types des résultats est ∅BC, ∅B indiquant que le premier résultat est d'un type ∅BC défini par l'utilisateur dans une description de type et le deuxième de type booléen (∅B). Dans l'énoncé "CONSTRUCTION-COMMANDE-VALIDE", l'argument "bon-commande" n'est pas typé. Par conséquent, on aura dans la liste des arguments le type "∅NON-SPECIFIE" indiquant l'absence de type pour l'argument. La liste des types des résultats est ∅BC indiquant que le type du résultat est du type ∅BC.

5.1.3 Conception de l'algorithme

5.1.3.1 Structure de l'algorithme

Nous pouvons donner une représentation arborescente de la structure de l'algorithme en utilisant des diagrammes à la Jackson /Jack,75/



./...

5.1.3.2 Détail de l'algorithme

```
PROCEDURE : création du récapitulatif des profils des énoncés
    FAIRE;
       *CREER récapitulatifdes profils VIDE;
        POUR CHAQUE énoncé du dossier
            FAIRE;
            *CREER profil d'énoncé VIDE;
            *RECOPIER identificateur d'énoncé A LA SUITE DE profil
                                                           d'énoncé;
            *FAIRE création de la liste des types correspondant à
                                    la liste des arguments FIN-FAIRE;
            *FAIRE création de la liste des types correspondant à
                                    la liste des résultats FIN-FAIRE;
            *ADJOINDRE profil d'énoncé A LA SUITE DE récapitulatif
                                    des profils;
            FIN-FAIRE;
     FIN-FAIRE;

PROCEDURE : création de la liste des types correspondant à la
            liste des arguments.
    FAIRE;
      *CREER liste de types VIDE;
       SI liste des arguments EXISTE DANS en-tête de l'énoncé
          ALORS POUR CHAQUE argument de la liste des arguments de
                                         l'en-tête de l'énoncé
            FAIRE:
              *RECHERCHER type de l'argument DANS corps de l'énoncé;
              SI type de l'argument EXISTE DANS corps de l'énoncé
                 ALORS *RECOPIER type de l'argument A LA SUITE DE
                                        liste de types
                 SINON *ADJOINDRE type particulier A LA SUITE DE
                                        liste de types;
              FIN-SI;
            FIN-FAIRE;
       FIN-SI
       ADJOINDRE liste de types A LA SUITE DE profil d'énoncé
FIN-FAIRE;
```

./...

```
PROCEDURE : création de la liste des types correspondant à la liste
            des résultats
    FAIRE;
       *CREER liste de types VIDE;
       POUR CHAQUE résultat de la liste des résultats de l'en-tête de
                                                          l'énoncé
           FAIRE :
           *RECHERCHER type du résultat DANS corps de l'énoncé
            SI type du résultat EXISTE DANS corps de l'énoncé
               ALORS *RECOPIER type du résultat A LA SUITE DE
                                  liste de types
            SINON *ADJOINDRE type particulier A LA SUITE DE liste de
                                  types;
             FIN-SI;
           FIN-FAIRE;
      ADJOINDRE liste de types A LA SUITE DE profil d'énoncé
    FIN-FAIRE;
```

5.1.4 Quelques détails sur l'implémentation de la procédure

L'implémentation de l'algorithme s'est faite sous forme d'une procédure Mentol pour plusieurs raisons :

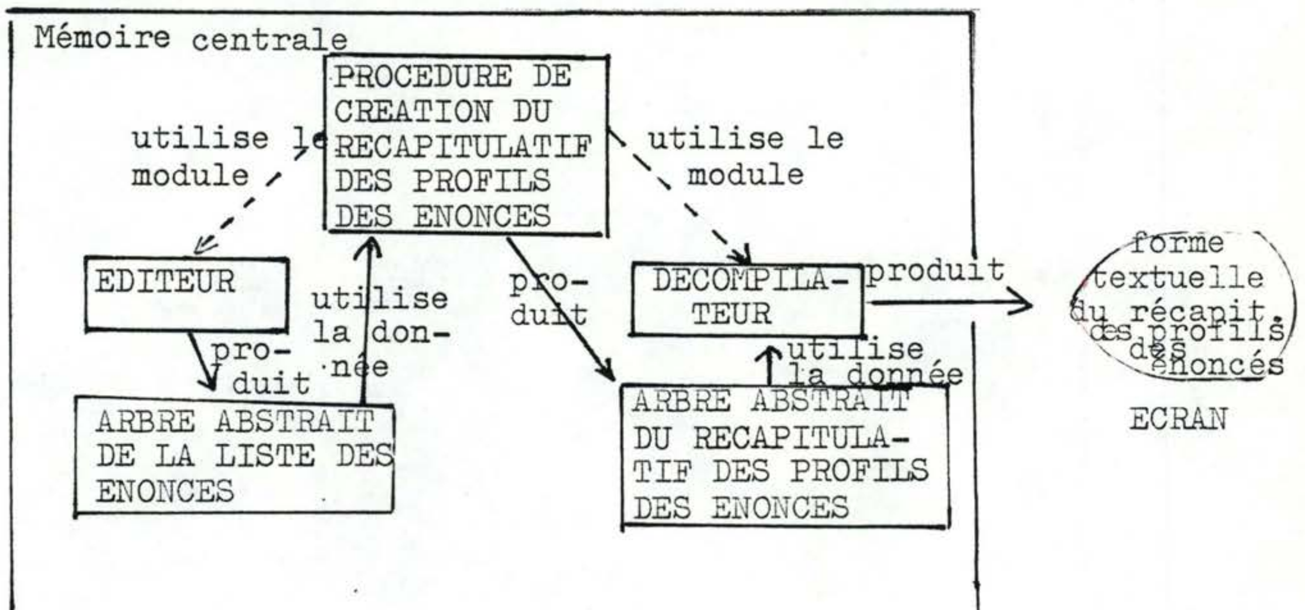
- Elle permet l'utilisation de l'arbre abstrait créé à partir de la liste des énoncés lors de la phase d'édition.
  Nous avons à plusieurs reprises parlé de l'avantage de travailler sur l'arbre abstrait, à savoir qu'il est relativement aisé de rechercher dans le texte certains de ses éléments car l'arbre renferme de l'information sur la structure de ce texte (Ex. : liste des résultats, listes des arguments, ...).

- L'arbre abstrait ne peut être accédé que par le langage Mentol.

Le dossier, les énoncés et leurs composants, ainsi que le récapitulatif des profils des énoncés seront représentés par un arbre en mémoire.

La forme textuelle du récapitulatif est obtenue par décompilation de l'arbre lui correspondant.

Nous pouvons à présent donner un schéma qui situe la procédure dans son contexte :



La procédure doit disposer en mémoire :

- l'arbre correspondant au dossier. Il doit contenir au minimum un énoncé qui peut éventuellement être réduit à un en-tête complet et à une définition.

5.2 Impressions regroupées

5.2.1 Opportunité

L'utilisateur désire obtenir un état imprimé structuré de la spécification comprenant à la fois le dossier, le récapitulatif des profils d'énoncés et la bibliothèque des types, le tout encadré de titres. Il est souhaitable que la liste des énoncés soit ordonnée de manière à ce que les énoncés gardés soient immédiatement suivis de leur garde (pour rappel, les gardes sont aussi des énoncés présents dans le dossier). Ceci permet d'avoir en regard l'énoncé et sa garde et évite à l'utilisateur de devoir chercher ces éléments dans le dossier.

L'utilisateur souhaite également que le texte des énoncés soit structuré en groupes d'éléments. Dans un énoncé en SPES, 5 types d'éléments sont à distinguer : les définitions formelles, informelles, de type, les commentaires et les pseudo-commentaires (c.à.d. les informations concernant les références d'énoncés et les gardes tirées du dossier). Rappelons que ces éléments apparaissent dans un énoncé dans un ordre quelconque. Il est également souhaitable de pouvoir agencer ces différents groupes au choix.

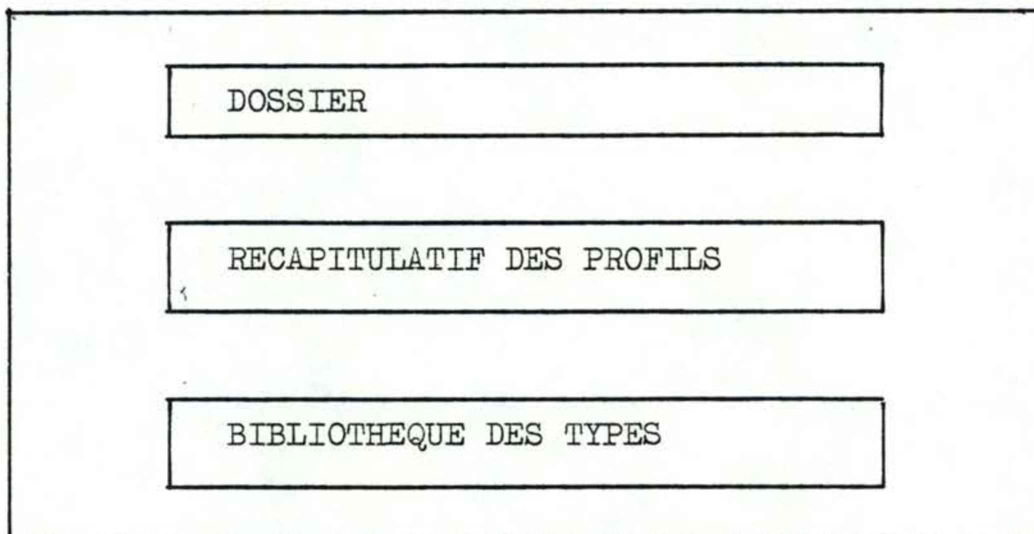
Le regroupement et l'agencement permettent donc de structurer les textes d'énoncés en paragraphes, ce qui permet à l'utilisateur une recherche plus aisée et plus rapide des informations qui l'intéressent.

5.2.2 Spécification de la procédure

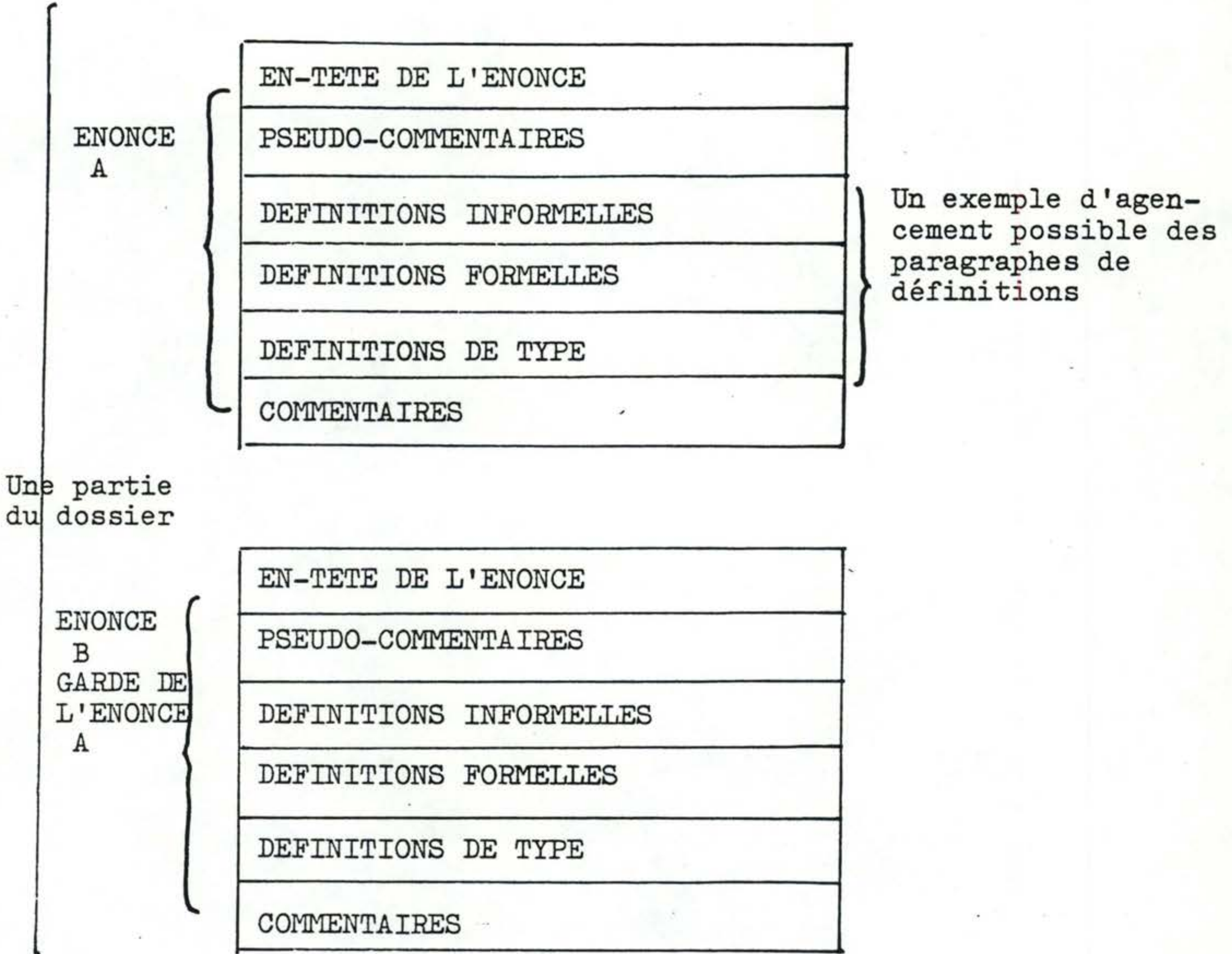
La procédure permettra de créer un état imprimé à partir d'une spécification en mémoire (pour rappel, une spécification comprend le dossier des énoncés et la bibliothèque des types).

Nous renvoyons le lecteur au paragraphe 3.2.2 pour la description de la structure d'un énoncé et au paragraphe 5.1.2 pour la structure des profils d'énoncés.

L'état imprimé aura la structure suivante :



En ce qui concerne la présentation du dossier, nous tiendrons compte des remarques formulées au paragraphe précédent. Toutefois, seuls les définitions formelles, les définitions informelles et les définitions de type seront placées dans l'ordre déterminé par l'utilisateur. Cet ordre sera le même pour tous les énoncés du dossier. Quant aux paragraphes des pseudo-commentaires et des commentaires, ils seront respectivement placés juste après l'en-tête de l'énoncé et à la suite des autres paragraphes de l'énoncé. Nous obtiendrons donc pour un énoncé gardé la structure suivante :



Le dossier de l'état imprimé, comprend les mêmes énoncés que le dossier en mémoire, mais leur ordre et éventuellement leur nombre sont modifiés par rapport au dossier en mémoire. (Pour rappel, les gardes d'énoncés suivent immédiatement les énoncés qu'elles gardent, si un énoncé en garde plusieurs, il est recopié à la suite de chacun des énoncés qu'il garde).

Le récapitulatif des profils d'énoncés sera obtenu à partir du dossier. Nous renvoyons le lecteur au paragraphe 5.1.2 pour la spécification de ce récapitulatif.

La bibliothèque des types comprend un certain nombre de descriptions. Une description de type peut avoir les structures suivantes :

a) <u>description stricte ou une description d'adjonction</u>	b) <u>description d'expression</u>
EN-TETE DE TYPE	EN-TETE DE TYPE
PARTIE OPERATION	DESCRIPTEUR
PARTIE DECLARATION	
PARTIE REGLE	

Nous renvoyons le lecteur au paragraphe se rapportant à la définition du langage SPES pour le contenu de chacune de ces descriptions.

La bibliothèque des types de l'état imprimé sera identique à celle présente en mémoire.

Pour terminer, signalons que sur l'état imprimé doivent figurer des titres adéquats en en-tête :
- de l'état imprimé;
- du dossier;
- du récapitulatif des profils d'énoncés
- de la bibliothèque des types
- d'un paragraphe de définitions informelles
- d'un paragraphe de définitions formelles
- d'un paragraphe de définitions de type
- d'un paragraphe de commentaires
- d'une garde d'un énoncé.

Pour illustrer ce qui précède, nous donnons ci-dessous un exemple de spécification en SPES. Les figures 1 et 2 représentent respectivement le dossier et la bibliothèque des types présents en mémoire. Les figures 3, 4 et 5 représentent l'état imprimé.

DOSSIER EN MEMOIRE

```
   comm,val CONS_COM
comm : $BC
comm ? bon de commande interne
val : $B
val ? resultat booleen indiquant si la commande client est valide ou non
comm,val =  si  COMMANDE_CORR(bc) alors  CONS_COMM_VAL(bc),vrai
 sinon bc,faux
bc : $BONCOM
bc ? bon de commande client
bc = donnee


   co CONS_COMM_VAL (bc)
! CONS_COMM_VAL garde par COMMANDE_CORR(bc) dans CONS_COM
co : $BC
co ? bon de commande interne
bc : $BONCOM
bc ? bon de commande client
co tq id_cli_val(co) = id_cli et li_pro_id(co) = li_pro
id_cli : $ID_CLI
id_cli ? identification du client figurant sur le bon de commande interne
li_pro : $LI_PRO
li_pro ? lignes de commande figurant sur le bon de commande interne
bc tq id_cli_comm(bc) = ident_cli et li_pro_comm(bc) = lib_prod
id_cli =  CONS_ID_CLI(ident_cli)
li_pro =  CONS_LI_PRO(lib_prod)
ident_cli : $ID_CLI
ident_cli ? identification du client figurant sur le bon de commande client
lib_prod : $LI_PRO
lib_prod ? lignes de commande figurant sur le bon de commande client

   r COMMANDE_CORR (bc)
! COMMANDE_CORR garde positive de CONS_COMM_VAL dans CONS_COM
r : $B
bc : $BONCOM
r ? resultat booleen indiquant si la commande est correcte ou non
bc ? bon de commande client
bc tq id_cli_comm(bc) = ident_cli et li_pro_comm(bc) = lib_prod et
                                                   signature(bc) = sign
r =  CLI_IDENTIF(ident_cli) et  LI_PROD_ID(lib_prod) et  SIGNATURE_IDENT(sign)
ident_cli ? identification du client figurant sur le bon de commande client
lib_prod ? lignes de commande figurant sur le bon de commande client
sign ? signature figurant sur le bon de commande client
ident_cli : $ID_CLI
lib_prod : $LI_PRO
sign : $SIGNE
```

(fig. 1)

BIBLIOTHEQUE DES TYPES EN MEMOIRE

```
SORTE $BC==struct [id_cli_val : $IDCLI,li_pro_id : $LI_PRO]


SORTE $BC

 OP id_cli_val : $BC ->$IDCLI
    li_pro_id : $BC ->$LI_PRO
    gr : $IDCLI,$LI_PRO ->$BC

DEC m1 : $BC

REG gr(id_cli_val(m1)!li_pro_id(m1))==m1




SORTE $BONCOM==struct [id_cli_comm : $ID_CLI,li_pro_comm : $LI_PRO,montant_comm : $MONTANT,signature
      : $SIGNE]

SORTE $BONCOM

 OP id_cli_comm : $BONCOM ->$ID_CLI
    signature : $BONCOM ->$SIGNE
    montant_comm : $BONCOM ->$MONTANT
    li_pro_comm : $BONCOM ->$LI_PRO
    gr : $ID_CLI,$SIGNE,$MONTANT,$LI_PRO ->$BONCOM

DEC m1 : $BONCOM

REG gr(id_cli_comm(m1)!signature(m1)!montant_comm(m1)!li_pro_comm(m1))==m1


SORTE $LI_PRO==suite de [$LIGNE]
```

(fig.2)

ETAT IMPRIME

```
% ==========================================================================================
%
%
.                          E T A T   D E   S O R T I E   D E   L A   S P E C I F I C A T I O N
%
%
% ==========================================================================================
%
%
%
%
  comm,val CONS_COM
% ==========================================================================================
%
%
%
% ------------------------------------------------------------------------------------------

%
% L E X I Q U E
%
% ------------------------------------------------------------------------------------------

%
comm ? bon de commande interne
val ? resultat booleen indiquant si la commande client est valide ou non
bc ? bon de commande client
%
% ------------------------------------------------------------------------------------------

%
% D E F I N I T I O N S   D E   S O R T E S
%
% ------------------------------------------------------------------------------------------

%
comm : $BC
val : $B
bc : $BONCOM
%
% ------------------------------------------------------------------------------------------

%
% S T R U C T U R E   D E S   T R A I T E M E N T S
%
% ------------------------------------------------------------------------------------------

%
comm,val =  si  COMMANDE_CORR(bc) alors  CONS_COMM_VAL(bc),vrai
 sinon bc,faux
bc = donnee
%
%
%
%
% ******************************************************************************************

  co CONS_COMM_VAL (bc)
% ==========================================================================================
%
%
! CONS_COMM_VAL garde par COMMANDE_CORR(bc) dans CONS_COM
%
% ------------------------------------------------------------------------------------------

%
% L E X I Q U E
%
% ------------------------------------------------------------------------------------------

%
co ? bon de commande interne
bc ? bon de commande client
id_cli ? identification du client figurant sur le bon de commande interne
li_pro ? lignes de commande figurant sur le bon de commande interne
ident_cli ? identification du client figurant sur le bon de commande client
lib_prod ? lignes de commande figurant sur le bon de commande client
%
```

(fig 3)

```
% ----------------------------------------------------------------------------------------------
%
% D E F I N I T I O N S   D E   S O R T E S
%
% ----------------------------------------------------------------------------------------------
%
co : $BC
bc : $BONCOM
id_cli : $ID_CLI
li_pro : $LI_PRO
ident_cli : $ID_CLI
lib_prod : $LI_PRO
%
% ----------------------------------------------------------------------------------------------
%
% S T R U C T U R E   D E S   T R A I T E M E N T S
%
% ----------------------------------------------------------------------------------------------
%
co tq id_cli_val(co) = id_cli et li_pro_id(co) = li_pro
bc tq id_cli_comm(bc) = ident_cli et li_pro_comm(bc) = lib_prod
id_cli =  CONS_ID_CLI(ident_cli)
li_pro =  CONS_LI_PRO(lib_prod)
%
% ----------------------------------------------------------------------------------------------
%
% S T R U C T U R E   D E   L A   G A R D E
%
% ----------------------------------------------------------------------------------------------
   r COMMANDE_CORR (bc)
% ==============================================================================================
%
%
  COMMANDE_CORR garde positive de CONS_COMM_VAL dans CONS_COM
%
% ----------------------------------------------------------------------------------------------
%
% L E X I Q U E
%
% ----------------------------------------------------------------------------------------------
%
r ? resultat booleen indiquant si la commande est correcte ou non
bc ? bon de commande client
ident_cli ? identification du client figurant sur le bon de commande client
lib_prod ? lignes de commande figurant sur le bon de commande client
sign ? signature figurant sur le bon de commande client
%
% ----------------------------------------------------------------------------------------------
%
% D E F I N I T I O N S   D E   S O R T E S
%
% ----------------------------------------------------------------------------------------------
%
r : $B
bc : $BONCOM
ident_cli : $ID_CLI
lib_prod : $LI_PRO
sign : $SIGNE
%
% ----------------------------------------------------------------------------------------------
%
% S T R U C T U R E   D E S   T R A I T E M E N T S
%
% ----------------------------------------------------------------------------------------------
%
bc tq id_cli_comm(bc) = ident_cli et li_pro_comm(bc) = lib_prod et signature(bc) = sign
r =  CLI_IDENTIF(ident_cli) et  LI_PROD_ID(lib_prod) et  SIGNATURE_IDENT(sign)
%
%
%
% **********************************************************************************************
```

(fig 4)

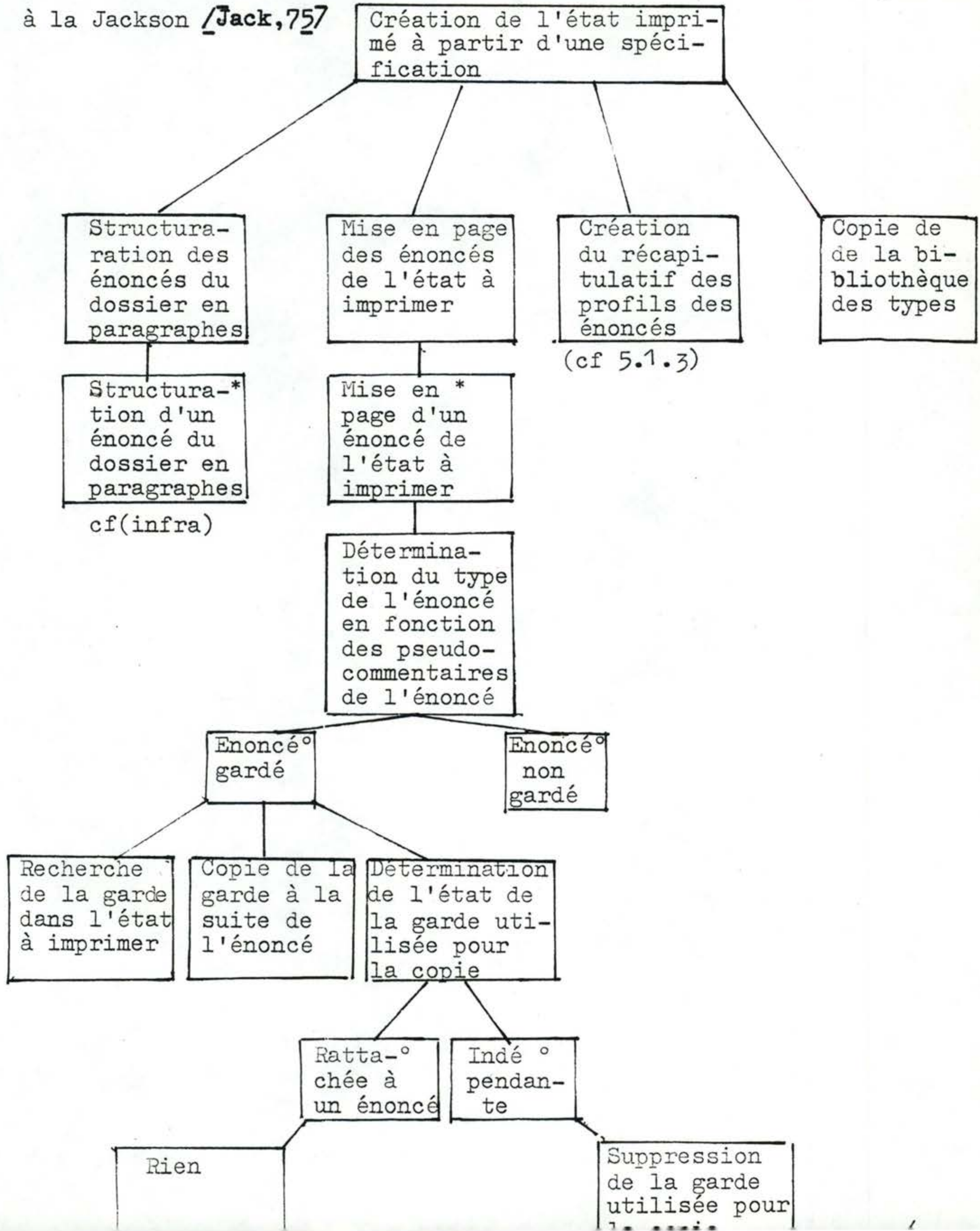
```
% ==========================================================================
%
%
                                   P R O F I L     D E S   E N O N C E S
%
%
% ==========================================================================
%
%
%
%
! CONS_COM : --->$BC,$B
! CONS_COMM_VAL : $BONCOM--->$BC
! COMMANDE_CORR : $BONCOM--->$B

% ==========================================================================
%
%
                                   D E S C R I P T I O N   D E   S O R T E S
%
%
% ==========================================================================
%
%
%
%
SORTE $BC==struct [id_cli_val : $IDCLI,li_pro_id : $LI_PRO]


SORTE $BC

 OP id_cli_val : $BC ->$IDCLI
    li_pro_id : $BC ->$LI_PRO
    gr : $IDCLI,$LI_PRO ->$BC

DEC m1 : $BC

REG gr(id_cli_val(m1):li_pro_id(m1))==m1


SORTE $BONCOM==struct [id_cli_comm : $ID_CLI,li_pro_comm : $LI_PRO,montant_comm : $MONTANT,signature
      : $SIGNE]

SORTE $BONCOM

 OP id_cli_comm : $BONCOM ->$ID_CLI
    signature : $BONCOM ->$SIGNE
    montant_comm : $BONCOM ->$MONTANT
    li_pro_comm : $BONCOM ->$LI_PRO
    gr : $ID_CLI,$SIGNE,$MONTANT,$LI_PRO ->$BONCOM

DEC m1 : $BONCOM

REG gr(id_cli_comm(m1):signature(m1):montant_comm(m1):li_pro_comm(m1))==m1


SORTE $LI_PRO==suite de [$LIGNE]
```

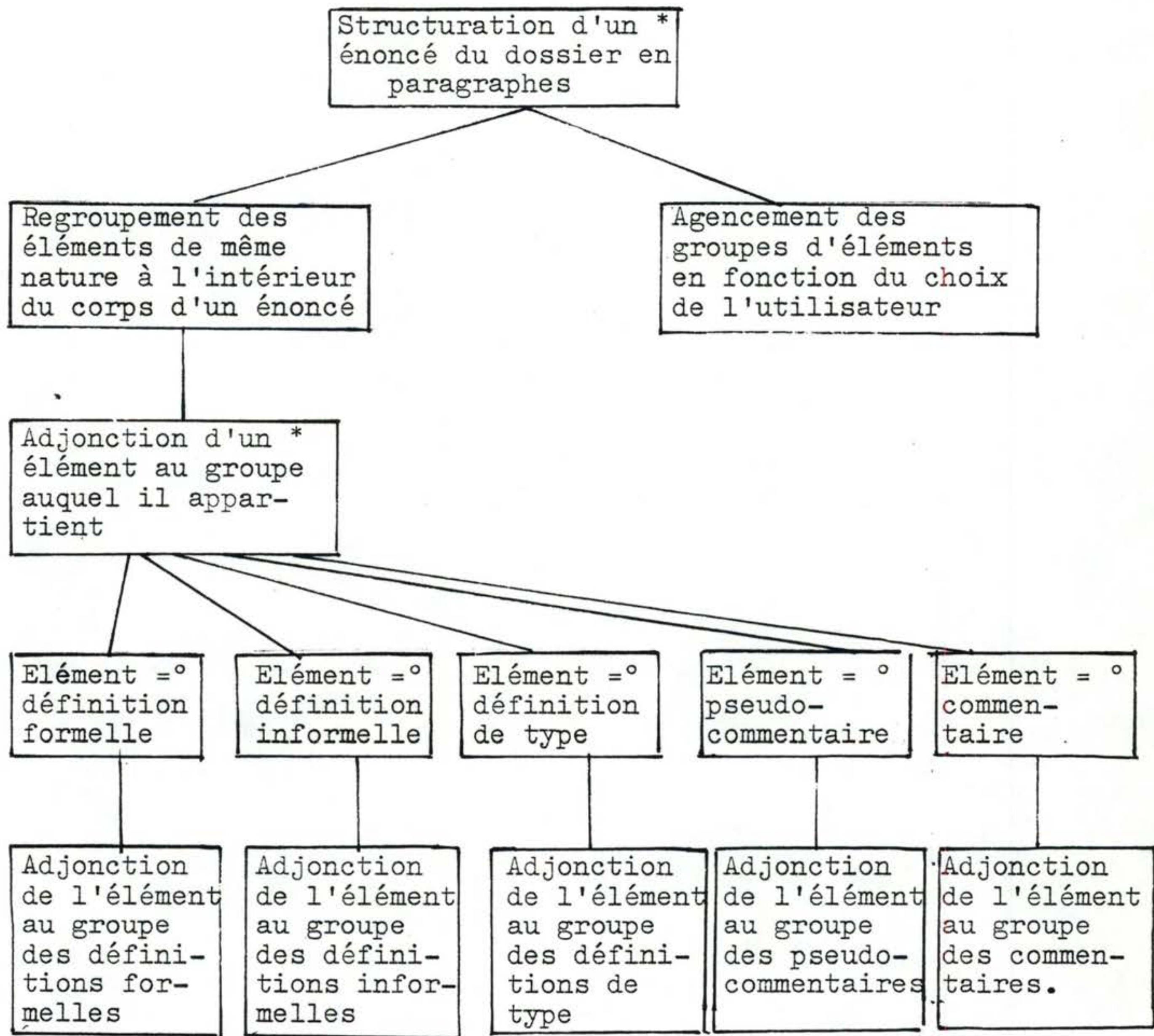
(fig 5)

5.2.3 Conception de l'algorithme

5.2.3.1 Structure générale de l'algorithme

Nous pouvons donner la structure générale de de l'algorithme sous la forme d'un arbre en utilisant des diagrammes à la Jackson **/Jack,75/**





5.2.3.2 <u>Détails de conception de l'algorithme</u>

<u>PROCEDURE</u> : création état imprimé

* <u>FAIRE</u> :
- *<u>CREER</u> état à imprimer <u>VIDE</u>;
- *<u>ADJOINDRE</u> en-tête d'un état <u>A</u> état à imprimer;
- *<u>DEMANDER</u> ordre de présentation des paragraphes de définition;
- *<u>ADJOINDRE</u> en-tête d'un dossier <u>A LA SUITE DE</u> état à imprimer;
- *<u>FAIRE</u> : STRUCTURATION DES ENONCES EN PARAGRAPHES <u>FIN-FAIRE</u>;
- *<u>FAIRE</u> : MISE EN PAGE DES ENONCES <u>FIN-FAIRE</u>;
- *<u>ADJOINDRE</u> en-tête d'une bibliothèque des types <u>A LA SUITE DE</u> état à imprimer;
- *<u>FAIRE</u> : CREATION DU RECAPITULATIF DES PROFILS DES ENONCES <u>FIN-FAIRE;</u>
- *<u>ADJOINDRE</u> récapitulatif des profils <u>A LA SUITE DE</u> état à imprimer;
- *<u>RECOPIER</u> bibliothèque des types <u>A LA SUITE DE</u> état à imprimer;

<u>FIN-FAIRE</u>;

./...

```
PROCEDURE : Structuration-des-énoncés en paragraphes
*FAIRE :
     POUR CHAQUE énoncé du dossier
     *FAIRE :
         *CREER paragraphe-def-informelles VIDE;
         *CREER paragraphe-def-formelles VIDE;
         *CREER paragraphe-def-types VIDE;
         *CREER paragraphe-pseudo-commentaires VIDE;
         *CREER paragraphe-commentaires VIDE;
         *ADJOINDRE en-tête-def-informelle A paragraphe-def-informelles;
         *ADJOINDRE en-tête-def-formelle A paragraphe-def-formelles;
         *ADJOINDRE en-tête-def-type A paragraphe-def-types;
         *ADJOINDRE en-tête-commentaire A paragraphe-commentaires;
          POUR CHAQUE élément du corps d'un énoncé
              *FAIRE :
                  SI élément EST une def-informelle ALORS*RECOPIER
                     définition-informelle A LA SUITE DE paragraphe-
                                                     def-informelles;
                  FIN-SI;
                  SI élément EST une def-formelle ALORS*RECOPIER
                     définition formelle A LA SUITE DE paragraphe-
                                                     def-formelles;
                  FIN-SI;
                  SI élément EST une def-type ALORS*RECOPIER
                     définition de type A LA SUITE DE paragraphe-de-
                                                     type;
                  FIN-SI;
                  SI élément EST un pseudo-commentaire ALORS*RECOPIER
                     pseudo-commentaire A LA SUITE DE paragraphe-
                                                pseudo-commentaires;
                  FIN-SI;
                  SI élément EST un commentaire ALORS*RECOPIER
                     commentaire A LA SUITE DE paragraphe-commentaires;
              FIN-FAIRE;
                  *RECOPIER en-tête de l'énoncé A LA SUITE DE  état à
                                                          imprimer;
                  SI paragraphe pseudo-commentaires CONTIENT AU MOINS u
                     pseudo-commentaire ALORS*ADJOINDRE paragraphe-
                                                pseudo-commentaires;

                  A LA SUITE DE état à imprimer
                  FIN-SI;
                  *ADJOINDRE paragraphe-def-informelles,paragraphe-def-
                             formelles, paragraphe-def-type A LA SUITE
                                             DE état à imprimer;
                             EN FONCTION DE ordre de présentation des
                                             paragraphes de définitions;
                  SI paragraphe-commentaire CONTIENT AU MOINS un
                                                       commentaire,
                    ALORS*ADJOINDRE paragraphe-commentaire A LA SUITE D
                    état à imprimer
                    FIN-SI;
     FIN-FAIRE;
FIN-FAIRE;
```

./...

PROCEDURE : mise en page des énoncés

```
  *FAIRE :
     POUR CHAQUE énoncé de l'état-à-imprimer
       *FAIRE :
        *DETERMINER SI l'énoncé EST gardé EN FONCTION DES
         pseudo-commentaires de l'énoncé;
        SI l'énoncé EST gardé ALORS
        *FAIRE
           *RECHERCHER la garde de l'énoncé DANS l'état à imprimer
           *RECOPIER la garde de l'énoncé A LA SUITE DE l'énoncé;
           SI la garde utilisée pour la copie N'EST PAS rattachée
              à un énoncé ALORS *SUPPRIMER garde non rattachée;
           FIN-SI;
        FIN-FAIRE;
        FIN-SI;
    FIN-FAIRE;
 FIN FAIRE;
```

Pour déterminer si l'énoncé traité est gardé ou non, nous recherchons dans l'énoncé s'il existe un pseudo-commentaire du type :

!	Identificateur de l'énoncé traité	EST GARDE PAR	Identificateur d'énoncé

(Les caractères entourés représentent des mots-réservés du langage

S'il existe, la procédure sait à la fois que l'énoncé est gardé et par quel énoncé il est gardé. Il suffira donc de rechercher cet énoncé dans l'état à imprimer.

La suppression dans l'état à imprimer de la garde utilisée pour la copie, mais non rattachée à un énoncé, permet d'obtenir un état à imprimer dans lequel tous les énoncés jouant le rôle de garde sont rattachés aux énoncés qu'ils gardent.

PROCEDURE : création du récapitulatif des profils des énoncés.
(cf. paragraphe 5.1.3.2)

./...

5.3.4 Quelques détails d'implémentation

L'algorithme a été implémenté sous forme d'une procédure Mentol pour plusieurs raisons :

- Elle permet l'utilisation de l'arbre abstrait créé à partir de la liste des énoncés lors de la phase d'édition. Nous avons, à plusieurs reprises, signalé que cet arbre renferme de l'information sur la structure du texte, ce qui permet de faciliter la recherche de certains de ses éléments (Ex. : définition informelle, formelle, de type ...);

- L'arbre abstrait ne sait être manipulé qu'au moyen du langage Mentol.

Pour créer la partie correspondant à la liste des énoncés dans l'état imprimé, la procédure crée un arbre intermédiaire dans lequel elle placera, au fur et à mesure, les énoncés structurés en paragraphes (cet arbre intermédiaire correspondant à l'état à imprimer de l'algorithme). Pour créer les paragraphes d'un énoncé, la procédure crée 5 arbres : un pour le paragraphe des définitions formelles, un pour les définitions informelles, un pour les définitions de type, un pour les pseudo-commentaires, un pour les commentaires. (Ces 5 arbres correspondent aux 5 paragraphes de l'algorithme). Ensuite, elle parcourt le texte de l'énoncé (= corps de l'énoncé) et recopie le sous-arbre rencontré dans l'arbre qui lui correspond (Ex. : si la procédure rencontre une définition formelle, elle recopie celle-ci dans l'arbre correspondant au paragraphe des définitions informelles). Après traitement d'un énoncé, on obtient 5 arbres qui représentent les 5 paragraphes.

Le récapitulatif des profils des énoncés est obtenu par exécution de la procédure décrite au paragraphe 5.1.

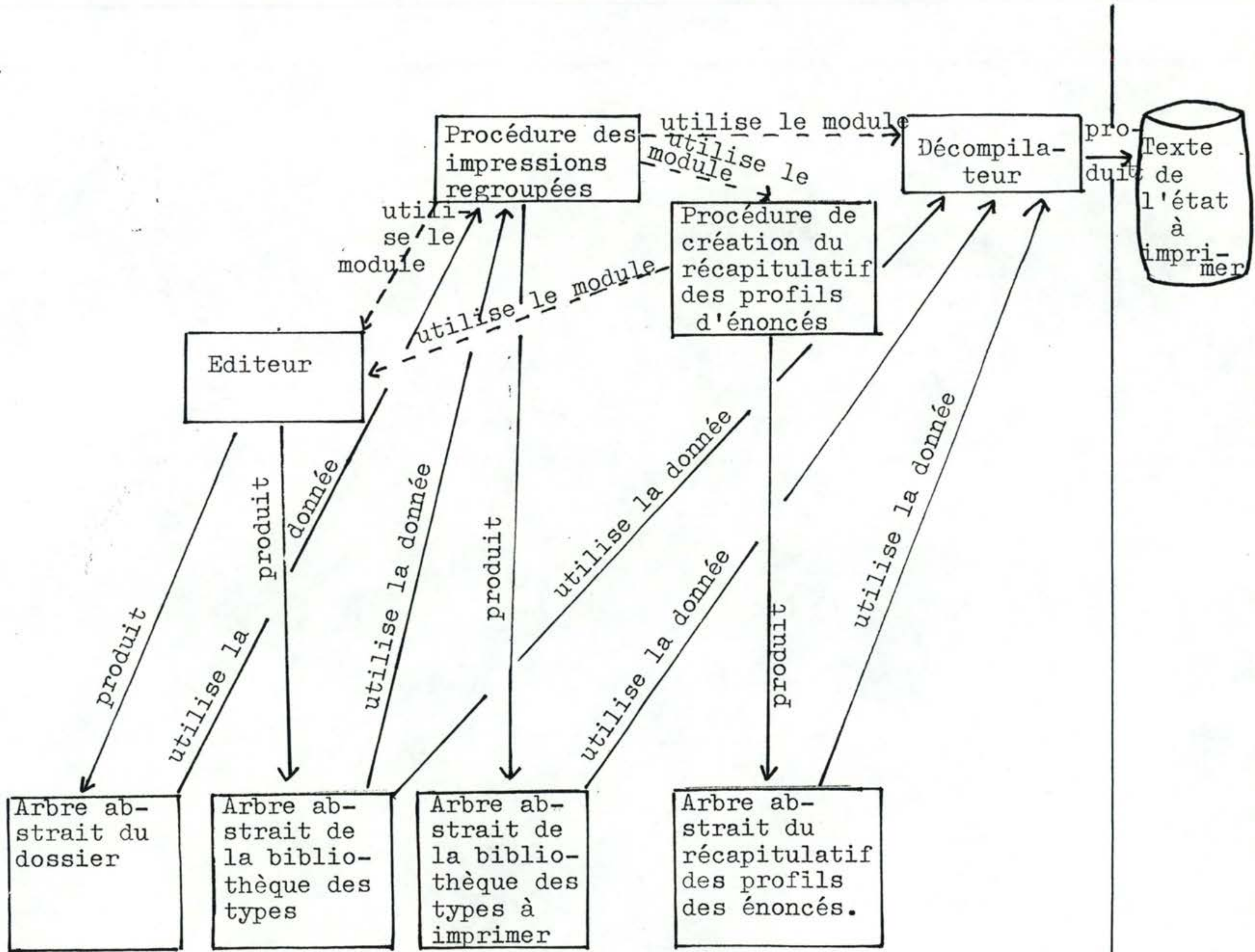
L'impression de l'état se fait en deux temps :

- La procédure utilise le décompilateur pour créer le texte correspondant à l'arbre intermédiaire, à l'arbre du récapitulatif des profils et à l'arbre de la bibliothèque des types. Ce texte est produit sur un fichier au lieu d'être affiché à l'écran, ce qui permet de mémoriser l'état imprimé qui peut alors être obtenu à la demande par impression du texte contenu dans le fichier.

La procédure doit disposer en mémoire centrale de :

- L'arbre représentant la liste des énoncés. Il doit contenir au minimum un énoncé;
- L'arbre représentant la bibliothèque des types. Il doit contenir au minimum une description de type.
- La procédure de création du récapitulatif des profils des énoncés.

Nous pouvons à présent donner un schéma qui situe la procédure dans son contexte.


MEMOIRE CENTRALE
Procédure des impressions regroupées
utilise le module
Décompila-teur
utilise le module
pro-duit
Texte de l'état à imprimer
utilise le module
Procédure de création du récapitulatif des profils d'énoncés
utilise le module
Editeur
produit
produit
donnée
utilise la
utilise la donnée
produit
utilise la donnée
utilise la donnée
produit
utilise la donnée
Arbre abstrait du dossier
Arbre abstrait de la bibliothèque des types
Arbre abstrait de la bibliothèque des types à imprimer
Arbre abstrait du récapitulatif des profils des énoncés.

CONCLUSIONS

En guise de conclusion à ce travail nous proposons d'évaluer dans un premier temps les éléments imposés, à savoir le système MENTOR et le langage de spécification SPES. Nous passerons ensuite à une analyse critique de l'environnement de spécification créé.

Au dernier point, on abordera les extensions possibles à ce travail.

A. EVALUATION CRITIQUE DE MENTOR

L'intérêt principal du système MENTOR, est qu'il permet à un utilisateur de générer un éditeur syntaxique pour un langage donné, par sa description en une forme proche de la notation BNF.

Cet aspect générique du système MENTOR, trop rarement présent dans les autres systèmes, répond aux préoccupations actuelles de la recherche dans le domaine des éditeurs syntaxiques.

Cet aspect générique permet à l'utilisateur de se concentrer uniquement sur la description du langage, sans avoir à se soucier de l'analyse syntaxique, de la représentation interne, de constructions d'arbres...

Le second aspect intéressant du système MENTOR, est qu'il permet à l'utilisateur de se définir un ensemble d'outils constituant un environnement d'utilisation en rapport avec le langage défini.

Les outils créés par l'utilisateur ont la particularité de travailler sur des structures riches, à savoir la représentation arborescente du texte.

MENTOR possède son propre langage de manipulations d'arborescences (MENTOL), contenant des primitives simples et puissantes.

Cependant, notre expérience en tant qu'utilisateurs de MENTOL, nous amène à émettre certaines réserves.

./...

Le principal reproche que l'on peut formuler se rapporte à l'aspect peu lisible et esotérique d'une procédure MENTOL

La compréhension du texte d'une procédure nécessite la connaissance parfaite de la syntaxe du langage et de sa représentation interne.

La forme des structures de contrôle rend tout procédure difficilement compréhensible, car elle empêche de visualiser la structure logique de la procédure.

La mise au point des outils est rendue plus ardue encore par le manque de clarté des messages affichés lors de l'exécution d'une procédure, et surtout par le manque d'information relatif à l'endroit où l'erreur s'est produite.

Ce manque de clarté des messages est également à signaler pour la phase de génération d'un éditeur syntaxique pour un langage donné.

Une grande faiblesse du système MENTOR est la fragilité de sa documentation. Ainsi, par exemple, certaines procédures d'aide à la construction du décompilateur ne possèdent aucune spécification ou information relatives à leur objectif. De même, certaines contraintes liées au système MENTOR ne sont signalées nulle part, et seul leur violation permet de les relever. De plus, certaines notions introduites sont loin d'être compréhensibles, et insuffisamment définies.

Un autre inconvénient du système MENTOR a été relevé par son utilisation. Pour rappel, un texte peut être introduit sous MENTOR de façon semblable aux éditeurs classiques. Dans ce cas, l'analyse syntaxique de ce texte n'est réalisée qu'à la fin de

./...

son introduction. Dès lors, si une erreur est détectée, lors de l'analyse syntaxique, il y a impossibilité de générer un arbre représentant ce texte, et l'ensemble du texte créé est perdu.

Le dernier point de l'évaluation du système MENTOR concerne l'aspect fermé de celui-ci.

L'implémentation des outils manipulant les arborescences n'est en effet prévue, pour le moment, qu'en MENTOL. Il est regrettable que le système ne permette pas l'utilisation d'autres langages (tels que PASCAL...). Cependant, dans l'évolution actuelle du système MENTOR, cette possibilité est prise en considération.

B. <u>EVALUATION CRITIQUE DE SPES</u>

Avant toute chose, rappelons que notre objectif n'était pas de définir ce langage, même si dans la réalité, nous avons consacré beaucoup de temps à essayer d'en préciser la structure et d'en comprendre les mécanismes fondamentaux.

Une première critique formulable à son propos, est l'aspect peu naturel du langage. Ce langage est d'un usage relativement peu aisé pour un informaticien moyen, par son aspect trop mathématique et théorique.

Ce langage attend encore d'être confronté à des spécifications de problèmes de taille importante, afin d'être éventuellement modifié en fonction de l'enseignement qu'on en aura retiré.

Il est à signaler que même si ce langage a été conçu sur base de petits exemples, nous avons déjà pu le confronter à un problème de plus grande importance, à savoir la validation d'une commande dans le cas PETITPAS.

Eu égard à la relative jeunesse du langage, il est permis d'espérer une amélioration allant à l'encontre des remarques formulées ci-dessus.

C. APPRECIATION DES OUTILS DEVELOPPES

Les outils développés forment une partie seulement de l'environnement de spécification qu'il est possible de créer autour du langage SPES.

Les outils créés ont été choisi, parmi ceux qui vraisemblablement seront utiles au SPES-ifieur.

La principale critique que l'on puisse faire à leur propos est la redondance des contrôles effectués. Ces redondances sont nécessaires, eu égard à l'absence d'ordre précis d'utilisation de ces outils.

Nous avons essayé de créer un ensemble d'outils représentatifs d'un environnement de spécification : outils d'aide à l'introduction, outils de contrôle des règles sémantiques, outils de présentation...

Notre expérience en tant qu'utilisateur de l'environnement créé, nous suggère fortement le regroupement des différents outils en les intégrant dans un pilote.

Un pilote est un programme de plus haut niveau, servant de séquenceur aux différents outils, et obligeant le respect d'une démarche de spécification.

Un pilote possédant une vue globale de la spécification peut assurer le respect des règles sémantiques dès l'introduction du texte, permettant ainsi la suppression des redondances

./...

des contrôles des règles sémantiques dans les outils développés.

Le pilote servant de séquenceur des différents outils, il deviendra donc possible d'établir des préconditions sur ceux-ci, préconditions permettant la suppression de certaines vérifications devant être envisagées en l'absence d'hypothèse, quant à la séquence.

La création de ce pilote aurait pour conséquence la négation de l'hypothèse prise quant au non-dirigisme des outils, mais semble devoir malgré tout, s'imposer comme devant être une des prochaines améliorations du système développé.

D. EXTENSIONS POSSIBLES A CE TRAVAIL

Outre l'adjonction d'un pilote, devant être réalisée prioritairement, d'autres outils pourraient être ajoutés à l'environnement créé.

Une première adjonction utile est celle d'outils graphiques permettant notamment la visualisation du graphe des références entre énoncés, avec mention des conditions de références d'un énoncé dans un autre et des profils des énoncés apparaîssant dans le graphe.

Un second outil qu'il nous paraît intéressant de développer consiste en des fonctions de création et de consultations de bases de connaissances.

Celles-ci contiendraient de l'information relative à un domaine d'activité spécifique. L'information contenue serait par exemple composée de la structure des objets fréquemment utilisés dans un type de spécification (gestion clientèle, gestion de stock...) ainsi que des propositions de sémantique informelle pour ces objets.

Un troisième outil pouvant compléter utilement l'environnement de spécification, est un analyseur de cohérence et de complétude.

L'analyse de cohérence consiste à vérifier que toutes les alternatives d'une définition conditionnelle sont susceptibles d'être validées.

L'analyse de complétude consisterait à contrôler que l'ensemble des cas prévus corresponde à l'ensemble des cas à prévoir.

ANNEXE A

MENTOL (GUY, 80)

Nous donnons dans cette annexe la description de la syntaxe des commandes MENTOL. Dans ce qui suit, les caractères soulignés représentent des mots-réservés du langage. Les accolades indiquent qu'un choix doit se faire entre plusieurs éléments et les crochets indiquent les parties optionnelles.

1) Le repère

Un repère est une variable contenant l'adresse mémoire du noeud qu'il désigne. Un repère s'écrit :

@<nom de repère>

Signalons aussi que chaque schéma-prédéfini possède un repère qui lui est propre. Par exemple, le repère correspondant au schéma prédéfini "conditionnelle" sera @conditionnelle.

2) Les commandes de déplacement dans l'arbre

La syntaxe de ces commandes est la suivante :

<repéré> { U | S | L | R } [{ <entier> | * }]

La commande :

- "U" (up) permet d'atteindre dans l'arbre un noeud ascendant du noeud considéré.
- "S" (son) permet d'atteindre dans l'arbre un noeud descendant du 1er ordre du noeud considéré.
- "L" (left) permet d'atteindre dans l'arbre un noeud situé à gauche du noeud considéré. Ces deux noeuds se trouvent sur un même niveau dans l'arbre.
- "R" (right) permet d'atteindre dans l'arbre un noeud situé à droite du noeud considéré. Ces deux noeuds se trouvent sur un même niveau dans l'arbre.

./...

Le déplacement dans l'arbre se fait toujours à partir du noeud désigné par le repère. L'entier indique le facteur de répétition de la commande. Ainsi, la commande @AL5 permettra d'atteindre le 5ème noeud situé à gauche du noeud désigné par le repère @A, et serait équivalente à @ALLLLL.

L'entier peut être omis, auquel cas la valeur 1 sera prise par défaut. L'entier peut être remplacé par le caractère "*" indiquant que la commande doit être répétée jusqu'à ce qu'il ne soit plus possible de l'exécuter.

Notons, pour terminer, que dans le cas de la commande S, l'entier désigne plutôt l'ordre d'apparition du noeud dans le sous-arbre considéré (le parcours se fait de la gauche vers la droite).

Dans les paragraphes suivants, nous parlerons souvent d'adresse structurelle. L'adresse structurelle est constituée soit d'un repère, soit d'un repère et d'une des commandes détaillées ci-dessus.

3) Assignation d'un repère à un noeud

La syntaxe d'une assignation d'un repère à un noeud est la suivante :
soit <repère> : <adresse structurelle>
ou <repère> : &

La première règle correspond à la manière d'indiquer au système qu'un repère doit désigner un noeud situé à l'adresse structurelle.

Lorsque le symbole & est utilisé, on indique au système que le repère devra désigner la racine de l'arbre correspondant au texte introduit à la console.

./...

4) Manipulation d'arborescences

4.1 Copie

La commande de copie <repère> = <adresse structurelle> permet à l'utilisateur de créer un arbre qui est la copie de l'arbre dont la racine est située à l'adresse structurelle donnée. Ensuite le repère précisé dans la commande sera assigné à la racine de l'arbre créé.

4.2 Destruction

La commande de destruction D <adresse structurelle> permet à l'utilisateur de détruire l'arbre dont la racine est donnée par l'adresse structurelle.

4.3 Remplacement

La commande de remplacement <adresse structurelle> C <adresse structurelle> a pour but de remplacer l'arbre dont la racine est donnée par l'adresse structurelle de droite, par l'arbre dont la racine est donnée par l'adresse structurelle de gauche. MENTOR vérifie que l'arbre à substituer a pour racine un opérateur appartenant au phylum requis à l'endroit du remplacement. En cas d'échec, la commande n'est pas exécutée.

5) Recherche d'une occurence d'un sous-arbre particulier

La syntaxe de ces commandes est la suivante :

<adresse structurelle> F <repère>
<adresse structurelle> FF <repère>

Elles permettent la recherche de la première occurence

d'un sous-arbre correspondant au sous-arbre dont la racine est donnée par le repère, dans le sous-arbre dont la racine est donnée par l'adresse structurelle. La recherche s'effectue en parcourant l'arbre en profondeur d'abord et en largeur ensuite. La commande F effectue uniquement cette recherche dans le sous-arbre, tandis que la commande FF effectue cette recherche dans le sous-arbre, mais aussi dans les sous-arbres situés à droite du noeud désigné par l'adresse structurelle. Après exécution le repère@K désignera la racine du sous-arbre correspondant à l'occurence recherchée.

6) Affichage

La commande d'affichage <adresse structurelle>P <entier> permet à l'utilisateur de visualiser à l'écran le sous-arbre dont la racine est donnée par l'adresse structurelle. L'entier indique la profondeur de l'arbre à afficher. Rappelons que cette commande fait appel au décompilateur.

7) Les structures de contrôle

7.1 La séquence de commandes

La séquence de commandes est constituée d'un nombre quelconque de commandes séparées par le caractère ";" .

Les commandes sont exécutées en séquence de gauche à droite. L'exécution d'une commande peut résulter en un succès, ou en un échec. Si une des commandes ne peut s'exécuter, on interrompt la séquence de commandes. Un indicateur permet de savoir si la séquence de commandes s'est déroulée normalement ou si elle a été interrompue (cas d'échec d'une commande). Dans le cas où la commande qui ne peut s'exécuter est suivie d'une commande de traitement d'exception, on poursuivra normalement la séquence de commandes.

./...

7.2 La boucle

La boucle n'est rien d'autre qu'une séquence de commandes itérée. Elle s'écrit de la manière suivante :

(séquence de commandes) [{ <entier> / * }]

L'entier désigne le facteur de répétition de la séquence de commandes. Il peut être omis, dans ce cas il sera pris égal à 1. L'entier peut être remplacé par le caractère "*" indiquant que la séquence de commande sera itérée indéfiniment. Le contrôle de la boucle est souvent exprimé à l'intérieur de la séquence de commandes à l'aide, par exemple, des commandes de traitement d'exception et de sortie de boucle.

7.3 La sortie de boucle

La commande de sortie de boucle s'écrit :

Ø [{ <entier> / * }]

Cette commande provoque la sortie de boucle sur un certain nombre de niveaux d'imbrication. Ce nombre est précisé par l'entier. Il peut être omis, dans ce cas il sera pris égale à "1". Il peut être remplacé par le caractère "*", dans ce cas on sortira de tous les niveaux d'imbrication.

7.4 La commande de traitement d'exception

L'exécution d'une commande peut se solder par un succès ou un échec. Un indicateur détiendra une de ces valeurs après l'exécution d'une commande.

./...

La commande de traitement d'exception permet de tester la valeur de cet indicateur et selon les cas, effectuer l'un ou l'autre des traitements y précisés.

Elle s'écrit

?< commande 1> ,< commande 2 >

Si la commande précédant la commande de traitement d'exception a réussi , la 1ère commande sera exécutée, sinon la 2ème. Les commandes 1 et 2 peuvent être des séquences de commandes. Dans ce cas, l'utilisateur veillera à les entourer de parenthèses.

7.5 La commande de liaison de commandes

Elle permet également de tester la valeur de l'indicateur.

Elle s'écrit :

<commande 1> / <commande 2>

La commande 1 est exécutée. Si elle réussit, la commande 2 est exécutée et on termine l'exécution de la séquence de commandes courante. Si elle échoue, on exécute la commande suivante de la liste des commandes. Les commandes 1 et 2 peuvent être des séquences de commandes. Dans ce cas, l'utilisateur veillera à les entourer de parenthèses.

ANNEXE B

Algorithme d'évaluation du type du résultat d'une expression

```
*Evaluation(exp,eno,des,typ)
   des ⟵ 1;
   analyse(exp,eno,des,typ);
*Analyse(exp,eno,des,typ)
  Si exp est une constante entière alors type ⟵ ØN;
  Si exp est une constante caractère alors type ⟵ ØC;
  Si exp est une constante booléene alors type ⟵ ØB;
  Si exp est une constante réelle alors type ⟵ ØR;
  Si exp est un objet alors an-objet(exp,eno,des,typ);
  Si exp est un objet indicé alors an-ob-ind(exp,eno,des,typ);
  Si exp est une référence de fonction alors an-ref-fonct(exp,eno,
                                                     des,typ);
  Si exp est une référence d'énoncé alors an-ref-eno(exp,eno,des,typ);
  Si exp est un opérateur-bin-bool alors an-op-bin-bool(exp,eno,
                                                     des,typ);
  Si exp est un opérateur-un-bool alors an-op-un-bool(exp,eno,des,
                                                     typ);
  Si exp est une expression parenthétisée alors analyse (exp sans
                                  parenthèse, eno,des,typ);
  Si exp est un symbole surchargeable alors an-symb-surch(exp,eno,
                                                     des,typ);
  Si exp est une expression avec quantificateur alors an-quant(exp,
                                                     eno,des,typ);
*An-objet(exp,eno,des,typ)
  chercher dans eno la définition de type de exp;
  Si recherche concluante
    alors typ ⟵ type présent dans la définition de type;
    sinon des ⟵ 3;
    afficher "définition de type manquante",exp;
Fin-si;.
*An-ob-ind(exp,eno,des,typ)
  rloc ⟵ 1;
  an-objet(exp sans indice, eno,rloc,type)
  Si rloc ≠ 3
   alors t ⟵ type;
         obj ⟵ exp sans indice;
        pour chaque indice et tant que rloc ≠ 3
   rech :    faire chercher la définition d'expression de t;
                  si recherche concluante
                    alors si la définition trouvée est un iden-
                                            tificateur de type
                        alors aller en rech;
                      fin-si;
                  sinon rloc ⟵ 3;
                      afficher "définition d'expression
                                 manquante",t ;
                  fin-si;
```

./...

```
                    si rloc≠3 et la définition de t n'utilise
                              pas le constructeur "suite"
                       alors rloc ⟵ 3;
                             afficher "expression non indiçable",obj;
                       sinon typ ⟵ t;
                             s ⟵ paramètre du constructeur suite;
                             obj ⟵ obj + indice venant d'être
                                            analysé;
                 fin-si;
          fin-faire;
      fin-si;
   des ⟵ max(des,rloc);.
  *An-ref-fonct(exp,eno,res,type)
      créer une vile vide "file";
      rloc  ⟵ 1;
      pour chaque paramètre par et tant que rloc≠3
        faire analyse(par,eno,rloc,type);
              si rloc ≠3
                alors file ⟵file + type;
              fin-si;
        fin-faire;
      si rloc ≠3
        alors chercher une description de fonction en utilisant le
                    nom de la fonction et file;
              si recherche concluante
              alors type ⟵ type du résultat de la description
                                            trouvée;
              sinon rloc ⟵ 3;
              fin-si;
      fin-si;
      res ⟵ max(rloc,res);.
  *An-ref-eno(expr,eno,res,type)
       rloc ⟵ 1;
       recherche de l'énoncé dont l'identificateur id est présent dans
                                                          exp;
       si recherche non fructueuse
         alors res ⟵ 3;
               afficher "énoncé non défini", id;
         sinon pour chaque résultat r de id et tant que rloc ≠ 3
         faire analyse (r,enoncé id,rloc,sorte-bin);
               si rloc≠3
                  alors sorte ⟵ sorte + sorte-bin;
               fin-si;
         fin-faire;
      fin-si;
      res ⟵ max(rloc,res);.
```

./...

```
*An-op-bin-bool(expr,eno,res,type)
   rloc ← 1;
   analyse (premier opérande de expr,eno,rloc,type);
   si rloc = 3 ou type ≠ ØB
      alors afficher "le type de l'expression doit être booléen",
                          premier opérande de expr;
            res ← max(res,2);
      sinon res ← max(res,rloc);
   fin-si;
   analyse (second opérande de expr,eno,rloc,type)
   si rloc = 3 ou type ≠ ØB
      alors afficher "le type de l'expression doit être booléen",
                    second opérande de expr;
            res ← max(res,2);
      sinon res ← max(res,rloc);
   fin-si;
   type ← ØB;.

*An-op-un-bool(expr,eno,rloc,type)
   rloc ← 1;
   analyse(opérande de expr,eno,rloc,type)
   si rloc = 3 ou type ≠ ØB
      alors afficher "le type de l'expression doit être booléen",
                    opérande de expr;
            res ← max(res,2);
      sinon res ← max(res,rloc);
   fin-si;
    type ← ØB;.

*An-symb-surch(expr,enon,res,type)
     rloc ← 1
     analyse (premier opérande,eno,rloc,type)
     si rloc ≠ 3
        alors analyse (second  opérande,eno,rloc,sorte-bin)
              si rloc ≠ 3
                 alors recherche du symbole avec comme profil  sorte,
                                                     sorte-bin;
                       si recherche concluante
                          alors type ← type du résultat;
                          sinon rloc ← 3;
                                afficher "opération inconnue",
                                         symbole, sorte,sorte-bin;
                       fin-si;
              fin-si;
      fin-si;
      res ← max(rloc,res);.

*An-quant(expr,eno,res,type)
   créer une définition de type pour l'objet lié au quantificateur;
   insérer cette définition dans eno;
   rloc ← 1;
   analyse(predict-associe,eno,rloc,type);
   si rloc = 3 ou type ≠ ØB
   alors afficher "le résultat de l'expression doit être booléen",
                       prédicat-associé;
         rloc ← 2
   fin-si;
   type ← ØB;
   res ← max(res,rloc);.
```

ANNEXE C

Algorithme du vérificateur de type

```
*Vérificateur (énoncé)
   pour chaque définition formelle deffor de énoncé
       faire si deffor est une définition implicite
                alors defim(deffor, énoncé)
                sinon defexp(deffor,énoncé)
             fin-si;
       fin-faire;.
*Defimp(def,eno)
   évaluation(partie droite de def,éno,res,type);
   si res = 3 ou non(comp(type,$B))
      alors afficher "le type de l'expression doit être booléen",
                      partie droite de déf;
   fin-si;
   ob-ind-ref-eno(partie droite de def,eno);.
*Ob-ind-ref-eno(exp,eno)
   pour chaque référence d'énoncé A de exp
        faire app-énoncé (A,eno);
        fin-faire;
   pour chaque objet indicé oi de exp
        faire ob-ind(oi,eno);
        fin-faire;.
*Ob-ind(oi,eno)
 pour chaque indice i de oi
 faire évaluation (i,eno,r,typ);
       si r = 3 ou non(comp(typ,$N)
          alors afficher "le type de l'expression doit être entier",
                                                                  i;
           fin-si;
 fin-faire;.

*App-énoncé(id,en)
 si id ne correspond pas à un énoncé archivé
    alors afficher "énoncé non archivé", id;
          aller en sortie;
    fin-si;
 créer une file p,
  pour chaque paramètre formel parfor de if
       faire évaluation(parfor,id,res,typ);
             si res = 3
                alors p ⟵ p + $INDEF;
                sinon p ⟵ p + typ ;
             fin-si;
       fin-faire;
res ← 1;
positionner p sur son premier élément;
```

./...

```
pour chaque paramètre effectif pareff de id
     faire évaluation (pareff,en,res,typ);
           si res≠3
              alors si comp(p,typ)
                       alors afficher "paramètre de mauvais type",
                                                    pareff;
                    fin-si;
                    p ←— suivant de p;
                    si erreur
                       alors passer au paramètre effectif suivant;
                             si erreur
                             alors aller en sortie;
                             sinon afficher "trop de paramètres
                                           effectifs",id;
                       fin-si;
                    fin-si;
              fin-si;
     fin-faire;
 afficher "pas assez de paramètre effectif",id;
sortie :;.

*Def-exp(def,eno)
 créer une file r;
 pour chaque objet ob de la partie gauche de def
     faire évaluation(ob,eno,res,typ);
           si res = 3
              alors r ←— r + $INDEF;
              sinon r ←— r + typ;
           fin-si;
     fin-faire;
si def est une conditionnelle
   alors condit(r,def,eno);
ou si def est une définition  de suite de la forme "jusqu'à"
      alors defjqa(r,def,eno):
ou si ref est une définition de suite de la forme "pour"
      alors defpour(r,def,eno);
sinon groupe(partie droite de def,eno,r);

*Condit(r,def,eno)
 pour chaque alternative alt de def
     faire ob-ind-ref-eno(condition de alt,eno);
           évaluation(condition de alt,eno,res,typ);
           si res = 3 ou non(comp(typ,$B))
                alors afficher "l'expression doit être booléene",
                               condition de alt;
                fin-si;
           groupe(partie"alors"de alt,eno,r);
     fin-faire;
groupe(partie"sinon"de def,eno,r);.
```

```
*Defjqa(r,def,eno)
 ob-ind-ref-eno(condition de def,eno);
 évaluation(condition de def,eno,res,typ);
 si res = 3 ou non(comp(typ,ØB))
    alors afficher "le résultat de l'expression doit être booléen",
                                   condition de def;
    fin-si;
 corps-définition(corps,eno);.

*Defpour(r,def,eno)
 pour chaque borne de l'intervalle de définition de def
      faire ob-ind-ref-eno(borne,eno);
            évaluation(borne,eno,res,typ);
            si res = 3 ou non(comp(typ,ØN))
               alors afficher "le résultat de l'expression doit
                        être entier", borne;
            fin-si;
      fin-faire;
 créer une définition de type pour l'objet d'indiçage et l'insérer
 dans eno;
 corps-definition(corps-eno);
 ôter de eno la définition de type introduite;.

 *Corps définition(c,eno)
       def-exp(définition principale de c,eno);
       def-exp(partie initialisation de c,eno);.

*Groupe(exp,eno,r)
 positionner r sur son premier élément;
 pour chaque expression e presente dans exp
      faire si e est différent de donnée
               alors évaluation(e, eno,res,sor)
            si res≠3
               alors si non(comp(type,file))
                        alors afficher "incompatilbité de type pour
                                 l'expression",e;
                     fin-si;
               fin-si;
            fin-si;
            r ⟵ suivant de r;
            si erreur
               alors prendre l'expression suivante dans exp:
                     si correct
                        alors afficher "trop d'expression dans la
                                   définition",g;
                     fin-si ;
                     aller en sortie;
               fin-si;
     fin-faire;
  afficher "pas assez d'expression dans",exp;
sortie : ob-ind-ref-en(exp,eno);.
```

./...

ANNEXE D

Procédé de construction du constructeur de table de vérité

Afin de déterminer les conditions élémentaires au départ de l'arborescence du prédicat-arc, la démarche suivante peut être suivie, le premier noeud considéré étant le sommet de l'arborescence.

- si le noeud considéré représente un objet ou une constante de type booléen, la condition élémentaire est cet objet ou cette constante;
- si le noeud considéré représente une fonction ou un opérateur dont tous les arguments sont de type booléens, alors appliquer récursivement cette démarche pour chacun des arguments;
- si le noeud considéré représente une fonction ou un opérateur dont un au moins des arguments n'est pas de type booléen, la condition élémentaire est l'expression correspondant au noeud.

Toutes les conditions élémentaires étant déterminées, il reste bien évidemment à déterminer les suites de booléens associées aux conditions élémentaires.

Afin d'y arriver, chaque opérateur ou fonction dont tous les arguments sont de type booléen doit posséder une table de décision, soit créée une fois pour toutes, soit créée pour une spécification par l'utilisateur.

Exemples

- L'opérateur "et" possède la table de décision suivante, table définie une fois pour toutes.

a	1	1	0	0
b	1	0	1	0
et	1	0	0	0

- La fonction ou-exclusif définie par un spécifieur dans l'élaboration d'une spécification particulière, et ayant trois arguments, se verra associer par l'utilisateur la table de décision suivante

a	1	0	0	1	1	1	0	0
b	0	1	0	1	1	0	1	0
c	0	0	1	1	0	1	1	0
ou-exclusif	1	1	1	0	0	0	0	0

Ces tables pourraient être archivées sous forme de règles définissant les opérateurs dans les descriptions strictes.

Exemple

La fonction ou-exclusif aurait sa table de décision archivée de manière semblable au fragment suivant :

ou-exclusif(a,b,c) == vrai si a et non(b) et non(c);
ou-exclusif(a,b,c) == vrai si non(a) et b et non(c);
ou-exclusif(a,b,c) == faux si a et b et non(c)

A chaque noeud correspondant à une expression à résultat booléen de l'arborescence sera attachée une liste PREVAL reprenant les valeurs booléenes autorisées comme résultat de la sous-arborescence débutant au noeud considéré.

Le sommet de l'arborescence a comme liste PREVAL associée le singleton "vrai", seule valeur autorisée pour l'ensemble de l'arborescence.

La détermination des listes PREVAL associéesaux noeuds non-sommet de l'arborescence (listes que nous appellerons PREVAL') sera appliquée par récurrence selon les règles suivantes, où nous

./...

appellerons noeud courant, le noeud auquel correspond PREVAL, et noeuds fils,les noeuds correspondants aux arguments du noeud courant :

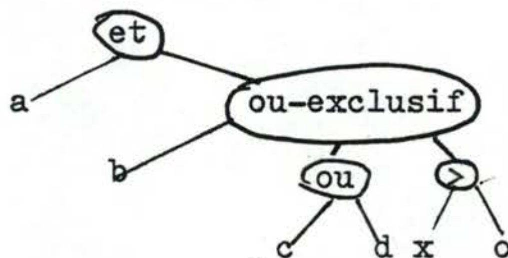
- pour chacun des éléments de la liste PREVAL, si cet élément est "vrai" (resp."faux"), on sélectionnera dans la table de décision associée à l'opérateur représenté par noeud-courant,toutes les règles fournissant la valeur "vrai" (resp."faux"), règles faisant chacunes intervenir les n arguments de l'opérateur;
- les p règles étant sélectionnées, les n files sont remplies de p valeurs, la ième valeur de la file PREVAL associée au jème noeud -fils étant la valeur "vrai" ou "faux" admise pour le jème argument dans la ième règle.

La sélection dans la table de décision des règles fournissant une valeur particulière ("vrai" ou"faux") déterminera toujours au moins une règle, à moins qu'il ne s'agisse d'un opérateur ou fonction fournissant la valeur "faux" (ou"vrai") quels que soient ses arguments, opérateurs n'ayant donc aucun intérêt.

Le processus est appliqué avec récurrence pour chacun des noeuds-fils dont le type de tous les arguments est booléen.

Exemple

Soit l'arborescence suivante, dans laquelle a,b,c et d sont des expressions à résultat booléen, et x > o représentant la comparaison de deux entiers.



./...

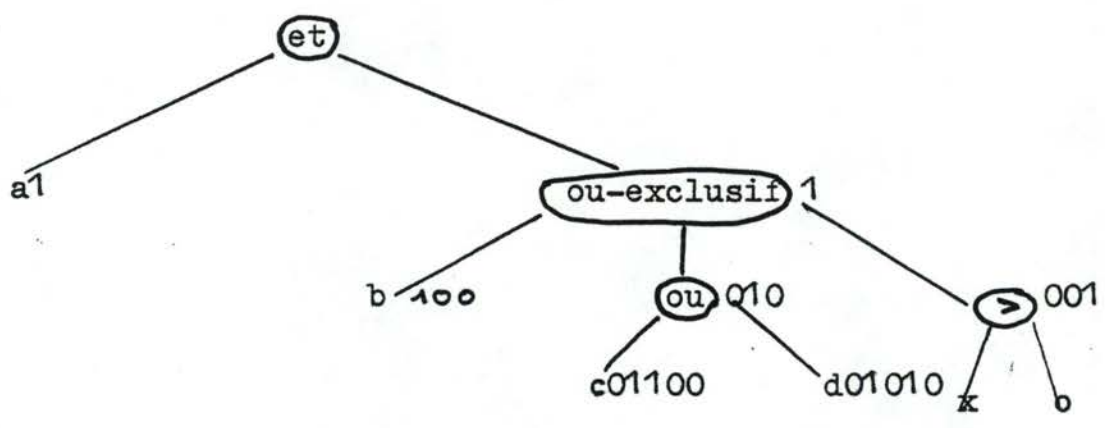
La table PREVAL correspondant au noeud "et" est le singleton "vrai", "et" étant le sommet de l'arborescence. Pour fournir la valeur "vrai" à l'opération "et", les deux arguments doivent avoir la valeur "vrai". La table PREVAL associée à "a" de même que celle associée au noeud "ou-exclusif" se réduit donc elle aussi au singleton "vrai".

Pour fournir la valeur "vrai" à l'opérateur "ou-exclusif", l'un seulement des trois arguments peut être "vrai". Les tables PREVAL associées à "b", "ou" et ">" sont donc respectivement "vrai", "faux", "faux"; "faux","vrai","faux";"faux,"faux","vrai".

La table PREVAL associée à c et d reprendra successivement toutes les combinaisons de valeur rendant le "ou" successivement faux, vrai et faux, conformément à sa table PREVAL.

La table PREVAL associée à c est dès lors la suivante : faux ,vrai, vrai, faux, faux, et celle associée à d : faux, vrai, faux,vrai, faux.

En associant les tables PREVAL à leurs noeuds, on obtient l'arborescence suivante, ou 1 est noté pour "vrai" et 0 pour "faux".



./...

Afin de pouvoir reconstituer la table de vérité, nous introduisons un séparateur dans les listes PREVAL. Ces séparateurs seront insérés entre les valeurs provenant de règles distinctes, et seront distincts d'un niveau à l'autre et y seront insérés durant la construction de la table PREVAL.

Les nouvelles tables ainsi définies auront donc la forme terminale suivante :

Nom du noeud	PREVAL associée
et	1
a	1
ou-exclusif	1
b	12020
ou	02120
c	021313020
d	021303120
>	02021

Détermination de la table de vérité

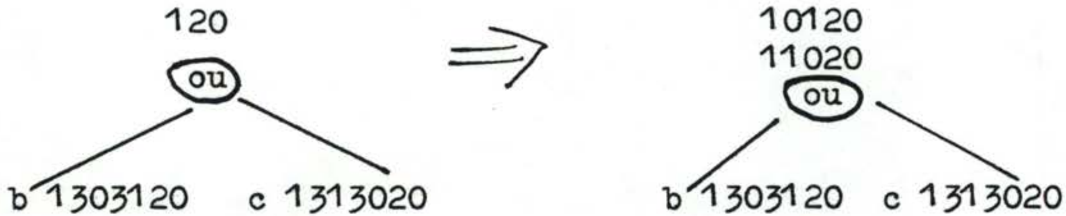
Il convient ici de rassembler les tables PREVAL afin d'obtenir la table de vérité du prédicat-arc.

Si le noeud considéré est une condition élémentaire, sa table PREVAL est inchangée.

Si les noeuds-fils sont tous les conditions élémentaires, la table PREVAL du noeud considéré est constituée des tables PREVAL' de ses noeuds-fils; nous positionnerons dans ce cas un indicateur

renc à "vrai

<u>Exemple</u>



Si tous les noeuds fils ne sont pas tous des conditions élémentaires, la première constatation à faire est que chacun des n noeud-fils possède dans sa table PREVAL'le même nombre de séparateurs émanant des règles déduites au niveau supérieur. Nous appellerons NTABLEi le nombre de valeurs comprises dans la table TABLE entre le i-1 et le ième séparateur, et généraliserons la notion de séparateur à l'indicateur de début ou de fin de ligne.

Si tous les noeuds-fils ne sont pas des conditions élémentaires, il y a récurrence pour le premier fils. Pour les fils suivants, nous distinguerons les noeuds conditions élémentaires et les autres. Dans le premier cas, si ième est positionné (c.à.d. si dans l'analyse des fils actuels, un noeud n'étant pas condition élémentaire a été rencontré), alors il faut reproduire en NPREVAL i exemplaires chaque colonne de PREVAL'située entre le i-1 et le ième séparateur, puis ajouter PREVAL' à PREVAL.

Si renc n'est pas positionné, l'ajout de PREVAL' à PREVAL représentera la table de vérité partielle des arguments déjà analysés.

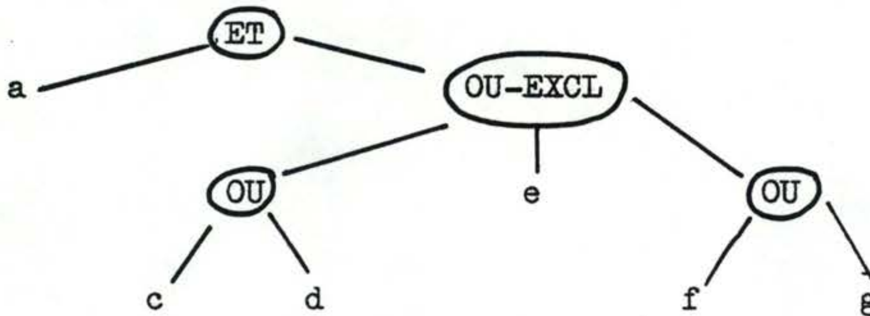
Si le noeud n'est pas une condition élémentaire, alors il faut déterminer par récurrence sa table PREVAL. Lorsque cette étape est terminée, il faut reproduire en NPREVAL'i exemplaires les colonnes de PREVAL, et en NPREVAL i les groupes de colonnes de PREVAL' entre i-1 et le ième séparateur.

./...

Avant la sortie de cette règle, et quel que soit le cas, il faut modifier PREVAL de façon à ce q'uil ne contienne plus que les séparateurs autorisés.

Exemple

Soit l'arborescence suivante:



Les tables PREVAL des différents noeuds sont les suivantes :

Nom du noeud	PREVAL
ET	1
a	1
OU-EXCL	1
OU	12020
c	131302020
d	130312020
e	02120
OU	02021
f	020213130
g	020213031

Le noeud initial est "et" dont les fils ne sont pas des conditions élémentaires.

PREVAL ← 1 (valeur du PREVAL du 1er fils)

recurrence pour ou-excl.

./...

ses fils ne sont pas tous des conditions élémentaires

récurrence pour le 1er fils ou

les deux fils sont des conditions élémentaires.

⟶ PREVAL ⟵ 131302020

130312020

après élimination des séparateurs

PREVAL ⟵ 1102020

1012020

La table PREVAL' du noeud e est 02120.

Ce noeud est une condition élémentaire, et renc étant positionné, le traitement suivant est effectué.

- duplication en NPREVALi exemplaire de chaque colonne de PREVAL' située entre le i-1 et le ième séparateur, et inversément

PREVAL devient dès lors : 0002120

PREVAL n'étant pas modifié par le dupliquage, sa nouvelle valeur est PREVAL+PREVAL' cdd

1102020

1012020

0002120

L'argument suivant n'étant pas une condition élémentaire, il y a détermination par récurrence de sa nouvelle table PREVAL. Ses deux arguments étant des conditions élémentaires, PREVAL est la juxta position des deux PREVAL' à savoir

020213130

020213031

De cette table, il faut encore retirer les séparateurs indésirables.PREVAL' de "OU" devient dès lors :

0202110

0202101.

Il faut ensuite dupliquer en NPREVAL'i exemplaires les colonnes de PREVAL, et en NPREVALi des groupes de colonnes de PREVAL'.

./...

PREVAL devient : 110202000
101202000
000212000

et PREVAL' : 000202110
000202101

La nouvelle table PREVAL associée au noeud "OU-EXCL" est donc la suivante : 110202000
101202000
000212000
000202110
000202101

La table PREVAL du noeud "OU-EXCL" devient, après suppression des séparateurs : 1100000
1010000
0001000
0000110
0000101

Cette table PREVAL redevenant PREVAL' est fusionnée avec celle existante pour le premier argument.
Le cas envisagé pour le "et" était celui où tous les fils n'étaient pas conditions élémentaires.
La table PREVAL du premier argument est le singleton 1.
Il faut dupliquer en NPREVAL'i exemplaires les colonnes de PREVAL et en NPREVALi exemplaires les groupes de colonnes de PREVAL'.
PREVAL devient 1111111, et PREVAL' reste inchangé
La fusion des deux tables fournit dès lors :

1 1 1 1 1 1 1
1 1 0 0 0 0 0
1 0 1 0 0 0 0
0 0 0 1 0 0 0
0 0 0 0 1 1 0
0 0 0 0 1 0 1

./...

Si l'algorithme de détermination des conditions élémentaires est appliqué à l'exemple, il produira, dans l'ordre, les conditions élémentaires : a,c,d,e,f,g.

La table de vérité peut dès lors être construite par association du ième résultat de cet algorithme à la ième ligne de la table PREVAL, afin d'obtenir la table de vérité suivante associée au prédicat-arc "a <u>et</u> ou-exclusif(c <u>ou</u> d,e,f <u>ou</u> g)" :

a	1	1	1	1	1	1	1
c	1	1	0	0	0	0	0
d	1	0	1	0	0	0	0
e	0	0	0	1	0	0	0
f	0	0	0	0	1	1	0
g	0	0	0	0	1	0	1

REFERENCES BIBLIOGRAPHIQUES

[BOE,83] BOEHM, B.W.

Software Engineering Economics
Prentice Hall, Englewood Cliffs, N.J. 1983

[BOU,82] BOULLIER, P

"Manuel d'utilisation et mise-en-oeuvre du système SYNTAX sous MULTICS"
Disponible dans la documentation du système MULTICS de l'INRIA
Mars 1982 (INRIA).

[DUB,81] DUBOIS, E

"Une approche déductive à la spécification et au développement d'une application de gestion en temps réel"
M.S. Thesis, Institut d'Informatique
Namur, 1981

[DUB, 82] DUBOIS, E; FINANCE, J.P.; van LAMSWEERDE, A

"Towards a deductive approach to information system specification an design".
Kyoto conference 1982.

[FIN,79] FINANCE, J.P., DERNIAME, J.C.

"Types abstraits de donnée : spécification, utilisation et réalisation"
Ecole d'été de l'AFCET - MONASTIR

[FRA,81] FRASER, C.W.

"Syntax-directed editing of general datatructures" in Proc. ACM SIGPLAN/SIGOA
Conf text manipulation (Portland,Ore.,June 1981
ACM, New York, 1981 p. 17-21

[GER,80] GERMAIN, G
Les Nouvelles Techniques de Spécification du Logiciel
Journée de Synthèse,
Congrès AFCET Informatique 1980

[GUY,80] GUYARD, J, LESCANNE, P.
"Une introduction à MENTOR"
Document disponible au Centre de Recherche en Informatique de Nancy.

[JACK,75] JACKSON, M.
"Principles of programming design"
Academic Press, London, 1975

[LAM,82] van LAMSWEERDE, A
"Les outils d'aide au développement de logiciels : un aperçu des tendances actuelles"
XVèmes journées internationales de l'informatique et de l'automatisme - Paris, 16-18juin 1982

[LIS,74] LISKOV, B.H, ZILLES, S.N.
"Programming with abstract data types";
in SIGPLAN notices 9, 4, 1974

[MED,82] MEDINA-MORA,R
"Syntax-directed editing :
Towards integrated Programming Environments"
Department of Computer Science
Carnegie-Mellon University
March, 1982

[MEL,82] MELESE, B
"METAL, un langage de spécification pour le système MENTOR"
in T.S.I. - Technique et Science Informatique
vol 1, n° 4, 1982

./...

/‾MEL2,82_7 MELESE, B.
"METAL, un META-langage pour le système MENTOR"
Documentation disponible à l'INRIA

/‾MEL3,82_7 MELESE, B
"METAL, manuel d'utilisation"
Publication prochaine, manuscript disponible à l'INRIA

/‾MEY,82_7 MEYROWITZ, N., VANDAM, A.
"Interactive editing systems"
in ACM COMPUTING SURVEYS
Vol. 14, Number 3, September 1982 p. 321-353

/‾PAI,79_7 PAIR, C
"La construction des programmes"
in RAIRO - vol 13, n°2, 1979 p.113-139

/‾QUE,83_7 QUERE, A
"Présentation du langage SPES"
Publication prochaine, manuscript disponible au Centre de Recherche Informatique de Nancy.

/‾RIN,82_7 RIN, N.A.
"An interactive application, development system and support environment"
in Automated Tools for Information System Design, H.J.Schneider et A.J. Wassermann (eds) - North-Holland, 1982

/‾SAND,78_7 SANDEWALL, E
"Programming in an interactive environment : the LISP experience"
in COMPUTING SURVEYS, vol 10, n°1, March 1978

/‾TEIT,81a_7 TEITELBAUM,T, REPS, T.
"The Cornell synthesizer :
A syntax-directed programming environment"
in COMMUNICATIONS of the ACM
Vol. 24, number 5, September 1981 p.563-573

/‾TEIT,81b_7 TEITELBAUM, T., REPS, T., HORWITZ, S.
"The why and wherefore of the Cornell program synthesizer"
in Proc ACM SIGPLAN/SIGOA conf.-text manipulation (Portland, Ore, June 1981)
ACM, New York, 1981 p. 8-16

/‾TEITE,81_7 TEITELMAN, W., MASISNTER, L.
"The programming environment"
Xerox Palo Alto Research Center
April 1981

/‾WILC,75_7 WILCOX, T., DAVIS, A., TINDALL, M.
"The design and implementation of a table driver, interactive diagnostic programming system"
in COMMUNICATIONS of the ACM
November 1976, Volume 19, number 11 p.609-611

BUMP
0 0 2 3 9 5 3 7 1
*FM B16/1983/23