

THESIS / THÈSE

MASTER EN SCIENCES INFORMATIQUES

Conception et Réalisation d'un Atelier de Développement dBase-III

Wasiak, Jan

Award date:
1989

Awarding institution:
Université 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.

**CONCEPTION ET REALISATION
D'UN
ATELIER DE DEVELOPPEMENT
DBASE-III**

Mémoire présenté par

JAN WASIAK

en vue de l'obtention du titre de
Licencié et Maître en Informatique

Promoteur : Professeur **JEAN-LUC HAINAUT**

Table des matières

Introduction.....	1
Partie I. Principes généraux de la conception d'une base de données	
I.1. Démarche de conception d'applications sur base de données.....	3
I.2. Ateliers de développement d'applications sur base de données.....	4
I.2.1. Fonctions.....	4
I.2.2. Architecture d'un atelier de développement.....	4
I.3. Principes d'un atelier de développement dBase.....	6
I.3.1. Fonctions de l'atelier.....	6
I.3.2. Architecture générale de l'atelier.....	6
I.3.3. Les modèles.....	8
I.3.3.1. Modèle de spécification de structures de données.....	8
I.3.3.2. Modèle de description d'algorithmes.....	11
I.3.4. Le schéma conceptuel de la base des spécifications.....	12
I.3.4.1. Introduction.....	12
I.3.4.2. Le schéma conceptuel de la base des spécifications.....	12
Partie II. Développement d'une base de données dBase.....	15
Partie III. Conception et réalisation d'un générateur d'applications dBase III	
III.1. Définition d'un langage procédural LDA.....	16
III.1.1. Introduction.....	16
III.1.2. Grammaire BNF d'un langage.....	16
III.1.3. Structures algorithmiques en LDA.....	17

III.1.3.1. La séquence d'instructions.....	17
III.1.3.2. Les constantes.....	17
III.1.3.3. Les variables.....	18
III.1.3.4. Les attributs d'un type d'entité.....	18
III.1.3.5. Les expressions.....	18
III.1.3.6. Les expressions d'ensemble d'entités...	19
III.1.3.7. L'assignation.....	21
III.1.3.8. L'assignation d'entité.....	21
III.1.3.9. Instruction d'affichage à l'écran.....	21
III.1.3.10. Instruction de lecture de données.....	22
III.1.3.11. La boucle énumérative.....	22
III.1.3.12. L'alternative.....	23
III.1.3.13. La boucle while.....	24
III.1.4. Synthèse de la grammaire BNF d'ADL.....	25
III.2. Description de dBase III.....	27
III.2.1. Présentation générale de dBase III.....	27
III.2.2. Les structures de données de dBase III.....	27
III.2.3. Les fichiers de dBase III.....	27
III.2.4. Un exemple de Base de données.....	28
III.2.5. Description des commandes de dBase III.....	28
III.3. Conception d'un compilateur ADL - dBase III.....	33
III.3.1. Vue générale d'un compilateur.....	33
III.3.2. Analyse lexicale.....	34
III.3.3. Analyse syntaxique.....	34
III.3.3.1. Introduction.....	34

III.3.3.2. Notion de forme de phrase et de phrase..	35
III.3.3.3. Représentation arborescente.....	36
III.3.3.4. Classification des analyseurs syntaxiques.....	37
III.3.3.5. L'analyse syntaxique "bottom-up".....	37
III.3.3.6. La table d'analyse.....	40
III.3.3.7. Construction de la table d'analyse.....	41
III.3.4. Génération de code.....	47
III.6. Règles de traduction.....	52
III.7. Réalisation.....	60
III.7.1. Introduction.....	61
III.7.2. Architecture logique de l'atelier.....	62
III.7.3. Description des modules.....	62
III.7.4. Implémentation de l'atelier.....	63
III.8. Exemples.....	65
Conclusions.....	72
Bibliographie.....	73
Annexes ¹	

¹ l'implémentation et le mode d'emploi de l'atelier se trouvent dans un appendice séparé.

Je tiens tout spécialement à remercier

Le Professeur Jean-Luc Hainaut sans qui ce sujet de mémoire ne m'aurait pas été confié. Il a su, malgré ses nombreuses occupations, rester disponible. Je le remercie pour ses enseignements, pour ses remarques et ses conseils.

Les professeurs de l'Institut d'Informatique qui nous ont prodigué leur science et qui ont oeuvré à donner toute sa signification au qualitatif "universitaire " de notre formation.

Tous ceux et celles qui ont participé de près ou de loin à la réalisation de ce mémoire.

INTRODUCTION

Dans une organisation, l'information constitue une ressource vitale au même titre que les personnes, les moyens financiers ou les équipements. Aujourd'hui les bases de données sont les supports privilégiés pour stocker ces informations. L'efficacité des comportements d'une organisation repose en partie sur la qualité des informations utilisées dans l'organisation. Il s'avère dès lors de plus en plus indispensable pour les organisations de gérer au mieux la ressource "information". Mais la gestion des informations et la conception des bases de données sont complexes.

Le concept de base de données se voit associer deux objectifs. Le premier est d'être une représentation fidèle d'un système réel et le second est de constituer un serveur de données opérationnel et efficace pour une gamme de traitements.

dBase III d'Ashton-Tate est un système de gestion de base de données disponible sur micro-ordinateurs. Il offre à l'utilisateur la possibilité de concevoir et de gérer une base de données. Il offre également des structures algorithmiques qui permettent de rédiger des programmes "base de données". Hélas, c'est un outil difficile à utiliser et son langage de programmation est sommaire. De plus ce langage de programmation prend beaucoup de code, ce qui est source d'erreurs.

D'où l'utilité de concevoir un atelier de conception d'applications dBase. L'atelier doit permettre d'aider l'utilisateur à concevoir et à utiliser une base de données dBase.

La structure de l'atelier tente de réaliser deux objectifs généraux.

L'atelier doit avant tout être simple pour permettre aux utilisateurs non-spécialisés d'écrire des programmes orientés base de données de façon simple et efficace. L'atelier traduira ce langage simple et efficace automatiquement en langage dBase. C'est la fonction de génération dBase de l'atelier.

L'atelier doit offrir un environnement complet qui permet de spécifier une base de données. C'est la fonction de documentation de l'atelier.

Ce mémoire s'articule autour de trois grandes parties, consacrées aux principes généraux de la conception d'une base de données, au développement d'une base de données dBase et à la conception et la réalisation d'un générateur d'applications dBase III.

La première partie a pour objectif de montrer comment on peut systématiser une conception de base de données afin d'obtenir une solution exécutable sous forme d'un schéma de structure de données et des programmes rédigés dans un langage de programmation. Les deux derniers chapitres de cette partie traitent sur les ateliers de développement d'applications sur base de données en général et sur les ateliers de développement d'applications dBase en particulier. Il y est montré comment il est possible d'automatiser une partie du processus de conception. Ces outils sont étudiés à partir de leurs principales fonctions, puis par leur architecture. Cette partie propose en clôture un modèle de structuration des données et un modèle de traitements.

La deuxième partie consacrée à la conception d'une base de données dBase sera développée par Monsieur Allamehzadeh Abollhasan.

La troisième partie présente la conception et la réalisation d'un générateur d'applications dBase III. Le premier chapitre de cette partie traite du langage de traitement, le langage procédural LDA. Dans le chapitre suivant on trouve une présentation du système de gestion de base de données dBase III et un aperçu des commandes dBase III. Le troisième chapitre porte sur la conception du générateur dBase. Ce chapitre est complété par une définition des règles de traduction, qui représentent le quatrième chapitre. La réalisation de l'atelier est présentée dans le cinquième chapitre. Un exemple complet est développé en clôture de cette partie.

PARTIE I

PRINCIPES GENERAUX DE LA CONCEPTION D'UNE BASE DE DONNEES

I.1. DEMARCHE DE CONCEPTION D'APPLICATIONS SUR BASE DE DONNEES

La démarche de conception de base de données se décompose en plusieurs phases : l'analyse conceptuelle, la conception logique et la conception physique. A chaque phase correspond un schéma. Ce découpage en phases est le résultat des recherches en base de données développées en [BENCI,74] et [ANSI,75].

L'analyse conceptuelle

Cette phase a pour but d'analyser les besoins des utilisateurs et de les traduire sous forme d'un schéma conceptuel. Ce schéma doit aider les concepteurs de la base de données à exprimer la sémantique des données, et les aider à comprendre cette sémantique en vue d'organiser un stockage adéquat des données et de créer des procédures d'accès efficaces ainsi que des programmes d'applications corrects. [BOD-PIG,83/88] consacre une étude très complète à l'analyse conceptuelle.

La conception logique

La phase de conception logique va produire un schéma logique qui reprend toute la sémantique exprimée dans le schéma conceptuel, et qui en outre décrit les besoins des applications en termes d'accès logiques tout en étant conforme à un SGBD. Dans [HAINAUT,86] on trouvera une démarche générale s'appliquant à différents SGBD. Cette phase construit des procédures LDA à partir des spécifications conceptuelles des traitements.

La conception physique

Cette phase vise à produire une base de données opérationnelle efficace. Cette phase réalise la traduction des procédures LDA de la phase logique en des procédures dBase III exécutables.

I.2. ATELIERS DE DEVELOPPEMENT D'APPLICATIONS SUR BASE DE DONNEES

I.2.1. Fonctions

La fonction première d'un atelier de conception est de gérer la documentation concernant une base de données en projet. Cette fonction se décompose en sous-fonctions de saisie, consultation et modification de spécifications, ainsi que de production d'une documentation sur papier. L'atelier peut aussi assurer les fonctions de transformation, de validation de spécifications ainsi que de production de descriptions destinées à un SGBD ou à destination d'un dictionnaire de données.

I.2.2. Architecture d'un atelier de développement

Un atelier de développement est construit autour d'une base de données qui contient la description de bases de données en cours de conception. Cette base de données porte le nom de base des spécifications.

Les différentes fonctions décrites ci-dessus sont prises en charge par des processeurs indépendants les uns des autres et opérant sur le contenu de la base des spécifications. La base des spécifications est organisée de manière à représenter des schémas Entité-Association. L'atelier construit sur base de cette architecture a été inspiré de [HAINAUT,86],[CADELLI,87]. Les principaux processeurs sont :

Monitoring et présentation : ce processeur contrôle la présentation sur écran ainsi que le dialogue avec l'utilisateur.

Navigateur : assure l'accès aux informations de la base des spécifications en fonction des demandes de l'utilisateur (accès par nom, par liste, par pointage, etc).

Chargeur de spécifications externes : introduit dans la base des spécifications des informations issues de sources extérieures.

Processeur de modification : réalise les modifications dans la base des spécifications en garantissant la cohérence.

Vérificateur de conformité : analyse les spécifications par rapport à un jeu de contraintes prédéfini.

Générateur de documentation : produit divers rapports.

Processeur de saisie : réalise la saisie des principaux objets de la base de données.

Générateur de descriptions exécutables : produit le texte dans un langage d'un SGBD.

I.3. PRINCIPES D'UN ATELIER DE DEVELOPPEMENT dBASE

I.3.1. Fonctions de l'atelier

On distingue quatre fonctions essentielles : la fonction de documentation, la fonction de validation, la fonction de génération dBase et la fonction de génération de rapport dBase.

La fonction de documentation

La fonction de documentation permet l'introduction et la gestion des spécifications d'une base de données en projet. Cette fonction permet aussi la production d'un rapport contenant les principaux objets de la base de données utiles à l'utilisateur.

La fonction de validation et de génération dBase

Un générateur dBase III permet de traduire des programmes base de données écrits dans un langage procédural nommé LDA (Langage de Description d'Algorithmes, fichiers avec une extension .ADL) en dBase III (fichiers avec une extension .PRG).

En traduisant, le générateur vérifie la sémantique et la syntaxe des textes rédigés en ADL. En cas d'erreur le compilateur génère un fichier de diagnostics.

Pour ce faire la compilation d'un programme d'application nécessite la connaissance des principaux objets de la base de données. C'est pour cette raison que le générateur doit avoir accès aux informations stockées dans la base des spécifications.

La fonction de génération de rapport dBase

Cette fonction permet la production de rapports contenant la structure dBase de la base de données.

I.3.2. Architecture générale de l'atelier

L'architecture de l'atelier est représentée par la figure de la page suivante. Les différentes fonctions décrites ci-dessus sont prises en charge par des modules opérant sur le contenu de la base des spécifications. Les principaux modules sont :

Le coordonnateur : ce module contrôle la présentation des informations sur écran ainsi que le dialogue avec l'utilisateur. Il coordonne l'activation des différents modules.

Le chargeur de spécifications : ce module permet d'introduire la base des spécifications.

Le générateur de documentation : ce module produit des rapports de la base des spécifications.

Le générateur de dBase III : produit le texte dBase III ou un fichier de diagnostics en cas d'erreurs.

Le générateur de rapport dBase : produit des rapports contenant la structure dBase d'une base de données.

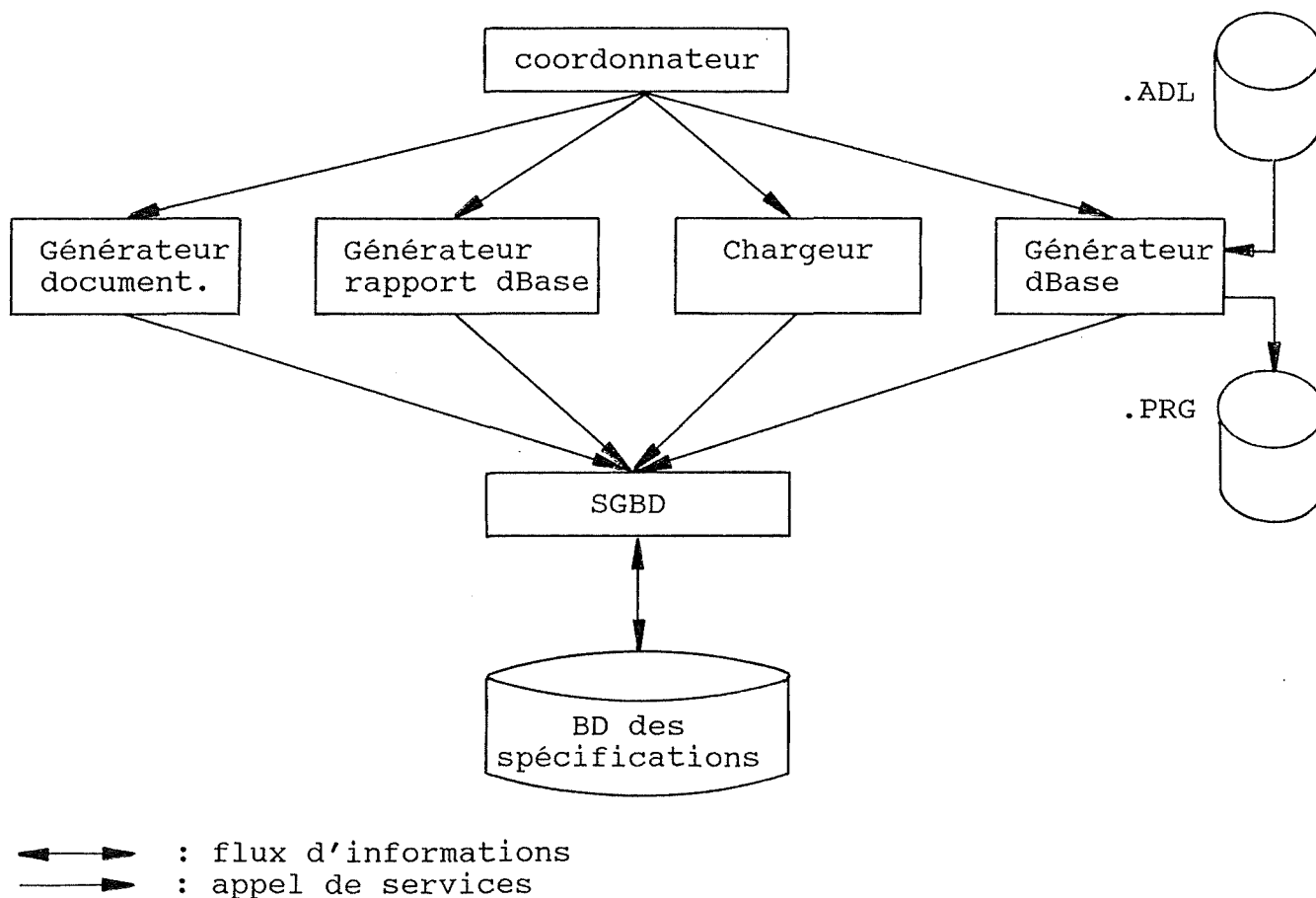


Figure 1.1. : Architecture générale de l'atelier de développement dBase.

I.3.3. Les modèles

Dans ce chapitre nous proposons un modèle et un outil destiné à produire une description conceptuelle d'une base de données. Le modèle le plus populaire à l'heure actuelle est le modèle Entité-Association. L'existence de représentations graphiques rend le modèle particulièrement attractif comme support de communication. Les principales sources en cette matière sont [TARDIEU,83], [BOD-PIG,83/88] et [CHEN,76].

En ce qui concerne les traitements nous serons amenés à spécifier le modèle de description d'algorithmes. Ce chapitre est consacré à l'étude de ces modèles.

I.3.3.1. Modèle de spécification de structures de données

Le modèle Entité/Association est un modèle qui permet de structurer les informations à l'aide des concepts d'entité, d'attribut, d'association et d'identifiant.

ENTITE et TYPE D'ENTITE

La notion fondamentale est celle d'entité. Une entité est une chose qui est considérée, dans un domaine d'application, comme un tout ayant une existence autonome. Les entités sont partitionnées en classes appelées **type d'entité**.

Ainsi, dans un domaine d'application dans lequel on repère des clients, des commandes, des lignes de commandes et des produits, on percevra des **types d'entité** que l'on nommera CLIENT, COMMANDE, LIGNECOM et PRODUIT. Comme indiqué dans la figure ci-dessous, chaque type d'entité est représenté graphiquement par un rectangle qui comporte un cartouche où figure le nom du type d'entité.

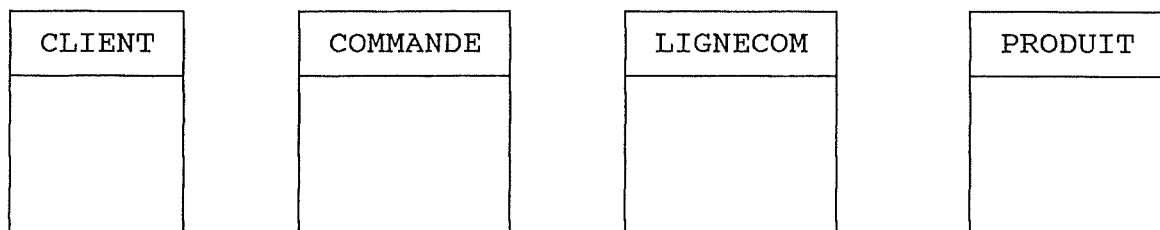


Figure 1.2. : Représentation de quatre types d'entité.

ATTRIBUTS D'UN TYPE D'ENTITE

Les propriétés d'un type d'entité sont représentées par des **attributs**. On parle d'attributs d'un type d'entité. Un attribut ne peut contenir qu'une seule valeur par entité (attribut simple).

Chaque client étant caractérisé par un numéro, un nom, une adresse et une localité, on dira que le type d'entité CLIENT est doté des attributs NCLI, NOM, ADRESSE et LOCALITE. De même, une commande étant caractérisée par un numéro de commande et une date, on dotera COMMANDE des attributs NCOM et DATE.

Le nom d'un attribut sera inscrit dans la partie inférieure du rectangle qui représente le type d'entité. La figure ci-dessous montre comment sont représentés les attributs des types d'entité retenus jusqu'ici.

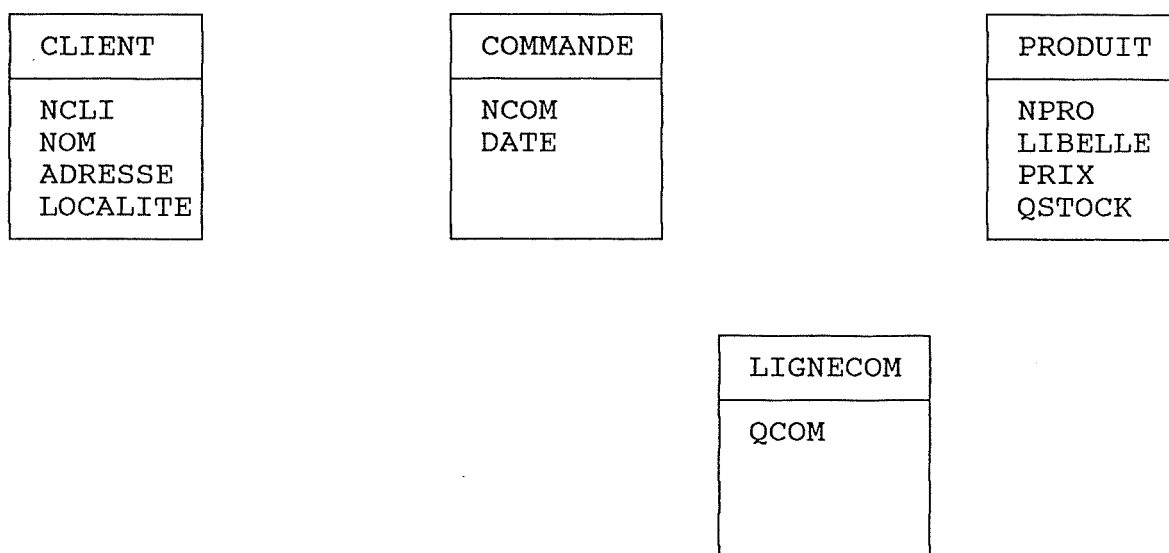


Figure 1.3. : Représentation de quatre types d'entité et de leurs attributs.

ASSOCIATION et TYPE D'ASSOCIATION

On admet que les entités sont en association les unes avec les autres. Une association est définie comme un groupe de deux entités, chacune d'elles jouant un rôle défini dans ce groupe. Les associations sont classées en **type d'association**.

Un client passe une commande, il existe donc une association entre cette commande et ce client. On dira qu'il existe un type d'association (appelé CC par exemple) entre les types d'entité CLIENT et COMMANDE. On définira également un type d'association entre COMMANDE et LIGNECOM, ainsi qu'un type d'association entre PRODUIT et LIGNECOM.

Un type d'association est représenté par un hexagone relié par des segments de droite aux rectangles qui représentent les types d'entité sur lesquels est défini le type d'association. Dans l'hexagone, on indique le nom du type d'association.

ROLE D'UN TYPE D'ASSOCIATION

Toutes les entités jouant un rôle déterminé dans les associations d'un type appartiennent au même type d'entité. Afin de préciser le nombre d'associations dans lesquelles une entité peut apparaître, on attachera à chaque rôle une contrainte de connectivité. La connectivité d'un type d'association est un couple d'entiers $i - j$, où :

- i indique le nombre minimum de fois que, à tout moment, toute entité doit assumer le rôle.
 $i \geq 0$
- j indique le nombre maximum de fois que, à tout moment, toute entité doit assumer le rôle.
 $j \geq 0$
 $j \geq i$

Considérons l'exemple représenté par la figure 1.4. L'interprétation des connectivités associées aux rôles est la suivante :

- un client peut passer un nombre quelconque de commandes,
 $j = N$
- un client peut exister sans avoir passé de commandes,
 $i = 0$
- une commande est passée par un et un seul client.
 $i = j = 1$

Dans ce document on n'admet que les types d'association dotés d'au moins un rôle de connectivité $0 - 1$ ou $1 - 1$.

Les types d'association et leurs rôles sont ajoutés au schéma sur la figure suivante.

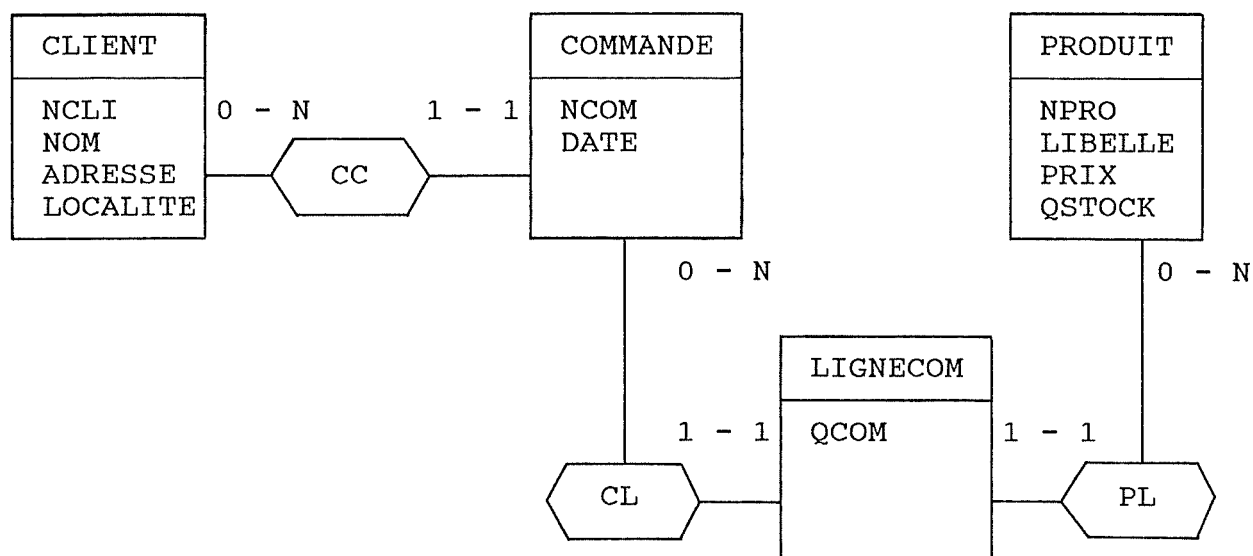


Figure 1.4. : Représentation des types d'entité, d'attributs, et des types d'association.

Tous les exemples qui seront donnés pour illustrer les concepts théoriques décrits dans la suite de ce document, seront essentiellement basés sur ce schéma.

IDENTIFIANT

Un identifiant d'un type d'entité est un attribut qui identifie les entités de ce type. CLIENT, par exemple possède un attribut NCLI tel qu'à chaque entité CLIENT correspond une valeur distincte. On dira que NCLI est l'identifiant de CLIENT.

I.3.3.2 Modèle de description d'algorithmes

Ce modèle est constitué d'un pseudo-langage (appelé LDA) permettant d'exprimer des algorithmes d'accès aux données d'une base. C'est un modèle descriptif d'algorithmes. Nous reparlerons de ce modèle plus en détail en III.1..

I.3.4. Le schéma conceptuel de la base des spécifications

I.3.4.1. Introduction

La base des spécifications contient la description de la base de données en projet. Le générateur dBase y a accès, afin de contrôler la validité des textes rédigés en LDA. La description de la base des spécifications sera exprimée selon le modèle Entité-Association.

I.3.4.2. Schéma conceptuel de la base des spécifications

Le schéma conceptuel d'une base de données a comme caractéristique son nom. Ainsi, on dotera le type d'entité SCHEMA de l'attribut NOM_S.

Une procédure caractérisée par son nom appartient à un schéma. On définira le type d'entité PROCEDURE avec comme attribut NOM_P.

Une entité collectionnée dans un schéma est caractérisée par un nom et une description. On nommera NOM_E et DESCRIPTION_E les attributs du type d'entité ENTITE.

Une entité est caractérisée par un ou plusieurs attributs. Un attribut a un nom, une description, un type et un identifiant (indiquant si l'attribut est identifiant) comme propriétés. On définira NOM_A, DESCRIPTION, TYPE et ID les attributs du type d'entité ATTRIBUT.

Une association du schéma conceptuel, caractérisée par les attributs NOM_A et DESCRIPTION_A, est définie entre deux entités où chacune assume un rôle donné. Le type d'entité ROLE a comme attributs NOM_R, ORIGINE et CIBLE.

Il existe un type d'association appelé CARACTERISTIQUE entre les types d'entité ENTITE et ATTRIBUT.

On définira également un type d'association COLLECTION_E entre SCHEMA et ENTITE, un type d'association APPARTENANCE entre PROCEDURE et SCHEMA, un type d'association COLLECTION_A entre SCHEMA et ASSOCIATION, un type d'association ASSUME entre ENTITE et ROLE et enfin un type d'association LIEN entre ROLE et ASSOCIATION.

Nous imposons à ce schéma les connectivités suivantes :

- une entité est collectionnée dans un et un seul schéma, connectivité : 1 - 1,

- un schéma peut collectionner un nombre quelconque de d'entités, connectivité : 1 - N,

- une association est collectionnée dans un et un seul schéma, connectivité : 1 - 1,

- un schéma ne peut collectionner aucune association, et peut collectionner un nombre quelconque d'associations (connectivité : 0 - N),

- une entité est caractérisée par au moins un attribut (connectivité : 1 - N) et un attribut caractérise une et une seule entité (connectivité : 1 - 1),

- un rôle est assumé par une et une seule entité (connectivité : 1 - 1) au sein d'une et une seule association (connectivité : 1 - 1),

- une entité ne peut assumer aucun rôle, et peut assumer un nombre quelconque de rôles (connectivité : 0 - N).

- une procédure appartient à un et un seul schéma (connectivité : 1 - 1).

- un schéma ne peut être utilisé dans aucune procédure, et peut être utilisé dans un nombre quelconque de procédures (connectivité : 0 - N).

Le schéma de la base des spécifications est représenté par la figure donnée à la page suivante.

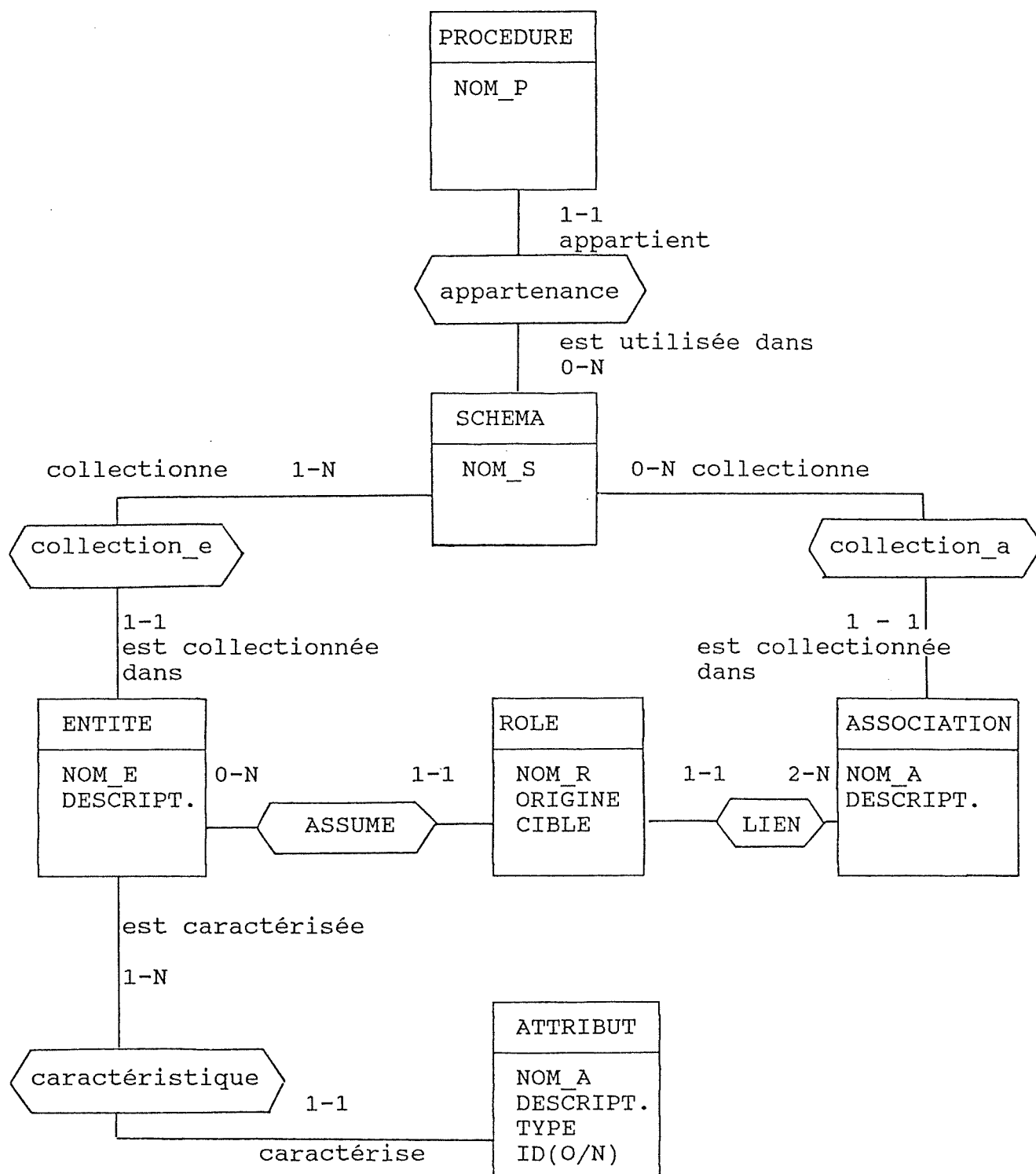


Figure 1.5. : Représentation du schéma conceptuel de la base des spécifications.

PARTIE II

DEVELOPPEMENT D'UNE BASE DE DONNEES DBASE

II. DEVELOPPEMENT D'UNE DASE DE DONNEES dBASE

La construction d'une base de données opérationnelle exige que le schéma Entité-Association soit conforme au SGBD. Il faut donc transformer le schéma Entité-Association initial en un schéma conforme.

Un schéma Entité-Association est dit conforme à un SGBD si à chaque instruction élémentaire de ce schéma correspond une construction élémentaire et une seule dans le modèle de données du SGBD.

Par exemple, un schéma qui vérifierait les contraintes suivantes serait reconnu conforme au SGBD dBase III.

- pas de type d'association,
- tout type d'entité est doté d'au moins un attribut obligatoire,
- un attribut ne peut prendre qu'une seule valeur.

L'étude de ces transformations est développée par Monsieur Allamehzadeh Abollhasan dans un mémoire parallèle.

PARTIE III

CONCEPTION ET REALISATION D'UN GENERATEUR D'APPLICATIONS DBASE III

III.1. DEFINITION DU LANGAGE PROCEDURAL LDA

III.1.1. Introduction

LDA (Langage de Description d'Algorithmes) réalise l'intégration de primitives d'accès aux données d'une base de données dans des structures algorithmiques traditionnelles. Basé sur le schéma Entité-Association, il permet d'exprimer tous les types d'accès à une base de données (accès par clé, accès séquentiel). La fonction principale d'une instruction LDA est de spécifier des actions portant sur chaque élément d'un ensemble d'entités.

III.1.2. Grammaire BNF d'un langage

Les règles de grammaire peuvent être décrites avec des grammaires BNF. L'intérêt d'utiliser une telle grammaire est de pouvoir donner une spécification précise et facile à comprendre de la syntaxe d'un langage de programmation, de pouvoir facilement adapter une grammaire au fur et à mesure de l'évolution d'un langage.

[AHO & ULLMAN,86] définissent une grammaire BNF comme un ensemble constitué de symboles terminaux, de classes syntaxiques, d'une variable distinguée et de productions où :

- les symboles terminaux sont les symboles de base à partir desquels les chaînes sont formées. Ce sont tous les symboles qui peuvent intervenir dans la représentation du texte du programme; ils sont considérés syntaxiquement comme atomiques.

- les classes syntaxiques désignent une concaténation d'autres classes syntaxiques et de symboles terminaux. Parmi celles-ci, une classe syntaxique particulière est appelée la "variable distinguée".

- la variable distinguée est la classe syntaxique associée au texte complet du programme.

- les productions d'une grammaire sont des combinaisons de symboles terminaux et de classes syntaxiques pour former des chaînes.

Une production a la forme :

classe syntaxique ::= chaîne de classes syntaxiques et/ou
de symboles terminaux.

Conventions :

Les symboles terminaux sont imprimés en gras.

Les classes syntaxiques apparaissent entre '<' et '>'.

':=' lie les parties gauche et droite d'une règle de production.

'|' indique deux choix possibles.

Tout ce qui apparaît entre "[" et "]" est optionnel.

"{" et "}" dénotent une répétition possible des caractères apparaissant entre ces deux symboles.

En général, la variable distinguée est le membre gauche de la première production.

III.1.3. Structures algorithmiques en LDA

III.1.3.1. La séquence d'instructions

Une séquence est une suite éventuellement vide d'instructions. Chaque instruction d'une séquence se termine par ';'. Le programme se terminera par '.'.

Règles BNF :

<séquence> ::= <instruction>. |
 <seq> <instruction>.

<seq> ::= <instruction>; |
 <seq> <instruction>;

III.1.2.2. Les constantes

Une constante est un entier ou bien une chaîne de caractères entre apostrophes.

Règles BNF :

<constante> ::= entier |
 <string>
<string> ::= ' caractère {caractère} '

Example :

'Dupont' est une constante qui désigne une valeur du type chaîne de caractères.

III.1.3.3. Les variables

Un type d'entité peut être désigné par une variable (ou identificateur). Les variables sont donc étendues aux variables d'entité. La valeur d'une telle variable est la référence d'un type d'entité.

Règle BNF :

```
<identificateur> ::= caractère {caractère | chiffre}
```

Exemple :

- CLI désigne l'entité CLIENT que référence la variable CLI.

III.1.3.4. Les attributs d'un type d'entité

Règle BNF :

$$\langle \text{attribut} \rangle ::= \langle \text{nom-attribut} \rangle (: \langle \text{variable-entité} \rangle)$$

où - <nom-attribut> est le nom d'un attribut du type d'entité que référence la variable d'entité <variable-entité>.

Interprétation

Cette expression désigne la valeur de l'attribut <nom-attribut> associé au type d'entité que référence la variable d'entité <variable-entité>.

III.1.3.5. Les expressions

Une expression peut être une expression arithmétique, une variable, un attribut d'un type d'entité, une constante ou un entier.

Règles BNF :

$$\langle \text{expression} \rangle ::= \langle \text{expression} \rangle \langle \text{opérateur-d'addition} \rangle \langle \text{terme} \rangle \mid \langle \text{terme} \rangle$$

```

<opérateur-d'addition> ::= + |
                        -

<terme> ::= <facteur> |
          <terme> <opérateur-multiplicateur> <facteur>

<opérateur-multiplicateur> ::= * |
                              /

<facteur> ::= identificateur |
              <constante> |
              <attribut> |
              ( <expression> )

```

Les expressions sont étendues aux expressions d'ensemble d'entités.

III.1.3.6. Les expressions d'ensemble d'entités

La plupart des instructions LDA spécifient des actions sur chaque élément d'un ensemble d'entités. Les entités de l'ensemble sont toutes du même type. L'ensemble peut contenir toutes les entités du type ou peut en être un sous-ensemble.

L'ensemble de toutes les entités est désigné par le nom du type d'entité. Par exemple, CLIENT désigne l'ensemble des entités du type d'entité CLIENT.

Un sous-ensemble d'entités est construit en spécifiant la condition de sélection que les entités du sous-ensemble doivent vérifier.

La désignation d'un ensemble est constituée d'un type d'entité, suivi éventuellement de l'expression d'une condition de sélection (notée <CSEL>). La forme BNF est la suivante :

```

<expression-entité> ::= <type-entité> [<CSEL>]

```

L'exemple suivant désigne les entités CLIENT dont la valeur de NOM est égale à 'Dupont' :

```

CLIENT (: NOM = 'Dupont' )

```

De manière à simplifier les expressions des conditions de sélection les plus courantes, nous userons de deux formes particulières, la condition d'association et la condition d'attribut.

Règles BNF :

`<CSEL> ::= <condition-att> |
 <condition-ass>`

La condition d'attribut spécifie les valeurs des attributs des types d'entité sélectionnés. La forme BNF d'une condition d'attribut est la suivante :

`<condition-att> ::= (:<nom_attribut> <oper-rel> <expression>)`

- où - `<nom_attribut>` est le nom d'un attribut d'un type d'entité.
 - `<oper-rel>` est un opérateur relationnel.
 - `<expression>` désigne une expression contenant une valeur.

La condition d'association spécifie les entités qui doivent être associées à une certaine entité. La forme BNF d'une condition d'association est la suivante :

`<condition-ass> ::= (<type-association> : <variable-entité>)`

- où - `<type-association>` est le nom d'un type d'association entre les types NOM1 et NOM2.
 - `<variable-entité>` est une variable que référence un type d'entité de type NOM2.
 - NOM1 est le nom du type des éléments évalués.

Interprétation de cette expression

Un élément du type NOM1 est retenu dans l'ensemble s'il est associé par le type d'association `<nom association>` à au moins une entité de type NOM2 désignée par `<variable-entité>`.

Un ensemble d'entités peut être vide, si aucune entité ne vérifie la condition de sélection.

Quelques exemples :

PRODUIT(PL : LIGNE) est vrai s'il existe au moins un PRODUIT associé à au moins une LIGNE;

CLIENT(:NOM = 'Dupont') est vrai s'il existe une entité CLIENT dont le NOM est égal à 'Dupont';

Une condition valide sera notée ci-dessous par `<condition>`.

III.1.3.7. L'assignation

L'instruction d'assignation est utilisée pour affecter une certaine valeur à une variable. Une assignation consiste en un identificateur de variable, suivi de l'opérateur d'assignation := , suivi d'une expression. L'expression et la variable doivent être du même type. La forme BNF d'une instruction d'assignation est la suivante :

`<assignation> ::= identificateur := <expression>`

Cette instruction est étendue aux variables de types d'entité.

III.1.3.8. L'assignation d'entité

L'assignation d'entité est une instruction d'assignation d'un ensemble d'entités à une variable d'entité. En général l'ensemble d'entités ne contient qu'une seule entité. S'il y en a plusieurs, une seulement est sélectionnée. Si l'ensemble est vide la variable d'entité est laissée inchangée.

Règle BNF de l'assignation d'entité :

`<assignation-entité> ::=`
`<variable-entité> := <expression-entité>`

Quelques exemples :

`CLI := CLIENT (: NCLI = X )`

Une occurrence du type d'entité CLIENT dont le numéro NCLI a une valeur égale à X est assignée à la variable CLI.

`COM := COMMANDE ( CC : CLI )`

Une occurrence du type d'entité COMMANDE associée par CC à une occurrence du type d'entité CLIENT désignée par CLI est assignée à la variable COM.

III.1.3.9. Instruction d'affichage à l'écran

Formes BNF :

`<write> ::= WRITE(<expression>)`

et

`<writeln> ::= WRITELN(<expression>)`

La première instruction affiche le contenu de <expression> à la ligne suivante, tandis que la seconde instruction affiche le contenu de <expression> à la suite du dernier caractère affiché.

III.1.3.10. Instruction de lecture de données

Forme BNF :

```
<read> ::= READ(<chaîne>,<var>)
```

Cette instruction affiche à l'écran la chaîne de caractères <chaîne> et attend l'introduction d'une donnée au clavier pour ranger celle-ci dans la variable <var>.

III.1.3.11. La boucle énumérative

La boucle énumérative permet au programmeur de spécifier des actions portant sur chaque élément d'un ensemble d'entités. L'élément courant de l'ensemble, c'est-à-dire la variable de la boucle, est désignée par une variable d'entité. La forme BNF d'une boucle énumérative est la suivante :

```
<for> ::= for  <variable-entité> := <expression-entité> do
                <séquence>
            endfor
```

où - <expression-entité> est l'expression d'un ensemble d'entités;
 - <variable-entité> est le nom d'une variable d'entité du même type que l'ensemble d'entité <expression-entité>.

Interprétation

Chaque élément de l'ensemble <expression-entité> est assigné à <variable-entité> et <séquence> est exécutée.

Quelques exemples :

Exemple 1

```
for cli:=client do
    writeln(NOM(:CLI));
endfor
```

Cette instruction affiche à l'écran les noms de tous les CLIENTS de la base de données.

Exemple 2

```
for cli:=client do
    for com:=commande(cc:cli) do
        writeln(NOM(:CLI));
    endfor;
endfor
```

Ces instructions affichent à l'écran les noms de tous les CLIENTS de la base de données associés à une COMMANDE.

Exemple 3

```
for CLI:=CLIENT(:LOCALITE = 'Namur') do
    writeln(NOM(:CLI));
endfor
```

Cette instruction affiche à l'écran tous les CLIENTS de Namur.

III.1.3.12. L'alternative

Formes BNF :

```
<if> ::= if <condition> then <séquence> endif
et
```

```
<if> ::= If <condition> then <séquence> else <séquence> endif
```

```

<condition> ::= <condition-n> |
               <condition-LDA>

<condition-n> ::= <expression> <oper-rel> <expression>

<oper-rel> ::= = | <> | < | <= | >= | >

<condition-LDA> ::= <ens>(<type-association> : <ens>) |
                   <ens> <condition-att>

<ens> ::= <type-entité> |
          <variable-entité>

```

Exemple :

```

if CLIENT(CC:COM) then
    writeln(NCLI(:COM));
endif

```

S'il y a au moins une occurrence du type d'entité CLIENT associée par CC à la variable COM, désignant une occurrence de l'entité COMMANDE que référence cette variable alors les valeurs de NCLI associées à la variable COM sont affichées à l'écran.

III.1.3.13. La boucle while

Forme BNF :

```

<while> ::= while <condition-n> do
           <séquence>
           enddo

```

où <condition-n> est une condition traditionnelle définie au point précédent.

Interprétation

Tant que la condition <condition> est évaluée à vrai, la séquence <séquence> est exécutée.

III.1.4. Synthèse de la Grammaire BNF d'ADL

```

<séquence> ::= <instruction> . |
               <seq> <instruction> .

<seq> ::= <instruction> ; |
           <seq> <instruction> ;

<instruction> ::= <write> |
                  <writeln> |
                  <read> |
                  <assignation> |
                  <if> |
                  <while> |
                  <for>

<write> ::= write ( <expression> )

<writeln> ::= writeln ( <expression> )

<constante> ::= <string> |
                entier

<string> ::= ' caractère { caractère } ' |
            nil

<read> ::= read ( <string> , identificateur )

<assignation> ::= identificateur := <expression>

<expression> ::= <expression> <opérateur-d'addition> <terme> |
                 <terme>

<opérateur-d'addition> ::= + |
                        -

<terme> ::= <facteur> |
            <terme> <opérateur-multiplicateur> <facteur>

<opérateur-multiplicateur> ::= * |
                             /

<facteur> ::= identificateur |
              <constante> |
              <attribut> |
              ( <expression> )

<attribut> ::= <nom-attribut> (: <variable-entité> )

<nom-attribut> ::= identificateur

```

```

<variable-entité> ::= identificateur

<if> ::= if <condition> then <séquence> endif |
        if <condition> then <séquence> else <séquence> endif

<condition> ::= <condition-n> |
                <condition-LDA>

<condition-n> ::= <expression> <oper-rel> <expression>

<oper-rel> ::= = | <> | < | <= | >= | >

<while> ::= while <condition-n> do <séquence> endwhile

<CSEL> ::= <condition-ass> |
            <condition-att>

<condition-ass> ::= ( <type-association> : <variable-entité> )

<condition-att> ::= (: <nom-attribut> <oper-rel> <expression> )

<condition-LDA> ::= <ens> <condition-att> |
                    <ens> ( <type-association> : <ens> )

<ens> ::= <type-entité> |
            <variable-entité>

<type-association> ::= identificateur

<type-entité> ::= identificateur

<for> ::= for <variable-entité> := <expression-entité> do
            <séquence> endfor

<expression-entité> ::= <type-entité> |
                        <type-entité> <CSEL>

<assignation-entité> ::= <variable-entité> := <expression-entité>

```

III.2. DESCRIPTION DE dBASE III

III.2.1. Présentation générale de dBase III

Le logiciel dBase III d'Ashton-Tate est la troisième version du logiciel de gestion de données dBase. dBase III est disponible sous MS-DOS et PC-DOS sur micro-ordinateurs PC et PC/AT d'IBM, ainsi que sur les compatibles. Ce logiciel exige un minimum de 256 K de mémoire centrale et peut fonctionner sur une machine qui n'est dotée que d'unités à disquettes.

dBase III offre à l'utilisateur un langage de commande interactif qui permet de définir un fichier, de modifier sa description, d'ajouter, de supprimer et de modifier des enregistrements, d'extraire des données d'un fichier, de consulter des enregistrements sélectionnés.

Il offre également des structures algorithmiques qui ajoutées au langage de commande permettent de rédiger des procédures et des programmes complexes. Ces derniers sont interprétés, ce qui n'assure pas des performances exceptionnelles en traitement de calcul. Ce n'est d'ailleurs pas le but de ce logiciel.

Outre les fonctions de gestion de fichiers et l'interpréteur, dBase III possède un gestionnaire d'écran, un générateur d'écran, un générateur de rapports, un générateur d'étiquettes et un éditeur de texte (sommaire) [HAINAUT,86].

III.2.2. Les structures de données de dBase III

Le fichier de données dBase III est constitué d'enregistrements (record) décomposés en champs (field). Un enregistrement est identifié par un numéro (Record#) qui correspond à sa position dans le fichier.

III.2.3. Les fichiers de dBase III

dBase utilise un certain nombre de types de fichier caractérisés chacun par une extension MS-DOS spécifique. Un fichier de données est caractérisé par l'extension .DBF. Ainsi le fichier CLIENT porte le nom CLIENT.DBF en MS-DOS.

Un fichier qui contient un programme ou des procédures porte l'extension .PRG.

III.2.4. Un exemple de base de données

Nous donnons le format et le contenu initial des fichiers qui servent d'exemple dans la suite de ce document en annexe 1.

III.2.5. Description des commandes

Ce chapitre donne une brève description des commandes dBase III. Une description plus complète est fournie dans le manuel accompagnant le logiciel [SIMPSON,87].

III.2.5.1. Séquence de commandes dBase III

Règles BNF :

```
<sequence> ::= <commande> | {<commande>}
<commande> ::= <IF> | <WHILE> | <ACCEPT> | <?> | <??> |
               <LOCATE> | <ASSIGNATION> | <SELECT>
```

III.2.5.2. Variable dBase

Règle BNF :

```
<var> ::= caractère {caractère}
```

III.2.5.3. L'assignation

Règle BNF :

```
<var> = <expr>
```

III.2.5.4. Expressions dBase

Règles BNF :

```
<expr> ::= <expr> + <terme> |
           <expr> - <terme> |
           <terme>

<terme> ::= <terme> * <facteur> |
           <terme> / <facteur> |
           <facteur>

<facteur> ::= ( <expr> ) |
             <var> |
             <alias> -> identificateur |
             .not. <facteur>
```

III.2.5.5. Condition dBase

Règles BNF :

`<condition> ::= <expr> <oper-rel> <expr>`

`<oper-rel> ::= = | < | > | <= | >= | <>`

III.2.5.6. Ouverture d'un fichier

Pour ouvrir un fichier, il faut introduire les commandes suivantes :

```
select entier
use fichier [ alias alias ]
```

- où - "entier" représente une constante entière;
 - "fichier" représente un nom de fichier dBase;
 - "select entier" rend courant l'une des 10 zones de travail. Dans chaque zone de travail, il est possible d'ouvrir (use) un fichier de données. Une zone de travail est désignée par le numéro du fichier qui y est ouvert. Le surnom de la zone de travail est soit le fichier, soit alias si celui-ci est spécifié.

III.2.5.7. Fermeture d'un fichier

La commande `close databases` ferme tous les fichiers de la base de données.

III.2.5.8. Affichage à l'écran

Il y a deux commandes pour afficher une expression à l'écran.

Formes BNF :

`<?> ::= ? <expr>`

et

`<??> ::= ?? <expr>`

La première commande affiche la valeur de `<expr>` à la ligne suivante. La seconde affiche la valeur de `expr` à la suite du dernier caractère affiché.

Exemple :

La commande :

```
? 'Nom du client'
```

affiche la chaîne de caractères 'Nom du client' à l'écran.

III.2.5.9. Introduction de données avec ACCEPT et INPUT

Forme BNF :

```
<accept> ::= ACCEPT <expr> TO <var>
```

Cette commande affiche la valeur de <expr> à l'écran et puis attend l'introduction d'une chaîne de caractères au clavier pour la ranger dans <var>.

Exemple :

La commande :

```
accept 'Nom du client ?' to NOM
```

entraîne l'affichage du message 'Nom du client ?'. Cette question reste posée jusqu'à ce qu'on introduise une chaîne de caractères qui est mémorisée dans la variable interne nom.

La commande INPUT est destinée à introduire une valeur numérique dans le programme; sa syntaxe est semblable à celle de ACCEPT.

III.2.5.10. La commande LOCATE

Structure de la commande :

```
<locate> ::= LOCATE [<portée>] FOR <condition>
```

```
<portée> ::= next entier
```

<portée> est l'expression d'un intervalle d'enregistrements consécutifs. La commande <locate> sert à trouver l'emplacement d'un enregistrement possédant certaines caractéristiques requises. Si <portée> est spécifiée, la recherche commence à partir de l'enregistrement courant et se termine lorsque la portée <portée> est dépassée, sinon la recherche commence au début du fichier et se poursuit jusqu'à la fin du fichier.

Exemple :

Pour demander à dBase des informations relatives à Dupond, on peut taper :

```
LOCATE FOR NOM = 'Dupond'
```

III.2.5.11. Structure de boucle

dBase propose deux commandes associées DO WHILE et ENDDO qui autorisent la répétition d'un bloc d'instructions compris entre les mots DO WHILE et ENDDO. dBase doit également connaître la condition contenue entre les deux formes DO WHILE et ENDDO.

Une boucle a la forme BNF suivante :

```
<while> ::= DO WHILE <condition>
```

```
          <séquence>
```

```
          ENDDO
```

III.2.5.12. L'alternative

Structure BNF de l'alternative :

```
<if> ::= IF <condition> <séquence> ENDIF |  
        IF <condition> <séquence1> ELSE <séquence2> ENDIF
```

Cette instruction spécifie qu'une action doit être exécutée si une certaine condition est vraie, si elle est fausse, deux cas peuvent se présenter :

- aucune instruction n'est exécutée ou
- l'instruction qui suit le mot ELSE est exécutée.

L'exemple suivant a pour but de faire afficher à l'écran pour chaque LIGNECOM de la base de données le prix à facturer.

```

select 1
use LIGNECOM alias LIG          (1)
select 2
use PRODUIT alias PRO          (2)

select LIG                      (3)
goto top                       (4)

do while .not. eof()

    select PRO                  (5)
    locate for NPRO = LIG->NPRO (6)

    if .not. eof()
        ?LIG->QCOM*PRIX
    endif                      (7)

select LIG
skip                          (8)
enddo                        (9)

```

- (1) ouverture du fichier LIGNECOM et définition de son surnom LIG.
- (2) ouverture du fichier PRODUIT et définition de son surnom PRO.
- (3) rend actif le fichier LIGNECOM.
- (4) rend courant le premier enregistrement du fichier actif.
- (5) rend courant le fichier PRODUIT.
- (6) accède à un enregistrement dont le champ NPRO du fichier actif est égal au champ NPRO du fichier LIGNECOM.
Pour utiliser un enregistrement d'un fichier non actif, il faut ajouter l'alias et une flèche avant le nom de champ.
- (7) Si la fin du fichier n'est pas atteint alors afficher LIG->QCOM*PRIX à l'écran.
- (8) l'instruction SKIP fait passer à l'enregistrement suivant.
- (9) la forme ENDDO provoque une boucle vers l'évaluation de la condition ".not.eof()" (fin du fichier atteinte?). Le processus continue jusqu'à la fin de la base de données.

III.3. CONCEPTION D'UN GENERATEUR ADL - dBase III

Les premières sections de ce chapitre sont consacrées aux principes généraux de la compilation. Dans les dernières sections, un compilateur en particulier est développé. Nous nous limiterons au strict nécessaire en priant le lecteur intéressé par le sujet de trouver une bibliographie exhaustive dans [AHO & ULLMAN,86].

III.3.1. Vue générale d'un compilateur

La compilation d'un texte en une traduction équivalente se fait en plusieurs phases : l'analyse lexicale, l'analyse syntaxique et la génération de code.

Le principe d'un compilateur est représenté par la figure suivante :

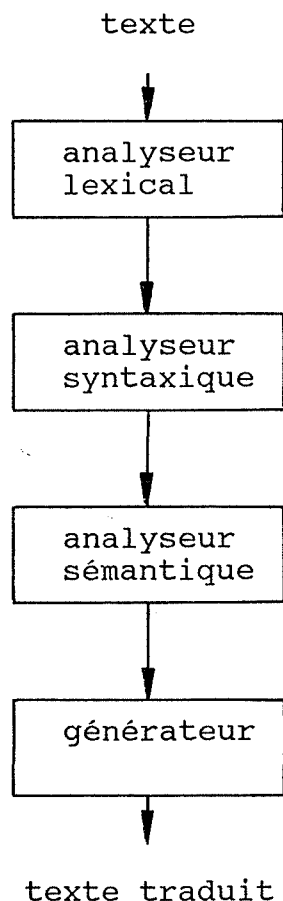


Figure 3.1. : Principe d'un compilateur.

Deux raisons poussent à séparer l'analyse syntaxique et l'analyse lexicale :

- une conception générale plus simple (l'analyseur lexical peut facilement supprimer des blancs et des commentaires, alors qu'il est moins facile d'incorporer des conventions pour les séparateurs et les commentaires dans un analyseur syntaxique),

- une conception plus soignée et plus efficace de chaque analyseur.

III.3.2. Analyse lexicale

Le rôle d'un analyseur lexical est de lire des caractères en entrée, et de produire une suite de symboles : les symboles terminaux d'une grammaire. L'analyseur syntaxique détermine ensuite si cette suite de symboles peut être dérivée des règles de cette grammaire, c'est-à-dire si cette suite de symboles appartient au langage.

III.3.3. Analyse syntaxique

En toute généralité, un analyseur syntaxique vérifie si la structure syntaxique d'un programme est bien formée par rapport aux règles de grammaire du langage cible.

III.3.3.1. Introduction

La syntaxe d'un langage est un ensemble de règles de grammaire dans lesquelles apparaissent les symboles terminaux du langage. La syntaxe décrit ce qu'on peut écrire, sans vouloir en exprimer la sémantique. Définir ainsi un langage revient à donner son alphabet, et décrire les suites de caractères qui appartiennent au langage (par exemple avec une notation BNF).

De telles règles permettent de spécifier un analyseur syntaxique qui détermine si une suite de caractères appartient oui ou non au langage.

III.3.3.2. Notion de forme de phrase, de phrase et de dérivation

Forme de phrase

- la variable distinguée est une forme de phrase
- $\forall 0 \leq i \leq n$:
 W_i est une classe syntaxique ou un symbole terminal

$\forall 0 \leq j \leq m$:
 Z_j est une classe syntaxique ou un symbole terminal

Si $W_0 W_1 \dots W_n$ est une forme de phrase et
 s'il existe une règle telle que $W_i = Z_0 Z_1 \dots Z_m$

alors

$W_0 W_1 \dots W_{i-1} Z_0 Z_1 \dots Z_m W_{i+1} \dots W_n$ est une forme de phrase.

Dérivation

Remplacer dans une forme de phrase une classe syntaxique par la chaîne de classes syntaxiques et/ou de symboles terminaux qui la caractérise s'appelle DERIVATION.

Nous dirons que A "dérive" B s'il existe une production $A ::= B$.

Phrase

Une phrase est une forme de phrase composée uniquement de symboles terminaux.

L'objectif de l'analyse syntaxique est donc de déterminer si un texte donné est une phrase dans un langage donné.

III.3.3.3. Représentation Arborescente

Un arbre de dérivation décrit l'ordre dans lequel les règles de grammaire ont été appliquées pour dériver un programme.

Considérant un sous-arbre de dérivation, la racine de celui-ci est étiquetée par le nom de la classe syntaxique située dans le membre gauche de la règle de grammaire qui s'applique; les feuilles (branches) du sous-arbre sont étiquetées par les noms des symboles terminaux (classes syntaxiques) situés dans le membre droit de cette règle. A la variable distinguée, on associe un noeud.

On pourra donc refléter la situation d'une phrase du langage par son arbre de dérivation.

Exemple

Considérons la grammaire suivante :

```

<instruction> ::= <if>
                | <assignation>
<assignation> ::= id := <expression>
<if> ::= if <condition> then <instruction> endif
<condition> ::= <expression> < <expression>
<expression> ::= <expression> + <terme>
                | <terme>
<terme> ::= <facteur>
<facteur> ::= id

```

Remarque :

id représente un identificateur.

L'expression

```
if a < b then c := d + e endif
```

a pour arbre de dérivation :

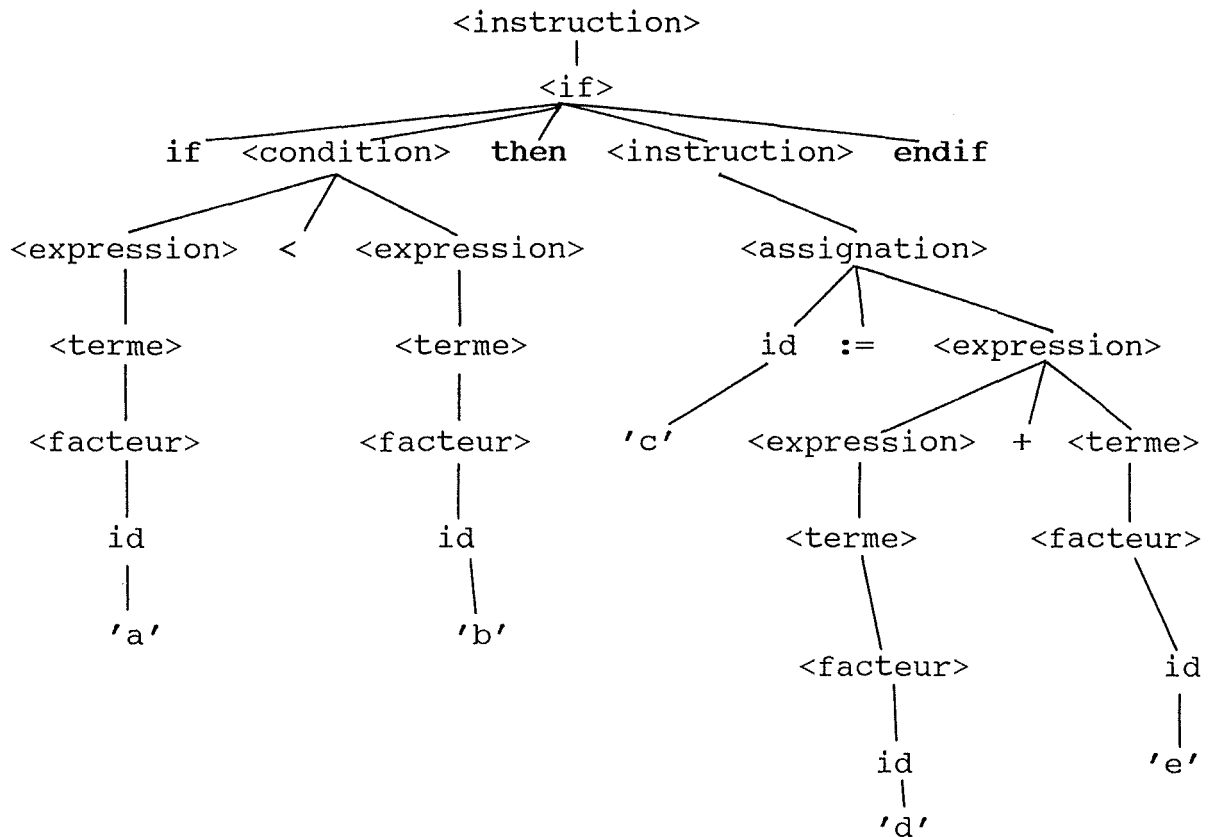


Figure 3.7. : Exemple d'un arbre de dérivation.

III.3.3.4. Classification des analyseurs syntaxiques

Les analyseurs syntaxiques les plus utilisés sont soit des analyseurs "top-down" soit des analyseurs "bottom-up".

Une analyse syntaxique "top-down" construit l'arbre de dérivation à partir de la racine de l'arbre jusqu'aux feuilles tandis qu'une analyse syntaxique "bottom-up" construit l'arbre en commençant par les feuilles et en remontant jusqu'à la racine.

III.3.3.5. L'analyse syntaxique "bottom-up"

Pour mettre en oeuvre l'analyseur syntaxique "bottom-up" on utilisera une technique appelée "shift/reduce parsing".

Le "shift/reduce parsing" essaye de construire un arbre de

dérivation pour une chaîne de symboles terminaux donnée en entrée en commençant par les feuilles et en continuant vers le haut jusqu'à la racine (bottom-up). On peut considérer que ce processus revient à réduire une chaîne de symboles terminaux en variable distinguée de la grammaire.

A chaque étape de réduction, un préfixe particulier de la chaîne d'entrée, qui coïncide avec le membre droit d'une production, est remplacé par le symbole du membre gauche associé. Si le préfixe est choisi correctement à chaque étape, une dérivation est réalisée à l'envers.

Exemple

Considérons la grammaire suivante :

<assignation> ::= id := <expression>	(règle 1)
<expression> ::= <expression> + <terme>	(règle 2)
<terme>	(règle 3)
<terme> ::= <facteur>	(règle 4)
<facteur> ::= id	(règle 5)

Après le processus d'analyse lexicale, le texte contenu dans un fichier et écrit sous forme de caractères est transformé en symboles terminaux.

Considérons la chaîne de symboles terminaux suivante :

id := id + id

Pour réaliser un analyseur syntaxique par la technique du "shift/reduce parsing" on utilise une pile pour mémoriser des symboles grammaticaux, et un tampon d'entrée qui contient la chaîne à analyser. Avant l'analyse syntaxique, le fond de la pile et la fin du tampon d'entrée sont marquées par le symbole '\$'.

Au début de l'analyse syntaxique le tampon d'entrée contient donc, mis à part le symbole '\$', le texte à analyser sous forme d'une chaîne de symboles terminaux.

Schématiquement nous avons la situation suivante :

pile

\$	
----	--

tampon d'entrée

id	:=	id	+	id	\$
----	----	----	---	----	----

L'analyseur syntaxique fait glisser zéro ou plusieurs symboles du tampon d'entrée sur la pile (c'est une action "shift") jusqu'à ce que la suite de symboles empilés coïncide avec le membre droit d'une production, dans ce cas on "réduit" (c'est une action "reduce") la pile en dépilant la plus longue suite de symboles reconnue et en empilant le symbole de la classe syntaxique du membre gauche de la production correspondant à cette suite.

L'analyseur syntaxique répète ce processus jusqu'à ce que la pile contienne la variable distinguée, <assignation> dans notre exemple, et que le tampon soit vide.

Le processus "shift/reduce parsing" est représenté dans le tableau suivant :

pile	tampon	action	
\$	id:=id+id\$	shift	
\$id	:=id+id\$	shift	
\$id:=	id+id\$	shift	
\$id:=id	+id\$	reduce	règle 5
\$id:=<facteur>	+id\$	reduce	règle 4
\$id:=<terme>	+id\$	reduce	règle 3
\$id:=<expression>	+id\$	shift	
\$id:=<expression>+	id\$	shift	
\$id:=<expression> + id	\$	reduce	règle 5
\$id:=<expression>+<facteur>	\$	reduce	règle 4
\$id:=<expression>+<terme>	\$	reduce	règle 2
\$id:=<expression>	\$	reduce	règle 1
\$<assignation>	\$		

La phrase `id := id + id` peut être réduite en <assignation>, on dira que cette chaîne appartient au langage.

L'analyseur syntaxique doit donc effectuer une opération shift ou bien une opération reduce. Le choix de cette opération est donné par une table d'analyse (syntaxique).

III.3.3.6. La table d'analyse (ou table SR)

La table d'analyse, qui est produite automatiquement, a la forme suivante.

Les colonnes de la table d'analyse représentent les symboles terminaux de la grammaire tandis que les lignes représentent tous les symboles grammaticaux (symboles terminaux et variables syntaxiques). La fin d'une ligne et la fin d'une colonne sont marquées par le symbole '\$', symbole qui n'appartient pas à la grammaire.

La table d'analyse de notre petit exemple peut se représenter par le tableau suivant :

		symboles terminaux			
		id	+	:=	\$
variables syntaxiques	<assignation>	E	E	E	R
	<expression>	E	S	E	R
	<terme>	E	R	E	R
	<facteur>	E	R	E	R
	id	E	R	S	R
symboles terminaux	+	S	E	E	E
	:=	S	E	E	E
	\$	S	E	E	E

Figure 3.8. : Format de la table d'analyse ou table SR.

Le sommet de la pile et le premier élément du tampon d'entrée forment un couple. Ce couple aboutit à une valeur dans la table

d'analyse qui indique l'opération à effectuer.

Il y a trois valeurs possibles dans la table d'analyse :

S : l'opération à effectuer est SHIFT

R : l'opération à effectuer est REDUCE

E : la situation est erronée

Deux cas d'erreur

- L'analyseur syntaxique trouve une valeur E dans la table SR : le texte n'est pas une phrase de la grammaire.

- L'analyseur syntaxique rencontre une suite de symboles qu'il n'arrive pas à réduire en variable distinguée.

Lors d'une erreur, l'analyseur syntaxique arrête d'analyser le texte en entrée et affiche un "syntax error".

III.3.3.7. Construction de la table d'analyse

Signification par rapport au processus d'analyse lexicale

Les couples (a,b) : $SR[a,b] = S$

sont des couples dont la suite $a b$ apparaît dans une règle de production.

Les couples (a,b) : $SR[a,b] = R$

sont des couples tels que a termine une partie droite d'une règle de production et que b soit un \$ ou un symbole terminal.

Les couples (a,b) : $SR[a,b] = E$

sont des couples dont la suite $a b$ ne peut jamais apparaître dans une forme de phrase.

Règles de construction de la relation shift et reduce

Nous allons définir les règles de "construction" de la relation "shift" et de la relation "reduce".

Règles pour la relation "shift"

règle (1-a) \$ shift <variable distinguée>

règle (1-b) Si X shift Y et

X est un \$ ou un symbole grammatical et
 $Y ::= L_0 \dots L_{n-1} \quad \text{où } n-1 \geq 0$

alors X shift L_0

règle (1-c) Si $Y ::= L_0 \dots L_{n-1} \quad \text{où } n-1 \geq 0$

alors L_0 shift L_1 , L_1 shift L_2 , ... , L_{n-2} shift L_{n-1}

En partant de la règle 1a et puis par application des règles (1-c) et (1-b) nous trouvons toutes les relations "shift" qui existent pour une grammaire donnée.

Il y a un risque de boucler car l'application de ces règles peut produire des couples déjà générés! Pour pouvoir automatiser ce processus, il faut s'intéresser uniquement aux couples non encore générés.

Règles pour la relation "reduce"

règle (2-a) <variable distinguée> reduce \$

règle (2-b) Si X shift Y et

Y est un \$ ou un symbole grammatical et

$X ::= L_0 \dots L_{n-1} \quad \text{où } n-1 \geq 0$

alors L_{n-1} reduce Y

règle (2-c) Si X reduce Y et

X est un \$ ou un symbole grammatical et

$Y ::= L_0 \dots L_{n-1}$ où $n-1 \geq 0$

alors X reduce L_0

règle (2-d) Si X reduce Y et

Y est un \$ ou un symbole grammatical et

si $X ::= L_0 \dots L_{n-1}$ où $n-1 \geq 0$

alors L_{n-1} reduce Y

En appliquant ces dernières règles, nous trouvons tous les couples de la table SR qui provoqueront un "reduce". Les couples qui ne produisent ni "shift" ni "reduce" sont des couples qui conduisent à une erreur E.

Cependant, pour éviter tout problème lors de l'analyse syntaxique, la grammaire doit vérifier certaines conditions.

Condition 1

Dans la table SR, il ne peut pas y avoir des couples (a,b) vérifiant à la fois a shift b et a reduce b.

Condition 2

Il ne peut y avoir dans la grammaire deux règles de production qui ont la même partie droite : l'opération de réduction est ainsi univoque.

Condition 3

Supposons qu'on ait deux règles de production :

$X ::= L_0 \dots L_{i-1} L_i L_{i+1} \dots L_{n-1}$
et

$Y ::= L_i L_{i+1} \dots L_{n-1}$

Y est un suffixe de X

Dans la table SR nous ne pouvons pas avoir L_{i-1} shift Y sinon la chaîne $L_0 \dots L_{i-1} L_i L_{i+1} \dots L_{n-1}$ sera toujours réduite à X et jamais à $L_i L_{i+1} Y$.

Exemple :

Construisons la table SR pour la grammaire suivante :

```

<assignation> ::= id := <expression>
<expression> ::= <expression> + <terme>
                | <terme>
<terme> ::= <facteur>
<facteur> ::= id

```

Les relations shift

L'application de la règle (1-a) produit la relation :

"\$" shift "<assignation>".

Puisque "<assignation> ::= id := <expression>", l'application de la règle (1-c) produit les relations :

"id" shift "==" et
 "==" shift "<expression>".

Puisque "<expression> := <expression> + <terme>", l'application de la règle (1-c) produit les relations :

"<expression>" shift "+" et
 "+" shift "<terme>".

Puisque "\$" shift "<assignation>" et "<assignation> ::= id := <expression>", l'application de la règle (1-b) produit la relation:

"\$" shift "id".

Puisque "==" shift "<expression>" et "<expression> ::= <terme>", l'application de la règle (1-b) produit la relation :

"==" shift "<terme>".

Puisque "+" shift "<terme>" et "<terme> ::= <facteur>", l'application de la règle (1-b) produit la relation :

"+" shift "<facteur>".

Puisque "==" shift "<terme>" et "<terme> ::= <facteur>",
l'application de la règle (1-b) produit la relation :

"==" shift "<facteur>".

Puisque "+" shift "<facteur>" et "<facteur> ::= id",
l'application de la règle (1-b) produit la relation :

"+" shift "id".

Puisque "==" shift "<facteur>" et "<facteur> ::= id",
l'application de la règle (1-b) produit la relation :

"==" shift "id".

Les relations reduce

L'application de la règle (2-a) produit la relation :

"<assignation>" reduce "\$".

Puisque "<expression>" shift "+" et "<expression> ::= <terme>", l'application de la règle (2-b) produit la relation :

"<terme>" reduce "+".

Puisque "<assignation>" reduce "\$" et "<assignation> ::= id
:=<expression>", l'application de la règle (2-d) produit la
relation :

"<expression>" reduce "\$".

Puisque "<terme>" reduce "+" et "<terme> ::= <facteur>",
l'application de la règle (2-d) produit la relation :

"<facteur>" reduce "+".

Puisque "<expression>" reduce "\$" et "<expression> ::= <expression> + <terme>", l'application de la règle (2-d) produit la relation :

"<terme>" reduce "\$".

Puisque "<facteur>" reduce "+" et "<facteur> ::= id",
l'application de la règle (2-d) produit la relation :

"id" reduce "+".

Puisque "<terme>" reduce "\$" et "<terme> ::= <facteur>",
l'application de la règle (2-d) produit la relation :

"<facteur>" reduce "\$".

Puisque "<facteur>" reduce "\$" et "<facteur> ::= id",
l'application de la règle (2-d) produit la relation :

"id" reduce "\$".

Toutes les relations forment bien la table SR de la page 40:

	id	+	:=	\$
<assignation>	E	E	E	R
<expression>	E	S	E	R
<terme>	E	R	E	R
<facteur>	E	R	E	R
id	E	R	S	R
+	S	E	E	E
:=	S	E	E	E
\$	S	E	E	E

III.3.4. La génération du code

III.3.4.1. Introduction

Au fur et à mesure de l'analyse syntaxique, le compilateur construit une forêt d'arbres de traduction.

Si l'analyse se termine par une conclusion positive, c'est-à-dire, si le texte est une phrase de la grammaire, alors cette forêt se résume à un seul arbre : l'arbre de traduction du texte source.

Cet arbre de traduction a la propriété suivante :

le parcours des noeuds terminaux dans l'ordre " à gauche de " donne la traduction de la phrase.

4.2 Principe

A chaque règle de la grammaire on fait correspondre une règle de traduction.

Chaque opération "reduce" aura pour effet supplémentaire d'associer à la variable syntaxique réduite son arbre de traduction, à partir de la règle de traduction de celle-ci.

Finalement, si on arrive à la variable distinguée, on aura automatiquement l'arbre de traduction correspondant au texte initial. Le parcours de cet arbre fournit la traduction.

Exemple

La traduction d'une règle BNF est notée par trad(règle BNF). Les chaînes de caractères entre apostrophes représentent le code dBase généré lors de la compilation. Il est implicitement sous-entendu que l'opération entre les différents attributs de traduction est la concaténation de chaînes de caractères au sens habituel. Aucun symbole particulier n'a donc été employé.

Reprenons la grammaire définie dans le chapitre III.3.3.3. et ajoutons-y les règles de traduction, ce qui donne :

```
<instruction> ::= <if> |
                  <assignation>
```

```
trad(<instruction>) = trad(<if>) |
                    trad(<assignation>)
```

```
<if> ::= if <condition> then <instruction> endif
```

```
trad(<if>) = 'if' trad(<condition>) trad(<instruction>)
```

'endif'

<condition> ::= <expression> < <expression>

trad(<condition>) = trad(<expression>) '<' trad(<expression>)

<expression> ::= <terme>

trad(<expression>) = trad(<terme>)

<terme> ::= <facteur>

trad(<terme>) = trad(<facteur>)

<facteur> ::= identifier

trad(<facteur>) = id

<assignation> ::= id := <expression>

trad(<assignation>) = id '=' trad(<expression>)

Supposons qu'on ait à traduire l'instruction

if a < b then c:=d endif.

L'application du processus de construction de l'arbre de traduction est explicitée ci-dessous :

pile	tampon d'entrée	action
\$	if id < id then id := id endif \$ a b c d	shift
\$ if	id < id then id := id endif \$ a b c d	shift
\$ if id	< id then id := id endif \$ b c d	reduce

\$	if	<facteur>
----	----	-----------

 < id then id := id endif \$
 | | |
 b c d

reduce

\$	if	<terme>
----	----	---------

 < id then id := id endif \$
 | | |
 b c d

reduce

\$	if	<expression>
----	----	--------------

 < id then id := id endif \$
 | | |
 b c d

shift

\$	if	<expression>	<
----	----	--------------	---

 id then id := id endif \$
 | | |
 b c d

shift

\$	if	<expression>	<	id
----	----	--------------	---	----

 then id := id endif \$
 | |
 c d

reduce

\$	if	<expression>	<	<facteur>
----	----	--------------	---	-----------

 then id := id endif \$
 | |
 c d

reduce

\$	if	<expression>	<	<terme>
----	----	--------------	---	---------

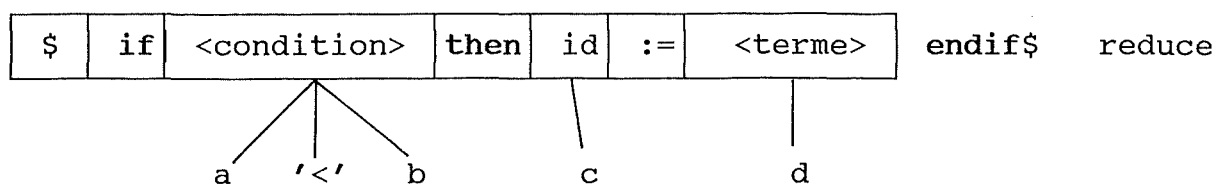
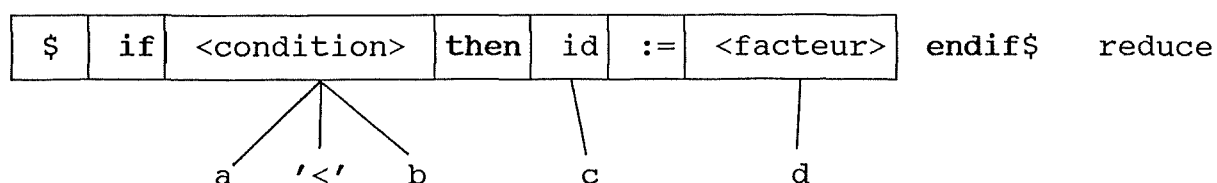
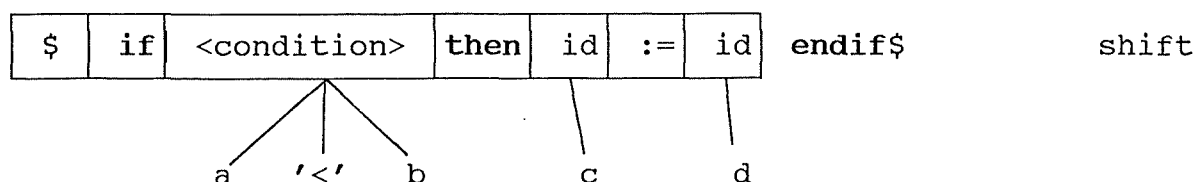
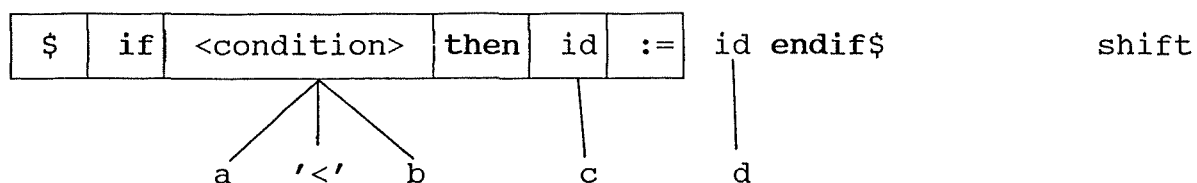
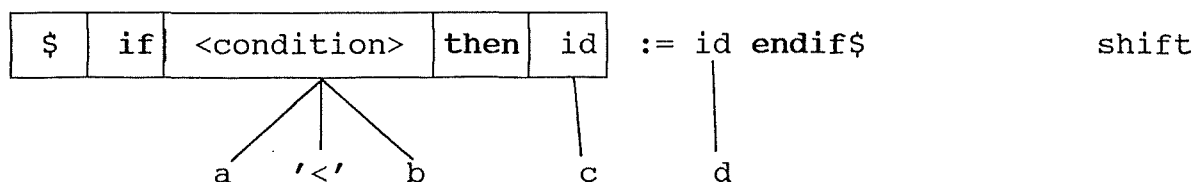
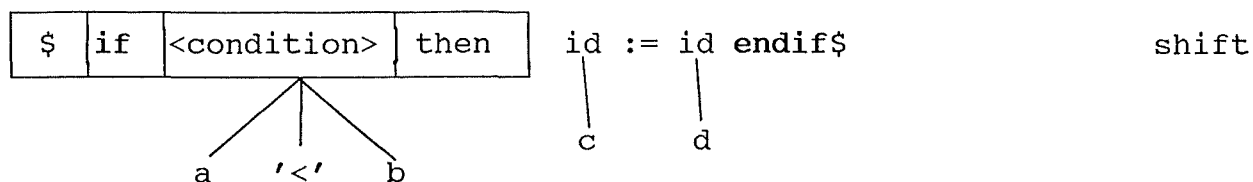
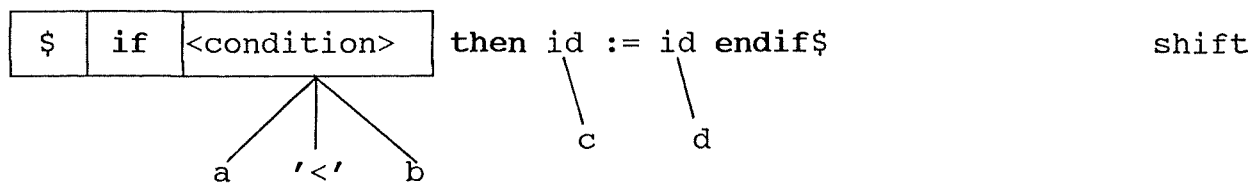
 then id := id endif \$
 | |
 c d

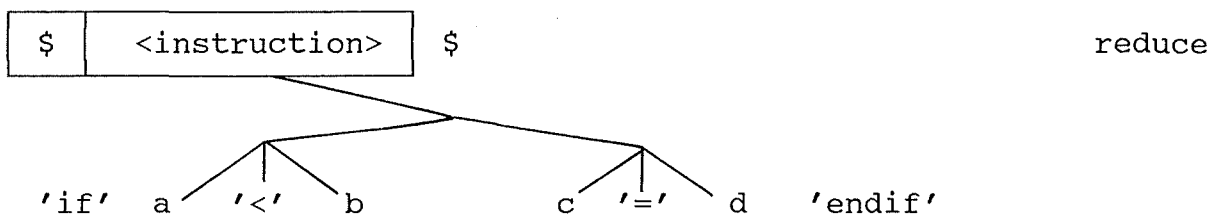
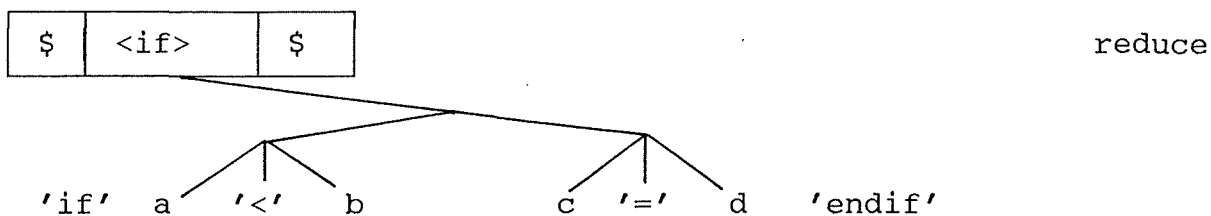
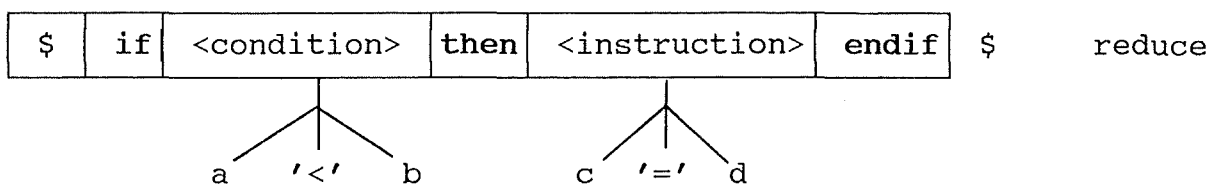
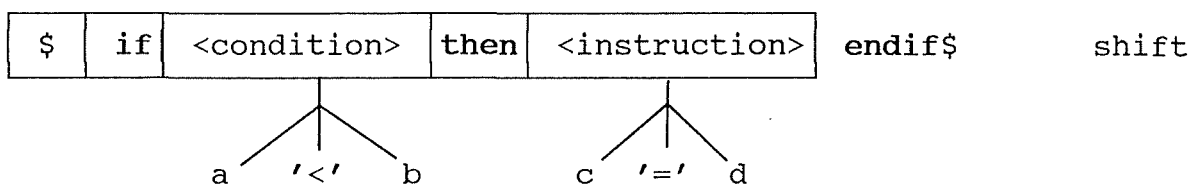
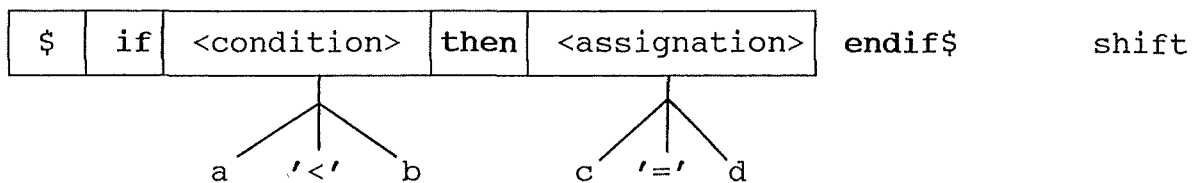
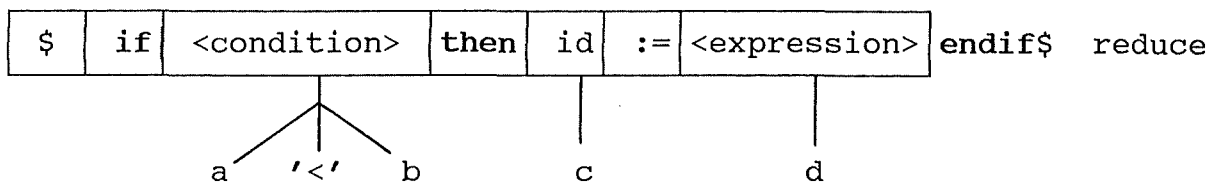
reduce

\$	if	<expression>	<	<expression>
----	----	--------------	---	--------------

 then id:=id endif \$
 | |
 c d

reduce





Le parcours des noeuds terminaux dans l'ordre "à gauche de" donne la traduction :

if a < b c = d endif

III.6. Règles de traduction

III.6.1. Introduction

Rappelons qu'à chaque règle BNF de la grammaire d'ADL on fait correspondre une règle de traduction. La traduction d'une règle BNF est notée par `trad( règle BNF )`. Les chaînes de caractères entre apostrophes représentent le code dBase généré lors de la compilation. Il est implicitement sous-entendu que l'opération entre les différents attributs de traduction est la concaténation de chaînes de caractères au sens habituel.

Remarquons que la traduction d'un identificateur, d'une chaîne de caractères, d'un nombre est respectivement cet identificateur, cette chaîne de caractères et ce nombre.

III.6.2. Les règles de traduction

Traduction d'une séquence

```
<séquence> ::= <instruction>. |
              <seq> <instruction>.
```

```
trad(<séquence>) = trad(<instruction> |
                      trad(<seq>) trad(<instruction>))
```

```
<seq> ::= <instruction>; |
         <seq> <instruction>;
```

```
trad(<seq>) = trad(<instruction> |
                  trad(<seq>) trad(<instruction>))
```

Traduction des instructions

```
<instruction> ::= <write> |
                  <writeln> |
                  <read> |
                  <assignation> |
                  <if> |
                  <while> |
                  <for>
```

```
trad(<instruction>) = trad(<write>) |
                    trad(<writeln>) |
                    trad(<read>) |
                    trad(<assignation>) |
                    trad(<if>) |
                    trad(<while>) |
                    trad(<for>)
```

<write> ::= write (<expression>)

trad(<write>) = '?'trad(<expression>)

<writeln> ::= writeln (<expression>)

trad(<writeln>) = '??'trad(<expression>)

<constante> ::= <string> |
entier

trad(<constante>) = trad(<string>) |
trad(entier)

<read> ::= read (<string> , identificateur)

Si la valeur de "identificateur" représente une chaîne de caractères alors :

trad(<read>) = 'accept' trad(<string>) 'to'
trad(identificateur)

Si la valeur de "identificateur" représente une constante entière alors :

trad(<read>) = 'input' trad(<string>) 'to'
trad(identificateur)

<assignation> ::= identificateur := <expression>

trad(<assignation>) = trad(identificateur) '='
trad(<expression>)

<expression> ::= <expression> <opérateur-d'addition> <terme> |
<terme>

trad(<expression>) = trad(<expression>) trad(<opérateur-
d'addition>) trad(<terme>) |
trad(<terme>)

<opérateur-d'addition> ::= + | -

trad(<opérateur-d'addition>) = '+' | '-'

<terme> ::= <facteur> |
<terme> <opérateur-multiplicateur> <facteur>

```
trad(<terme>) = trad(<facteur>) |
               trad(<terme>) trad(<opérateur-multiplicateur>)
               trad(<facteur>)
```

```
<opérateur-multiplicateur> ::= * | /
```

```
trad(<opérateur-multiplicateur>) = * | /
```

```
<facteur> ::= identificateur |
              <constante> |
              <attribut> |
              ( <expression> )
```

```
trad(<facteur>) = trad(identificateur) |
                  trad(<constante>) |
                  trad(<attribut>) |
                  '(' trad(<expression>) ')'
```

```
<attribut> ::= <nom-attribut> (: <variable-entité> )
```

Si <nom-attribut> est le nom d'un attribut d'un type d'entité que référence la variable d'entité <variable-entité> alors :

```
trad(<attribut>) = trad(<variable-entité>) '->' trad(<nom-
                  attribut>)
```

```
<condition-n> ::= <expression> <oper-rel> <expression>
```

Si les expressions <expression> désignent des valeurs compatibles alors :

```
trad(<condition-n>) = trad(<expression>) trad(<oper-rel>)
                    trad(<expression>)
```

```
<oper-rel> ::= = | <> | < | <= | >= | >
```

```
trad(<oper-rel>) ::= '=' | '<>' | '<' | '<=' | '>=' | '>'
```

```
<nom-attribut> ::= identificateur
```

Si <nom-attribut> est le nom d'un attribut d'un type d'entité de la base de données alors :

```
trad(<nom-attribut>) = trad(identificateur)
```

```
<variable-entité> ::= identificateur
```

Si <variable-entité> est le surnom d'un type d'entité de la base de données alors :

```
trad(<variable-entité>) = trad(identificateur)
```

L'assignation d'entité

`<assignation-entité> ::= <variable-entité> := <expression-entité>`

`trad(<assignation-entité>) = 'select' trad(<variable-entité>)
trad(<expression-entité>)`

`<expression-entité> ::= <type-entité> <CSEL>`

`trad(<expression-entité>) = trad(<CSEL>)`

`<CSEL> ::= <condition-ass> |
 <condition-att>`

Une condition d'association sera transformée en une condition d'attribut. Un type d'association est représentée en dBase de la manière suivante : on ajoute à l'entité cible un nouvel attribut qui correspond à l'identifiant de l'entité origine. Cet attribut permet de représenter l'association entre l'entité origine et l'entité cible.

Considérons l'assignation suivante :

`PRO:=PRODUIT(PL:LIG)`

Il existe entre PRODUIT et LIGNECOM un type d'association PL. Sachant que PRODUIT est identifié par NPRO, on représente PL par un nouveau champ NPRO dans le fichier LIGNECOM de telle sorte que la valeur NPRO identifie le PRODUIT associé à LIGNECOM. Ce processus est illustré par la figure suivante :

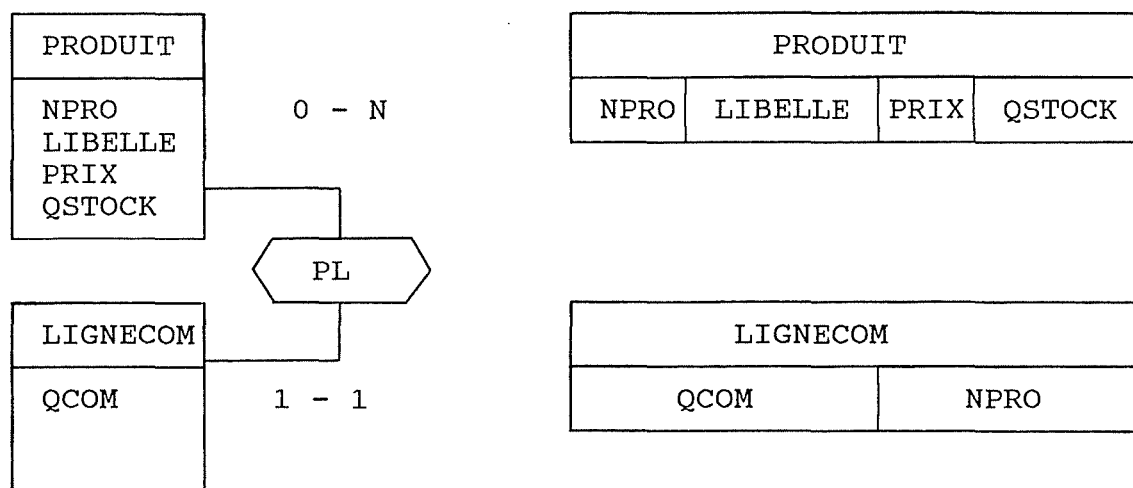


Figure 3.2. : Principe de représentation d'un type d'association.

On transformera la condition d'association comme suit :

PRO:= PRODUIT(NPRO = NPRO(:LIG))

où la valeur de NPRO de LIGNECOM est la référence d'un enregistrement PRODUIT

trad(<CSEL>) ::= trad(<condition-ass>) |
 trad(<condition-att>)

<condition-att> ::= (: <nom-attribut> <oper-rel> <expression>)

Si <nom-attribut> est un attribut du type d'entité évalué et si <nom-attribut> et <expression> désignent des valeurs comparables alors :

trad(<condition-att>) = 'locate next 10000 for ' trad(<nom-attribut>) trad(<oper-rel>) trad(<expression>)

<condition-ass> ::= (<type-association> : <variable-entité>)

Si <type-association> est le nom d'un type d'association entre le type d'entité que référence <variable-entité> et le type d'entité évalué alors:

trad(<condition-ass>) = 'locate next 10000 for ' VAR '='
 NOM '->' VAR

où - VAR désigne le champ du type d'entité cible que référence le type d'entité origine;
- NOM désigne le nom de la variable d'entité du type d'entité évaluée.

L'alternative

<if> ::= if <condition-n> then <seq> endif |

if <condition-n> then <seq> else <seq> endif

trad(<if>) = 'if' trad(<condition-n>) trad(<seq>) 'endif'|
 'if' trad(<condition-n>) trad(<seq>) 'else'
 trad(<seq>) 'endif'

<if> ::= if <ens> <condition-att> then <seq> endif |

if <ens> <condition-att> then <seq> else <seq> endif

```

trad(<if>) = 'select ' NOM
            'goto top '
            'N = 0 '
            'do while .not. eof() '
            trad(<condition-att>)
            'if eof() '
            'exit '
            'endif '
            'X' NOM '= recno()'
            'N = N + 1 '
            'select ' NOM
            'goto X' NOM ' + 1 '
            'enddo '
            'if N <> 0 '
            trad(<seq>)
            'endif '

```

où NOM désigne la variable qui référence le type d'entité <ens>.

```

<if> ::= if <ens> ( <type-association> : <ens> ) do <seq>
      endif

```

L'alternative avec une condition d'association sera traduite de manière différente selon que la connectivité du type d'association est 1 - N ou N - 1.

Si <type-association> est le nom d'un type d'association entre le type d'entité que référence <variable-entité1> et le type d'entité <type-entité> alors:

Si la connectivité du type d'association désigné par <type-association> est 1 - N alors la traduction devient :

```

trad(<if>) = 'select ' NOM2
            'N = 0 '
            'goto top '
            'do while .not. eof() '
            'locate next 10000 for ' VAR '=' NOM '->' VAR
            'if eof() '
            'exit '
            'endif '
            'X' NOM2 '= recno()'
            'N = N + 1 '
            'select ' NOM2
            'goto X' NOM2 ' + 1 '
            'enddo '
            'if N <> 0 '
            trad(<seq>)
            'endif'

```

sinon

```
trad(<if>) = 'select ' NOM2
            'N = 0 '
            'locate for ' VAR '=' NOM '->' VAR
            'if .not. eof() '
            'N = N + 1 '
            'endif'
            'if N <> 0 '
                trad(<seq>)
            'endif'
```

- où - VAR désigne le champ du type d'entité cible que référence le type d'entité origine;
 - NOM désigne la variable évaluée;
 - NOM2 désigne la variable que référence le type d'entité évalué.

La boucle while

```
<while> ::= while <condition-n> do <seq> endwhile

trad(<while>) ::= 'do while' trad(<condition-n>)
                  trad(<seq>) 'enddo'
```

La boucle énumérative

Accès séquentiel

```
<for> ::= for <variable-entité> := <type-entité> do
          <seq> endfor
```

Si <type-entité> représente un type d'entité de la base de données alors :

```
trad(<for>) = 'select ' trad(<variable-entité>)
              'goto top '
              'do while .not. eof() '
                  trad(<seq>)
              'select ' trad(<variable-entité>)
              'skip '
              'enddo '
```

Accès par clé

`<for> ::= for <variable-entité> := <type-entité> <condition-att> do <seq> endfor`

Si <type-entité> représente un type d'entité de la base de données alors :

```
trad(<for>) = 'select ' trad(<variable-entité>)
              'goto top '
              'do while .not. eof() '
                trad(<condition-att>)
                'if eof() '
                  'exit '
                'endif '

              'X'trad(<variable-entité>) '= recno()'
                trad(<seq>)
              'select ' trad(<variable-entité>)
              'goto X' trad(<variable-entité>) ' + 1 '
              'enddo '
```

Accès par association

`<for> ::= for <variable-entité> := <type-entité> ( <type-association> : <variable-entité2> ) do <seq> endfor`

La boucle énumérative avec une condition d'association sera traduite de manière différente selon que la connectivité du type d'association est 1 - N ou N - 1.

Si <type-association> est le nom d'un type d'association entre le type d'entité que référence <variable-entité1> et le type d'entité <type-entité> alors:

Si la connectivité du type d'association désigné par <type-association> est 1 - N alors la traduction de la boucle énumérative est la suivante :

```
trad(<for>) = 'select ' trad(<variable-entité>)
              'goto top '
              'do while .not. eof() '
                'locate next 10000 for ' VAR '=' trad(<variable-entité2>) '->' VAR
                'if eof() '
                  'exit '
                'endif '
              'X'trad(<variable-entité>) '= recno()'
                trad(<seq>)
              'select ' trad(<variable-entité>)
              'goto X' trad(<variable-entité>) ' + 1 '
              'enddo '
```

sinon

```
trad(<for>) = 'select ' trad(<variable-entité>)  
             'locate for ' VAR '=' trad(<variable-  
             entité2> '->' VAR  
             'if .not. eof() '  
             trad(<seq>)  
             'endif'
```

où - VAR désigne le champ du type d'entité cible que référence le type d'entité origine.

III.7. REALISATION

III.7.1. Introduction

L'atelier a été réalisé en TURBO PASCAL version 3 sur un ordinateur personnel.

Pour vérifier à la génération la validité des procédures rédigées en LDA, le générateur dBase doit avoir accès à la base des spécifications. La base des spécifications contient le schéma conceptuel, défini en I.4.4., de la base de données en cours de conception. Le système de gestion de base de données NDBS assumera le rôle de la base des spécifications.

NDBS, ou Network Data Base System, est un environnement base de données permettant le développement d'applications base de données en Pascal sur micro-ordinateur.

L'environnement NDBS comprend trois composants : un "data base handler", un "high-level language pre-processor" et un "data dictionary/schema processor" interactif.

Le "data base handler" est un ensemble de procédures qui offre au programmeur la possibilité d'ajouter, de supprimer et de consulter des informations sélectionnées. Le "data base handler" permettra d'introduire le contenu de la base des spécifications.

Le "schema processor" est un programme qui permet à l'utilisateur d'introduire la description de la base des spécifications, de l'afficher et de l'imprimer.

Le "high level language pre-processor" permet au programmeur d'écrire des programmes base de données en un langage appelé LDA-Pascal (Langage de description d'algorithmes imbriqué en Pascal). Ces programmes permettent d'accéder à la base des spécifications. Une étude plus complète de NDBS est disponible dans [HAINAUT,87].

III.7.2. Architecture logique de l'atelier de développement d'applications dBase

Nous avons décomposé notre atelier en modules relativement indépendants. Cette découpe en modules, bien connue en génie logiciel, offre l'avantage de la réutilisabilité éventuelle de certains de ces modules dans d'autres applications.

L'architecture logique de l'atelier est représentée à la page suivante.

III.7.3. Description des modules

Module coordonnateur

Ce module contrôle la présentation des informations sur écran ainsi que le dialogue avec l'utilisateur. Il coordonne l'activation des différents modules.

Module chargeur

Ce module permet d'introduire une description d'une base de données en projet.

Module générateur de rapport

Ce module permet de produire un rapport détaillé de la base des spécifications, ainsi que la description dBase d'une base de données.

Module GENERATEUR dBase

Ce module joue les rôles de gérant (scheduler) et de coordonnateur du générateur dBase, initialise tout ce qui doit l'être puis réalise la compilation effective. C'est dans ce module que se trouvent décrites toutes les constantes, variables globales et symboles de la grammaire.

Module ANALYSEUR LEXICAL

La fonction de ce module est de réaliser l'analyse lexicale, c'est-à-dire de lire un texte donné sous forme de chaîne de caractères et de le transformer, si possible, en termes de symboles terminaux de la grammaire. Tous les symboles terminaux sont stockés dans un tableau appelé tampon d'entrée.

Le module GETTOKEN lit le texte à analyser (qui est obligatoirement contenu dans un fichier), un token à la fois.

Module CONSTRUCTEUR

La fonction de ce module est de construire toutes les structures de données dont a besoin le compilateur : la table d'analyse contenant les opérations shift et reduce correspondant à notre grammaire, la table contenant toutes les règles de production et la table qui contiendra les rubans de traduction.

Pour ce faire, ce module se compose essentiellement de trois procédures, à savoir :

- la procédure BUILD_PRODUCTION qui construit le tableau contenant toutes les règles de la grammaire à partir d'un fichier permanent;
- la procédure BUILD_SRE qui construit la table d'analyse correspondant à la grammaire;
- la procédure BUILD_FS qui construit la table des rubans de traduction. Les rubans de traduction sont obtenus à partir d'un fichier dans lequel ils ont été sauvegardés.

Module TRADUCTEUR

La fonction de ce module est de réaliser l'analyse syntaxique. Au fur et à mesure de l'analyse syntaxique, ce module fait l'analyse sémantique et construit l'arbre de traduction. L'analyse sémantique nécessite l'accès à la base des spécifications NDBS.

Module PARCOUREUR D'ARBRE

La fonction de ce module est de parcourir l'arbre de traduction, en vue de produire, dans un fichier, le texte dBase.

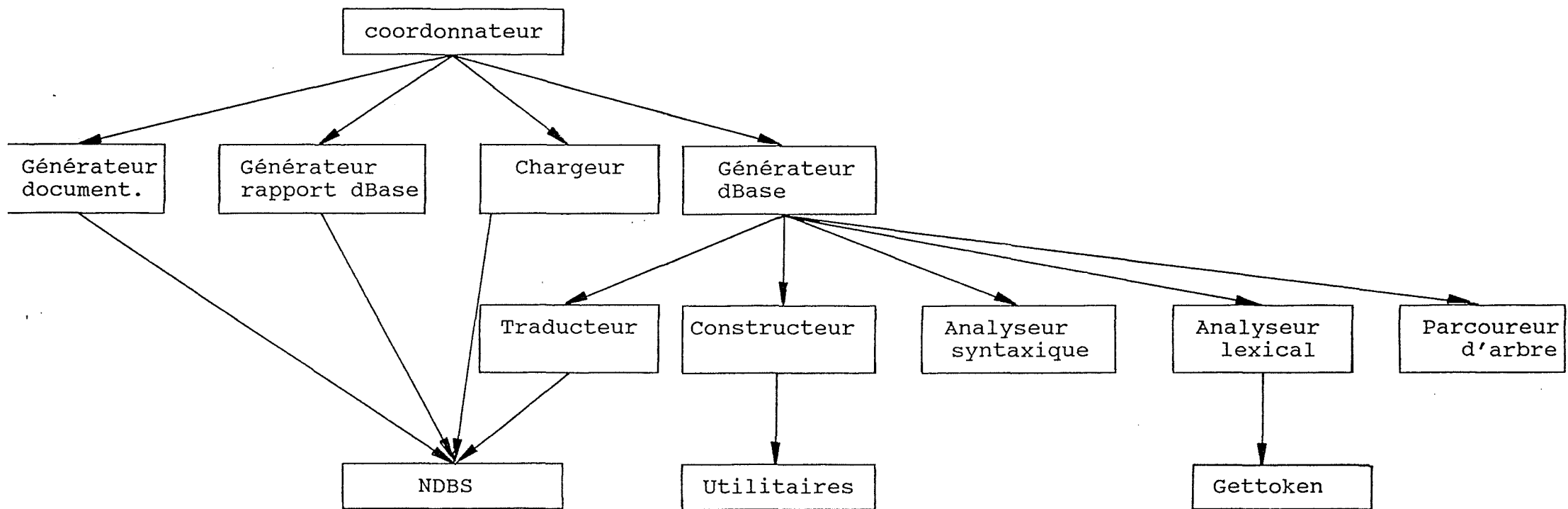
Module UTILITAIRES

La fonction de ce module est de fournir à l'ensemble des autres modules toute une panoplie de fonctions, d'outils ayant pour but de faciliter le travail de conception de ces modules. C'est ainsi que nous trouverons, entre autres, dans ce module des procédures permettant

- d'intervenir dans la construction de l'arbre de traduction;
- de modifier une pile;
- de gérer et de manipuler des pointeurs, etc.

III.7.4. Implémentation de l'atelier

Le texte des différents modules formant l'atelier que nous avons réalisé se trouve dans un appendice séparé.



→ : appel de services

Figure 3.10. : Architecture de l'atelier.

III.8. EXEMPLES

III.8.1. Domaine d'application

Le domaine d'application a été décrit en I.4.3. de ce mémoire. Le format et le contenu initial des fichiers est donné en annexe 1.

III.8.2. Le schéma Entité-Association

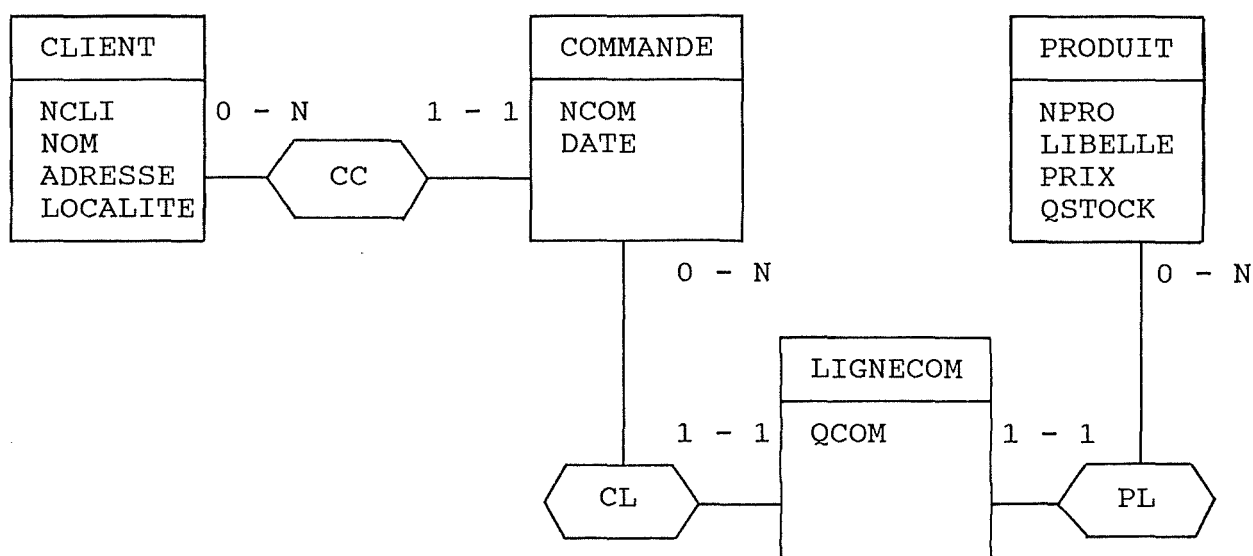


Figure 3.4. : Le schéma Entité-Association de l'exemple développé dans ce chapitre.

III.8.3. Spécification des traitements

Nous retenons les quatre fonctions suivantes :

Fonction 1 permet d'obtenir les libellés des produits avec leur identifiant et la qualité disponible.

Fonction 2 affiche pour chaque commande d'un client spécifié les informations sur ce client, cette commande, le produit et le prix à facturer.

Fonction 3 affiche à l'écran les clients habitant une certaine localité et ayant passé au moins une commande.

Fonction 4 donne tous les clients ayant commandé un certain produit.

Le texte ADL et le résultat de la génération ainsi que quelques exemples d'exécution du texte dBase III sur la base de données de support sont présentés dans les points suivants.

III.8.4. Le programma ADL

```
REPONSE := 'N';
```

```
while REPONSE = 'N' do
write('                                MENU PRINCIPAL');
write('                                -----');
write('1 : Tous les renseignements concernant tous les produits. ');
write('2 : Pour chaque LIGNECOM de chaque commande d'un CLIENT ');
write('    spécifié afficher les informations sur ce CLIENT ');
write('    cette commande, le PRODUIT et le prix facturer ');
write('3 : Tous les renseignements concernant tous les CLIENTS ');
write('    d'une certaine LOCALITE ayant passe au moins une ');
write('    COMMANDE ');
write('4 : Tous les clients ayant commandés un certain produit ');
write('5 : Sortie ');
```

```
read('Choisissez (1 à 5) dans le menu : ',REPONSE);
```

```
if REPONSE = '1' then
write('Numéro      Libellé      Quantité disponible ');
write('-----');
for pro:=produit do
    write(NPRO(:PRO));
    writeln(' ');
    writeln(LIBELLE(:PRO));
    writeln(QSTOCK(:PRO));
endfor;
write('-----');
REPONSE := 'N';
endif;
```

```
if REPONSE = '2' then
read('Numéro de client : ',NUM);
for com:=commande(:ncli=NUM) do
    for lig:=lignecom(cl:com) do
        for pro:=produit(pl:lig) do
            write('Client : ');
            writeln(NCLI(:com));
            writeln(' - Comm : ');
            writeln(NCOM(:com));
            writeln(DATE(:com));
            writeln(' - Produit : ');
            writeln(QCOM(:lig));
```

```

        writeln('x');
        writeln(NPRO(:lig));
        writeln(' - Total :');
        writeln(QCOM(:lig)*prix);
    endfor;
endfor;
endfor;
REPONSE := 'N';
endif;

if REPONSE = '3' then
read('Localité du client : ',LOC);
for cli:=client do
    if commande(cc:cli) then
        if LOCALITE(:CLI) = LOC then
            write(NCLI(:cli));
            writeln(' ');
            writeln(NOM(:cli));
            writeln(ADRESSE(:cli));
        endif;
    endif;
endfor;
REPONSE := 'N';
endif;

if REPONSE = '4' then
read('Numéro du produit : ',num);
for pro:=produit(:npro=num) do
    for lig:=lignecom(pl:pro) do
        for com:=commande(cl:lig) do
            for cli:=client(cc:com) do
                write(LIBELLE(:PRO));
                writeln(' ');
                writeln(NOM(:cli));
                writeln(' ');
                writeln(ADRESSE(:cli));
            endfor;
        endfor;
    endfor;
endfor;
REPONSE := 'N';
endif;
endwhile.

```

III.8.4. Le programma dBase III

```

close databases
select 1
use CLIENT alias CLI
select 2
use COMMANDE alias COM
select 3
use LIGNECOM alias LIG
select 4
use PRODUIT alias PRO
REPONSE = 'N'
do while REPONSE = 'N'
    ?'
    ?'
    ?'1 : Tous les renseignements concernant tous les produits.'
    ?'2 : Pour chaque LIGNECOM de chaque commande d'un CLIENT '
    ?'    spécifié afficher les informations sur ce CLIENT '
    ?'    cette commande le PRODUIT et le prix facturer'
    ?'3 : Tous les renseignements concernant tous les CLIENTS '
    ?'    d'une certaine LOCALITE ayant passé au moins une'
    ?'    COMMANDE'
    ?'4 : Tous les clients ayant commandés un certain produit'
    ?'5 : Sortie'
    accept 'Choisissez (1 à 5) dans le menu : ' to REPONSE
    if REPONSE = '1'
        ?'Numéro      Libellé      Quantité disponible'
        ?'-----'
        select PRO
        goto top
        do while .not. eof()
            ?PRO->NPRO
            ??'      '
            ??PRO->LIBELLE
            ??PRO->QSTOCK
        select PRO
        skip
        enddo
        ?'-----'
    REPONSE = 'N'
endif
if REPONSE = '2'
    accept 'Numéro de client : ' to NUM
    select COM
    goto top
    do while .not. eof()
        locate next 10000 for NCLI = NUM
        if eof()
            exit
        endif
        XCOM = recno()
    
```

```

select LIG
goto top
do while .not. eof()
    locate next 10000 for NCOM = COM->NCOM
    if eof()
        exit
    endif
    XLIG = recno()
    select PRO
    locate for NPRO = LIG->NPRO
    if .not. eof()
        ??'Client : '
        ??COM->NCLI
        ??' - Comm : '
        ??COM->NCOM
        ??COM->DATE
        ??' - Produit : '
        ??LIG->QCOM
        ??'x'
        ??LIG->NPRO
        ??' - Total : '
        ??LIG->QCOM*PRIX
    endif
    select LIG
    goto XLIG + 1
enddo

select COM
goto XCOM + 1
enddo
REPONSE = 'N'
endif
if REPONSE = '3'
accept 'Localité du client : ' to LOC
select CLI
goto top
do while .not. eof()
    select COM
    N = 0
    goto top
    do while .not. eof()
        locate next 10000 for NCLI = CLI->NCLI
        if eof()
            exit
        endif
        XCOM = recno()
        N = N + 1
        select COM
        goto XCOM + 1
    enddo
    if N <> 0
        if CLI->LOCALITE = LOC
            ?CLI->NCLI

```

```

        ??' '
        ??CLI->NOM
        ??CLI->ADRESSE
    endif
endif
select CLI
skip
enddo
REPONSE = 'N'
endif
if REPONSE = '4'
accept 'Numéro du produit : ' to NUM
select PRO
goto top
do while .not. eof()
    locate next 10000 for NPRO = NUM
    if eof()
        exit
    endif
    XPRO = recno()
    select LIG
    goto top
    do while .not. eof()
        locate next 10000 for NPRO = PRO->NPRO
        if eof()
            exit
        endif
        XLIG = recno()
        select COM
        locate for NCOM = LIG->NCOM
        if .not. eof()
            select CLI
            locate for NCLI = COM->NCLI
            if .not. eof()
                ?PRO->LIBELLE
                ??' '
                ??CLI->NOM
                ??' '
                ??CLI->ADRESSE
            endif
        endif
    endif
    select LIG
    goto XLIG + 1
enddo
select PRO
goto XPRO + 1
enddo
REPONSE = 'N'
endif
enddo
close databases

```

III.8.5. Exécution du texte dBase

. do menu.prg

MENU PRINCIPAL

- ```

1 : Tous les renseignements concernant tous les produits.
2 : Pour chaque LIGNECOM de chaque commande d un CLIENT
 spécifié afficher les informations sur ce CLIENT
 cette commande le PRODUIT et le prix à facturer
3 : Tous les renseignements concernant tous les CLIENTS
 d une certaine LOCALITE ayant passe au moins une
 COMMANDE
4 : Tous les clients ayant commandés un certain produit
5 : Sortie

```

Choisissez (1 à 5) dans le menu : 1

| Numéro | Libellé              | Quantité disponible |
|--------|----------------------|---------------------|
| PS222  | PL. SAPIN 200x20x2   | 1220                |
| PH222  | PL. HETRE 200x20x2   | 782                 |
| CS464  | CHEV. SAPIN 400x6x4  | 450                 |
| CS264  | CHEV. SAPIN 200x6x41 | 2690                |
| CS262  | CHEV. SAPIN 200x6x2  | 45                  |
| PA60   | POINTE ACIER 60 (1K) | 134                 |
| PA45   | POINTE ACIER 45 (1K) | 580                 |

```

PS222 PL. SAPIN 200x20x2 1220
PH222 PL. HETRE 200x20x2 782
CS464 CHEV. SAPIN 400x6x4 450
CS264 CHEV. SAPIN 200x6x41 2690
CS262 CHEV. SAPIN 200x6x2 45
PA60 POINTE ACIER 60 (1K) 134
PA45 POINTE ACIER 45 (1K) 580

```

Choisissez (1 à 5) dans le menu : 2

Numéro de client : C400

|                            |         |             |           |           |       |
|----------------------------|---------|-------------|-----------|-----------|-------|
| Client : C400 - Comm : 179 | 12/3/85 | - Produit : | 20xPA60   | - Total : | 1900  |
| Client : C400 - Comm : 179 | 12/3/85 | - Produit : | 60xCS262  | - Total : | 4500  |
| Client : C400 - Comm : 184 | 27/3/85 | - Produit : | 120xCS464 | - Total : | 26400 |
| Client : C400 - Comm : 184 | 27/3/85 | - Produit : | 20xPA45   | - Total : | 2100  |
| Client : C400 - Comm : 186 | 2/4/85  | - Produit : | 3xPA45    | - Total : | 315   |

Choisissez (1 à 5) dans le menu : 3

Localité du client : Liege

|      |          |                      |
|------|----------|----------------------|
| B512 | GILLET   | 14, r. de l'Ete      |
| F011 | PONCELET | 17, Clos des Erables |

Choisissez (1 à 5) dans le menu : 4

Numéro du produit : CS464

|                     |          |                      |
|---------------------|----------|----------------------|
| CHEV. SAPIN 400x6x4 | VANBIST  | 180, r. Florimont    |
| CHEV. SAPIN 400x6x4 | FERARD   | 65, r. du Tertre     |
| CHEV. SAPIN 400x6x4 | PONCELET | 17, Clos des Erables |
| CHEV. SAPIN 400x6x4 | GILLET   | 14, r. de l'Ete      |

## CONCLUSIONS

L'objectif de ce mémoire était de développer un outil d'aide à la conception d'applications dBase. Un outil qui, malgré la complexité des programmes de gestion, rend la conception d'une base de données et la programmation orientée base de données simple et efficace.

Pour ce faire nous avons présenté une démarche de conception de base de données qui réduit la complexité par fragmentation du problème. L'outil permet d'introduire une base de données selon le modèle Entité-Association et la description d'une collection de procédures rédigées en LDA. A partir de ces descriptions il est possible de produire automatiquement des programmes dBase III. L'outil rend la programmation orientée base de données plus simple et permet une mise au point rapide des programmes dBase.

Le générateur dBase est fait de telle façon qu'il est très simple et peu coûteux d'étendre la grammaire du langage procédural LDA et qu'il ne faut rien ajouter pour vérifier la syntaxe des règles de grammaire supplémentaires.

Outre les fonctions d'aide à la spécification, les fonctions de génération de texte dBase III et de rapport dBase III, il serait intéressant d'ajouter à l'atelier les fonctions suivantes :

- une génération automatique des écrans en dBase;
- une fonction de modification des spécifications qui permet un contrôle de cohérence;
- une fonction de validation des spécifications qui vérifie la conformité des spécifications à dBase III;
- une fonction de transformation des spécifications qui rend les spécifications conformes à dBase III.

Toutes ces fonctions rendraient l'atelier encore plus utile et efficace.

## BIBLIOGRAPHIE

[AHO-ULLMAN,86]

AHO, ULLMAN, Compilers, principles, techniques and tools, Addison Wesley Series in Computer Science, 1986.

[ANSI,75]

ANSI/X3/SPARC, Study group on Data Base Management System, Interim report, FDT (ACM-SIGMOD Bulletin), n°2,1975

[BENCI,74]

BENCI, BODART, CABANES, DEHENEFFE, HAINAUT, LEROY, RANDON, SAVOYSKI, THULY, Introductory report, in proc. of the Intern. Workshop on Data Structure Models for Information Systems, Presses Universitaires de Namur, 1975

[BOD-PIG,83/88]

BODART, PIGNEUR, Conception assistée des applications informatiques - 1. Etude d'opportunité et analyse conceptuelle, Masson, Paris, 1983/1988.

[CHEN,76]

CHEN, The Entity Relationship Model - Toward a unified view of data, ACM TODS,1976

[DATE,86]

DATE, An introduction to database systems, vol.1, Addison-Wesley,1986

[HAINAUT,86]

HAINAUT, Conception assistée des applications informatiques - 2. Conception de la base de données. Masson, Paris, 1986.

[HAINAUT,86]

HAINAUT, Aspects théoriques et pratiques des modèles Entité-Association, Institut d'Informatique, Facultés Universitaires de Namur, 1988.

[HAINAUT,86]

HAINAUT, Application de l'informatique à la résolution de problèmes, Institut d'Informatique, Facultés Universitaires de Namur, 1988.

[HAINAUT,87]

HAINAUT, NDBS A simple data base system for small computers, Institut d'Informatique, Facultés Universitaires de Namur, 1988.

[MACDONALD,84]

MACDONALD, Information Engineering to math fourth generation tools, Maidenhead

[PARNAS,74]

PARNAS, On a Buzzword : Hierarchical Structure, Information Processing 74 North Holland, 1974.

[SIMPSON,87]

SYMPSON, Introduction à dBase III, Sybex, Paris, 1987.

[TARDIEU,83]

TARDIEU, ROCHFELD, COLETTI, Méthode MERISE : principes et outils, Editions d'Organisation, Paris, 1983.

[ULLMAN,82]

ULLMAN, Principles of database systems, Computer Science Press, 1982.

[WEINBERG,80]

WEINBERG, Structured analysis, Prentice Hall, 1980

[ZELKOVITZ,79]

ZELKOVITZ,SHAW,GANNON, Principles of Software Engineering and Design, Prentice Hall, 1979.

## ANNEXES

## Le fichier CLIENT

### Description du fichier CLIENT.DBF

| Field | Field name | Type      | Width | Dec |
|-------|------------|-----------|-------|-----|
| 1     | NCLI       | Character | 4     |     |
| 2     | NOM        | Character | 12    |     |
| 3     | ADRESSE    | Character | 20    |     |
| 4     | LOCALITE   | Character | 12    |     |

Chaque enregistrement décrit un client. NCLI représente son numéro identifiant, NOM son nom, ADRESSE son adresse, LOCALITE sa localité de résidence.

### Contenu du fichier CLIENT.DBF

| Record # | NCLI | NOM        | ADRESSE              | LOCALITE |
|----------|------|------------|----------------------|----------|
| 1        | B112 | HANSENE A. | 23, a. Dumont        | Andenne  |
| 2        | C123 | MERCIER    | 25, r. Lemaitre      | Namur    |
| 3        | B332 | MONTI      | 112, R. Neuve        | Stavelot |
| 4        | FO10 | TOUSSAINT  | 5, r. Godefroid      | Andenne  |
| 5        | K111 | VANBIST    | 180, r. Fforimont    | Amay     |
| 6        | S127 | VANDERKA   | 3, av. des Roses     | Namur    |
| 7        | B512 | GILLET     | 14, r. de l'Ete      | Liege    |
| 8        | BO62 | GOFFIN     | 72, r. de la Gare    | Namur    |
| 9        | C400 | FERARD     | 65, r. du Tertre     | Andenne  |
| 10       | C003 | AVRON      | 8, ch. de la Cure    | Liege    |
| 11       | K729 | NEUMAN     | 40, r. Bransart      | Liege    |
| 12       | F011 | PONCELET   | 17, Clos des Erables | Liege    |
| 13       | L422 | FRANCK     | 60, r. de Wepion     | Namur    |
| 14       | S712 | GUILLAUME  | 14a, ch. des         | Wepion   |
| 15       | D063 | NADOY      | 201, bvd du Nord     | Liege    |
| 16       | F400 | JACOB      | 78, ch. du Moulin    | Louvain  |

## Le fichier COMMANDE

### Description du fichier COMMANDE.DBF

| Field | Field name | Type      | Width | Dec |
|-------|------------|-----------|-------|-----|
| 1     | NCOM       | Character | 4     |     |
| 2     | NCLI       | Character | 4     |     |
| 3     | DATE       | Character | 8     |     |

Chaque enregistrement décrit une commande passée par un client. NCOM représente le numéro du identifiant la commande, NCLI représente le numéro du client qui a passé la commande, DATE représente la date à laquelle la commande a été passée.

### Contenu du fichier COMMANDE.DBF

| Record # | NCOM | NCLI | DATE     |
|----------|------|------|----------|
| 1        | 178  | K111 | 12/03/85 |
| 2        | 179  | C400 | 12/03/85 |
| 3        | 182  | S127 | 14/03/85 |
| 4        | 184  | C400 | 27/03/85 |
| 5        | 185  | F011 | 02/03/85 |
| 6        | 186  | C400 | 02/04/85 |
| 7        | 188  | B412 | 03/04/85 |

## Le fichier LIGNECOM

### Description du fichier LIGNECOM.DBF

| Field | Field name | Type      | Width | Dec |
|-------|------------|-----------|-------|-----|
| 1     | NCOM       | Character | 4     |     |
| 2     | NPRO       | Character | 5     |     |
| 3     | QCOM       | Numeric   | 4     |     |

Un enregistrement décrit une ligne d'une commande spécifiant un produit. NCOM précise le numéro de la commande à laquelle la ligne appartient, NPRO spécifie le produit commandé et QCOM représente la quantité commandée.

### Contenu du fichier LIGNECOM.DBF

| Record # | NCOM | NPRO  | QCOM |
|----------|------|-------|------|
| 1        | 178  | CS464 | 25   |
| 2        | 179  | PA60  | 20   |
| 3        | 179  | CS262 | 60   |
| 4        | 182  | PA60  | 30   |
| 5        | 184  | CS464 | 120  |
| 6        | 184  | PA45  | 20   |
| 7        | 185  | PA60  | 15   |
| 8        | 185  | PS222 | 600  |
| 9        | 185  | CS464 | 260  |
| 10       | 186  | PA45  | 3    |
| 11       | 188  | PA60  | 70   |
| 12       | 188  | PH222 | 92   |
| 13       | 188  | CS464 | 180  |
| 14       | 188  | PA45  | 22   |

## Le fichier PRODUIT

### Description du fichier PRODUIT.DBF

| Field | Field name | Type      | Width | Dec |
|-------|------------|-----------|-------|-----|
| 1     | NPRO       | Character | 5     |     |
| 2     | LIBELLE    | Character | 20    |     |
| 3     | PRIX       | Numeric   | 5     |     |
| 4     | QSTOCK     | Numeric   | 6     |     |

Un enregistrement décrit un produit qui peut être commandé par un client. NPRO représente le numéro identifiant le produit, LIBELLE donne le nom conventionnel du produit, PRIX indique le prix à l'unité et QSTOCK la quantité restant en stock.

### Contenu du fichier PRODUIT.DBF

| Record # | NPRO  | LIBELLE              | PRIX | QSTOCK |
|----------|-------|----------------------|------|--------|
| 1        | CS262 | CHEV. SAPIN 200x6x2  | 75   | 45     |
| 2        | CS264 | CHEV. SAPIN 200x6x41 | 120  | 2690   |
| 3        | CS464 | CHEV. SAPIN 400x6x4  | 220  | 450    |
| 4        | PA45  | POINTE ACIER 45 (1K) | 105  | 580    |
| 5        | PA60  | POINTE ACIER 60 (1K) | 95   | 134    |
| 6        | PH222 | PL. HETRE 200x20x2   | 230  | 782    |
| 7        | PS222 | PL.SAPAIN 200x20x2   | 185  | 1220   |

**CONCEPTION ET REALISATION  
D'UN  
ATELIER DE DEVELOPPEMENT  
DBASE-III**

**Mode d'emploi et implémentation de l'atelier**

Mémoire présenté par  
**Jan Wasiak**  
en vue de l'obtention du titre de  
**Licencié et Maître en Informatique**

**Promoteur : Professeur Jean-Luc Hainaut**

## Table des matières

---

1. Mode d'emploi de l'atelier
2. Implémentation de l'atelier
  - 2.1. Représentation de la grammaire
  - 2.2. Représentation de la table d'analyse
  - 2.3. Les rubans de traduction
  - 2.4. Le texte PASCAL de l'atelier

Annexes

## 1. MODE D'EMPLOI

Pour lancer l'exécution de l'atelier il suffit de taper ADAdB.

L'écran principal de l'atelier est le suivant :

1. Gestion des bases de données.
2. Consultation des bases de données.
3. Rapport de la base de données.
4. Génération de programme dBase.
5. Quitter l'atelier.

Votre choix :

L'option 1 exécute un gestionnaire d'écran qui permet d'introduire et de modifier une description d'une base de données en projet.

L'option 3 permet de produire un rapport d'une base de données en projet. En annexe 1 se trouve le rapport de la base de données développée dans la première partie.

Lorsque l'utilisateur veut générer un programme dBase, il doit exécuter la quatrième option. Une "invite" apparaîtra alors pour lui demander le nom du fichier LDA à traduire. A partir de ce moment, la génération est lancée.

Quand elle est terminée, un des messages suivants apparaît :

La compilation à réussie.

Le texte dBase se trouve dans le fichier NOM\_FICHER.PRG.

Ou bien :

En cas d'erreur (erreur lors de la compilation ou le cas où le programme n'appartient pas à un schéma) l'atelier affichera le type d'erreur et la ligne dans le texte LDA où cette erreur se situe. Un fichier d'erreur NOM\_FICHER.ERR reprend ce qui a été affiché.

En cas de réussite, il n'y a plus qu'à lancer le programme dBase dans l'interpréteur dBase et prendre bonne note des résultats.

Avant de lancer la génération dBase, il faut donc d'abord créer un fichier contenant le texte rédigé en LDA. Ce fichier peut être créé à l'aide de n'importe quel traitement de texte.

L'option 5 permet de sortir de l'atelier.

## 2. L'IMPLEMENTATION DE L'ATELIER

### 2.1. Représentation de la grammaire

Toute règle de grammaire est constituée de symboles de la grammaire. Ces derniers sont représentés par un type énuméré (GRAMMAIRE\_SYMBOLE). A chaque symbole de la grammaire correspond donc un nombre entier qui désigne son rang dans cette énumération.

D'où l'idée de représenter toute règle de la grammaire par une suite de nombres entiers ordonnés de la façon suivante :

- toutes les suites ont une longueur maximale déterminée par la longueur de la plus longue règle de la grammaire;

- le premier nombre entier de la suite correspond au rang du symbole apparaissant dans la partie gauche de la règle de grammaire en question;

- le rang des symboles apparaissant dans la partie droite de la règle de grammaire est placé dans la suite de telle façon que la règle soit justifiée à droite : le dernier élément de la suite correspond au rang du dernier symbole de cette règle de grammaire. Ce procédé crée inévitablement dans la suite des éléments dépourvus de toute valeur, d'autant plus si la longueur de la règle est inférieure à la longueur maximale. A ces éléments de la suite sera associé le rang du symbole "NULL" conseillé dans le type énuméré.

La table complète des règles de la grammaire LDA mise sous cette forme se trouve en annexe 2 (GRAMFILE.DTA).

### 2.2. La table d'analyse

La table d'analyse est vue comme une matrice dont les indices représentent le rang des éléments énumérés dans le type énuméré GRAMMAIRE\_SYMBOLE.

La table d'analyse est sauvée dans un fichier permanent SRTFILE.DTA. Cette dernière décision a été prise pour éviter à chaque exécution de reconstruire la table d'analyse. La table d'analyse générée par l'atelier à partir de la grammaire définie au point précédent se trouve en annexe 3.

### 3.3. Les rubans de traduction

Les rubans élémentaires de traduction sont stockés dans le fichier FSFILE.DTA. Ce fichier se trouve en annexe 4.

### 3.4. Le texte PASCAL de l'atelier

const

```
{ ****
{ * Constantes propres à la grammaire * }
{ **** }
```

```
{ Le nombre de règles de production }
nb_production = 45;
```

```
{ La longueur, en symboles, de la plus longue règle de production }
long_max_prod = 16;
```

```
{ ****
{ * Constantes propres au compilateur * }
{ **** }
```

```
{ Hauteur maximale de la pile }
long_max_pile = 600;
```

```
{ Taille maximale du tank de traduction }
taille_tank = 31000;
```

```
{ Taille maximale de la table de travail }
max_table = 20;
```

type

```
{ ****
{ * Types liés à la grammaire * }
{ **** }
```

```
{ ****
{ * * }
{ * Les symboles de la grammaire : * }
{ * * }
{ * On distingue : - les variables syntaxiques, notées VS_, * }
{ * - les variables terminaux qui comprennent des * }
{ * variables terminales, notées VT_, * }
{ * - le symbole $, noté S_ENTREE_SYM, * }
{ * - un élément "neutre", noté VS_NULL. * }
{ * * }
{ * Convention : on énumère les symboles dans l'ordre suivant : * }
{ * * }
{ * VS_NULL, les variables syntaxiques, les variables * }
{ * terminales, les mots réservés et enfin S_ENTREE_SYM * }
{ * * }
{ **** }
```

SYMBOLE\_GRAMMAIRE =

```
(VS_NULL,
VS_INSTRUCTION, VS_ASSIGNMENT, VS_SEQUENCE, VS_EXPR, VS_TERM, VS_FACTOR,
VS_SEQ, VS_READ, VS_FOR, VS_CONDITION, VS_WRITE, VS_WRITELN,
```

```

VS_ASSIGNATION_ENTITE, VS_CONDITION_LDA, VS_INTERVAL, VS_WHILE, VS_IF,
VT_WRITE, VT_POINT, VT_SEMI_COLON, VT_ID, VT_STR, VT_ASSIGN, VT_PLUS,
VT_MOINS, VT_MULTIPLIC, VT_DIVISION, VT_PARENTHESE_OUVRANTE,
VT_PARENTHESE_FERMANTE, VT_READ, VT_FOR, VT_ENDFOR, VT_DO, VT_COMMA,
VT_EQUALITY, VT_COLON, VT_IF, VT_THEN, VT_ENDIF, VT_ELSE, VT_INEQUALITY,
VT_LESS_THEN, VT_GREATER_THEN, VT_LESS_OR_EQUAL, VT_GREATER_OR_EQUAL,
VT_WRITELN, VT_REL_OP, VT_WHILE, VT_ENDWHILE, VT_ENTIER, S_ENTREE_SYM);

{ Les variables syntaxiques de la grammaire }
gram_var_sym = VS_INSTRUCTION..VS_IF;

{ Les symboles terminaux }
gram_ter_sym = VT_WRITE..S_ENTREE_SYM;

{*****
{* Types liés au compilateur *}
{*****}

{ Forme d'une règle de production : le numéro, la partie gauche et
 la partie droite

 Convention : la partie droite est cadrée à droite et éventuellement
 précédée de VS_NULL's.
 Justification : cela permet d'avoir une structure constante.
}

REGLE_PRODUCTION =
 RECORD
 NUMERO_PROD : INTEGER;
 PART_GAUCHE : GRAM_VAR_SYM;
 PART_DROITE : ARRAY[0..long_max_prod] of SYMBOLE_GRAMMAIRE
 end;

SRE = (SHIFTACT,REDUCEACT,ERRORACT);

STRING_SPECIFIER = record
 LOWER : integer;
 UPPER : integer;
end;

{ Forme d'un TOKEN : le numéro de la ligne dans le fichier SOURCE,
 son type, et sa position dans le tank
}

TOKEN = RECORD
 LINE : integer;
 KIND : GRAM_TER_SYM;
 LIMITS : STRING_SPECIFIER;
end;

NODE_KIND = (TERM_SYMBOL, STANDARD, FINAL_STR,STANDARDDB);

{ Forme d'un pointeur vers un noeud de l'arbre de traduction
}

```

```
PTR = ^TREE_NODE;
```

```
{ Forme d'un noeud de l'arbre de traduction : pointer vers le
 noeud frère, pointeur vers le noeud fils, et le type de
 l'élément contenant le noeud
}
```

```
TREE_NODE = record
 FRERE : PTR;
 FILS : PTR;
 CASE KIND : NODE_KIND OF
 TERM_SYMBOL : (TERMINAL : TOKEN);
 STANDARD : (NON_TERMINAL : GRAM_TER_SYM);
 STANDARDB : (NON_TERMINALB : GRAM_VAR_SYM);
 FINAL_STR : (STR_BEGIN, STR_END : integer);
 END;
```

```
ROOT_PTR = PTR;
```

```
{ Les rubans "élémentaires" de traduction
}
```

```
SF_CONSTANTE = (SF_PLUS, SF_MOINS, SF_MULTIPLIC, SF_DIVISION, SF_WRITE,
 SF_ASSIGN, SF_READB, SF_READ, SF_SELECT, SF_GOTOTOP,
 SF_SKIP, SF_END, SF_DO, SF_USE, SF_ALIAS, SF_ARROW, SF_GOTO,
 SF_X, SF_INC, SF_IF, SF_EOF, SF_EXIT, SF_ENDIF, SF_LOCATE1,
 SF_LOCATE2, SF_RECNO, SF_NOT, SF_N, SF_N1, SF_N2,
 SF_PARENTHESE_OUVRANTE, SF_PARENTHESE_FERMANTE,
 SF_ELSE, SF_WHILE, SF_INEQUALITY, SF_LESS_THEN,
 SF_GREATER_THEN, SF_LESS_OR_EQUAL, SF_GREATER_OR_EQUAL,
 SF_ENDDO, SF_ENTER, SF_COUNT, SF_INPUT, SF_TABUL_P,
 SF_TABUL_M, SF_NULL);
```

```
{ Forme d'un élément de la pile : le symbole, le pointeur vers l'arbre
 de traduction associé
}
```

```
elem_pile = record
 SYMBOL : SYMBOLE_GRAMMAIRE;
 PTR_TO_TREE : PTR;
 LINE : integer;
end;
```

```
{*****
{* Types liés à l'analyse lexicale *
{*****
```

```
stlgl = string[80];
```

```
{ Le type d'un token
}
```

```
genre = (endline, endfile, word, space, int, symbol, comment);
```

```

tampon = record
 full : boolean;
 val : char;
end;

str = string[40];

{ ****
 * Types liés à l'analyse sémantique *
 **** }

{ La table de travail }

TABLE_TYPE = array[1..max_table] of record
 name : str;
 typ : str;
 origine : str;
 cible : str;
 bit : boolean;
end;

```

```

(*****
(*) Les variables (*)
(*****)

VAR

line_number, num_tank, num_table, nr_regle, i ,j , k : integer;

table : table_type;

TANK : record
 TOP : INTEGER;
 ELEM : ARRAY [1..taille_tank] OF CHAR
end;

TOPSYM : TOKEN;

{ structure de la pile }

PILE : RECORD
 TOP : integer;
 ELEM : array[0..long_max_pile] of ELEM_PILE;
end;

{ structure du tampon d'entrée }

ENTREE : RECORD
 TOP : integer;
 ELEM : array[0..long_max_pile] of ELEM_PILE;
end;

SOURCE, { fichier contenant le texte rédigé en ADL }
LISTER, { fichier contenant le texte dBase généré }
GRAMFILE, { fichier contenant la grammaire ADL }
SRTFILE, { fichier contenant la table d'analyse }
FSFILE { fichier contenant les rubans de traduction } : TEXT;

{ Un élément de la table d'analyse }

T : SRE;

{ La table d'analyse }

SRT : array[VS_INSTRUCTION..S_ENTREE_SYM,VT_WRITE..S_ENTREE_SYM] of SRE;

{ La table contenant la grammaire }

GRAMMAIRE : array[1..NB_PRODUCTION] of REGLE_PRODUCTION;

{ La table contenant les rubans de traduction }

```

```
FS : array[SF_CONSTANTE] of STRING_SPECIFIER;
{ Booléen indiquant si on a trouvé la variable distinguée }
VAR_DISTINGUEE : boolean;
p , q , r, s, ptr_l, ptr_r, ptr_tamp: PTR;
unchar : tampon;
typexp : genre;
f : text;
resultat, fichier, message, origine, cible,
TAMP2, TAMP1, TAMP, mot : stlgl;
typ1 : str;
INTER, CON, entite1, entite2, attribut_com, nom_attribut : stlgl;
con_ass, enter, continue_compilation,
yes, found, detecter_erreur : boolean;
```

```

{*****
{* Analyseur lexical *
{*
{* Ce module demande le nom du fichier à compiler. Si c'est un fichier *
{* existant et s'il appartient à une base de données ce module réalise *
{* l'analyse lexicale. C'est-à-dire le contenu du fichier est transformé*
{* en symboles terminaux qui sont stockés dans le tableaux ENTREE *
{*****

```

```
Function CAPITAL(ch : str) : str;
```

```
var i : integer;
```

```
begin
for i:=1 to length(ch) do ch[i] := upcase(ch[i]);
capital := ch;
end;
```

```
(*-----*)
```

```
Procedure FICHER_APPARTIENT_SCHEMA;
```

```
var PRO : TPROCEDUR;
 trouve : boolean;
```

```
begin
trouve := false;
dbfirst(PROCEDUR,PRO);
while dbfound and not trouve do
begin
if PRO.NOM_PROCEDURE = FICHER then trouve := true
else dbnext(PROCEDUR,PRO);
end;
if trouve = false then
begin
writeln;
writeln('Le fichier ',fichier,' n''appartient pas à un schéma');
continue_compilation := false;
end;
end;
```

```
(*-----*)
```

```
Procedure INITIALISATION (var source : text; var unchar : tampon);
```

```
type st20 = string[20];
```

```
var j : integer;
 PRO : TPROCEDUR;
```

```
begin
clrscr;
gotoxy(5,12);
write('Nom du fichier à traduire en dBase : ');
repeat
```

```

gotoxy(5,47);
readln(FICHER);
assign (SOURCE, FICHER);
{$I-}reset(SOURCE);{$I+}
if ioresult <> 0 then
 writeln('Le fichier ',fichier,' n''existe pas!');
until ioresult = 0;
writeln;
unchar.full:=false;
unchar.val:='?';
gotoxy(15,52);
FICHER := CAPITAL(FICHER);
RESULTAT := FICHER;
delete(RESULTAT,length(RESULTAT)-3,length(RESULTAT));
RESULTAT := RESULTAT + '.PRG';
write(' Compilation : ',FICHER,' -> ');
writeln(RESULTAT);
FICHER_APPARTIENT_SCHEMA;
end;

(*-----*)
Procedure GETTOKEN (var f:text; var st : stlgl ; var typexp : genre);

var x : char;
 i: integer;

procedure GETTOKEN2 ;

(* But : Distinguer les symboles faisant partie d'une même expression
(* à partir du fichier source f.
(* Unchar est une variable tampon qui mémorise le dernier caractère
(* lu si celui-ci fait partie de l'expression suivante (unchar.full
(* = true), mis à '?' sinon (unchar.full = false).

var tabu: char;

procedure CAS1;
(* Utilisation : L'expression est de type symbol.
(* Le dernier caractère de l'expression a été lu.
(* => unchar.val prend la valeur '?'.
(* unchar.full = false.

begin
 unchar.full:=false;
 unchar.val:='?';

```

```

 typexp:=symbol;
 st:=st+x
end;

procedure CASNUM;
(* Utilisation : Le caractère courant est un numérique.
* Résultat : L'expression est de type Int.
* L'expression finale est dans le paramètre st,
* résultat.
* Unchar.val = '?' si l'entier se termine en fin
* Unchar.full =false de ligne.
*
* Unchar.full =true et
* Unchar.val = premier caractère de l'expression
* suivante sinon.
begin
 typexp:=int;
 while ('0'<=x) and (x<='9') and not eoln(f) do
 begin
 st:=st+x;
 read (f,x)
 end;
 if eoln(f) and (('0'<=x) and (x<='9'))
 then begin st:=st+x;
 unchar.full:=false;
 unchar.val:='?'
 end
 else begin unchar.full:=true;
 unchar.val:=x
 end
 end;
end;

Procedure INEGALITE;
(* Utilisation : le dernier caractère lu est soit >, soit <. *)
(* Résultat : le contenu de st est >= ou <= ou <> et unchar.val='?' *)
(* unchar.full=false *)

(* > ou < et unchar.val = premier *)
(* caractère de l'expression suivante; unchar.full=true. *)

begin
 typexp:=symbol;
 st:=st+x;
 if not eoln(f)

```

```

then begin
 read (f,x);
 if (x='>') or (x='=')
 then begin st:=st+x;
 unchar.val:='?'; unchar.full:=false
 end
 else begin unchar.val:=x;
 unchar.full:=true
 end
 end;
end;

end;

Procedure POINT;
(* Utilisation : le dernier caractère lu est un point. *)
(* Résultat : le contenu de st est '..' et unchar.val='?' *)
(* unchar.full=false ou *)
(* le contenu de st est '.' et unchar.val = premier *)
(* caractère de l'expression suivante si eoln(f)=faux, *)
(* unchar.full = true. *)

begin
 typexp:=symbol;
 st:=st+x;
 if not eoln(f)
 then begin
 read (f,x);
 if x = '.' then begin
 st:=st+x;
 unchar.val:='?';
 unchar.full:=false
 end
 else begin
 unchar.val:=x;
 unchar.full:=true
 end
 end;
end;

end;

Procedure APOSTROPHE;
(* Utilisation : le dernier caractère lu est une apostrophe ('). *)
(* Résultat : l'expression est de type comment. *)
(* st = expression complète. *)
(* unchar.val=premier caractère de l'expression suivante *)
(* si celui-ci a été lu. *)
(* unchar.full=true *)
(* unchar.val= '?'et unchar.full=false sinon. *)

procedure commentaire;
begin
 while (x<>'') and (not eoln(f)) do
 begin
 st:=st+x;
 read(f,x)
 end
end

```

```

end;

begin
 typexp:=comment;
 st:=st+x;
 if not eoln(f)
 then begin
 read(f,x);
 commentaire;
 if (eoln(f) and (x='')) then begin st:=st+x;
 unchar.full:=false;
 unchar.val:='' end;

 if (not eoln(f) and (x=''))
 then begin
 st:=st+x;
 read (f,x);
 if (x='') then apostrophe
 else begin unchar.val:=x;
 unchar.full:=true
 end
 end
 end
 else
 begin
 if (eoln(f) and (x<>'') and (st[length(st)]<>''))
 then begin
 write('Erreur !!! Commentaire non achevé ');
 writeln(' en ligne ',line_number);
 DETECTER_ERREUR := true;
 repeat until keypressed;
 st:=st+x
 end;
 end
 end;
end;

```

```

Procedure LETTRE;
(* Utilisation : le dernier caractère lu est une lettre. *)
(* Résultat : l'expression est de type word. *)
(* st = expression complète. *)
(* unchar.val = premier caractère de l'expression suivante *)
(* et unchar.full = true. *)
(* unchar.val = '?' et unchar.full = false sinon. *)

```

```

begin
 typexp:=word;
 while (((('A'<= UpCase(x)) and (UpCase(x) <= 'Z'))
 or (('0'<=x) and (x<='9')) and (not eoln(f))) or (x='~')
 or (x='^') or (x='#')) do
 begin
 st:=st+x;
 read (f,x)
 end;
 if (('A'<=UpCase(x)) and (UpCase(x) <= 'Z'))
 or (('0'<=x) and (x<='9')) or (x = '~') or (x='^') or (x='#')

```

```

 then begin st:=st+x;
 unchar.full:=false;
 unchar.val:='?'
 end
 else begin unchar.full:=true;
 unchar.val:=x
 end
end;

end;

Procedure Asterix;
(* Utilisation : le dernier caractère lu est '*', il signale le début *)
(* d'un commentaire qui s'achève en fin de ligne. *)
(* Résultat : l'expression est de type comment. *)
(* st = l'expression complète. *)
(* unchar.val = '?', unchar.full:=false. *)

begin
 typexp:=comment;
 st:=st+x;
 while not eoln(f) do
 begin
 read(f,x);
 st:=st+x
 end;
 unchar.val:='?';
 unchar.full:=false
end;

Procedure Blanc;
(* Utilisation : dernier caractère lu est un blanc ou une tabulation. *)
(* Résultat : l'expression est de type space. *)
(* le contenu de st est équivalent au nombre de blancs *)
(* ou tabulations successifs. *)
(* unchar.full = true, *)
(* unchar.val = premier caractère de l'expression suivante. *)
(* Remarque : l'éditeur de texte Turbo Pascal supprime les blancs en *)
(* fin de ligne ! *)

var tabu:char;
begin
 tabu:=chr(9);
 typexp:=space;
 while ((x=' ') or (x=Tabu)) and (not eoln(f)) do
 begin
 st:=st+x;
 read (f,x)
 end;
 if (x=' ') or (x=tabu) then begin st:=st+x;
 unchar.val:='?';
 unchar.full:=false end
 else begin
 unchar.val:=x;

```

```

 unchar.full:=true end;

end;

(* Debut de la procedure Gettoken2 ----- *)
begin
if unchar.full =false then read (f,x)
 else x:=unchar.val;
 case x of
 ' ' : Blanc;
 ' ' : Blanc;
 (* Représentation par l'Editeur Pascal de *)
 (* la tabulation. *)
 'a'..'z': Lettre;
 'A'..'Z', '~', '#', '^': Lettre;
 '(' : cas1;
 '$' : cas1;
 ')' : cas1;
 ':' : begin
 typexp:=symbol;
 if eoln(f) then cas1
 else begin
 st:=st+x;
 read (f, x);
 if x= '='
 then cas1
 else begin unchar.val:=x;
 unchar.full:=true
 end
 end
 end
 end;
 '[' : cas1;
 ']' : cas1;
 '<', '>' : begin
 typexp:=symbol;
 if eoln(f) then inegalite
 else begin
 st:=st+x;
 read (f, x);
 if (x= '=') or (x='>')
 then cas1
 else begin unchar.val:=x;
 unchar.full:=true
 end
 end
 end
 end;
 '=' , '+' , '.' , ';' : cas1;
 '-' , '/' , '*' , '?' , ',' : cas1;
 ''' : apostrophe;
 '0'..'9': casnum;
 end
end;
end;

```

(\* debut de la procedure GETTOKEN -----\*)

```
begin
 st:='';
 if not eof(f)
 then begin
 if eoln(f)
 then begin
 if unchar.full = true
 then begin
 st:=st+unchar.val;
 case unchar.val of
 'a'..'z','~','#','^' : typexp:=word;
 'A'..'Z' : typexp:=word;
 '0'..'9' : typexp:=int;
 ' ' : typexp:=space;
 else typexp:=symbol;
 end;
 unchar.val:='?'; unchar.full:=false
 end
 else begin
 st:=chr(10)+chr(8);
 typexp:=endline;
 readln(f)
 end
 end
 else
 Gettoken2
 end
 else
 typexp:=endfile
 end;
end;
```

Procedure LEXICA;

```
var in_while : boolean;
 k:integer;
```

```
begin(LEXICA)
 line_number := 1;
 i := 0;
 num_tank := 1000;
 INITIALISATION(f,unchar);
 repeat
 repeat
 gettoken(f,mot,typexp);
 if typexp = endline then line_number := line_number + 1;
 until (typexp <> space) and (typexp <> endline);
 if typexp <> comment then mot := CAPITAL(mot);
 if (typexp <> endline) then
 begin
 i := i + 1;
 if (typexp = int) or (typexp = comment) or (typexp = word) then
 begin
```

```

if typexp = comment then ENTREE.ELEM[i].SYMBOL := VT_STR;
if typexp = int then ENTREE.ELEM[i].SYMBOL := VT_ENTIER
else
if mot = 'WRITE' then ENTREE.ELEM[i].SYMBOL:=VT_WRITE
else if mot = 'READ' then ENTREE.ELEM[i].SYMBOL:=VT_READ
else if mot = 'FOR' then ENTREE.ELEM[i].SYMBOL:=VT_FOR
else if mot = 'DO' then ENTREE.ELEM[i].SYMBOL:=VT_DO
else if mot = 'ENDFOR' then ENTREE.ELEM[i].SYMBOL:=VT_ENDFOR
else if mot = 'IF' then ENTREE.ELEM[i].SYMBOL:=VT_IF
else if mot = 'ENDIF' then ENTREE.ELEM[i].SYMBOL:=VT_ENDIF
else if mot = 'THEN' then ENTREE.ELEM[i].SYMBOL:=VT_THEN
else if mot = 'WHILE' then ENTREE.ELEM[i].SYMBOL:=VT_WHILE;
else if mot = 'ENDWHILE' then ENTREE.ELEM[i].SYMBOL:=VT_ENDIF
else if mot = 'ELSE' then ENTREE.ELEM[i].SYMBOL:=VT_ELSE
else if mot = 'WRITELN' then ENTREE.ELEM[i].SYMBOL:=VT_WRITELN
else ENTREE.ELEM[i].SYMBOL := VT_ID;
for k:=0 to length(mot) do TANK.ELEM[num_tank+k]:=mot[k+1];
new(p);
p^.kind:=TERM_SYMBOL;
if typexp = comment then p^.terminal.kind := VT_STR else
if typexp = int then p^.terminal.kind := VT_ENTIER else
p^.terminal.kind := VT_ID;
p^.terminal.line := line_number;
p^.terminal.limits.lower:=num_tank;
if length(mot) = 1 then
 p^.terminal.limits.upper := p^.terminal.limits.lower
else p^.terminal.limits.upper:=num_tank+length(mot)-1;
p^.frere:=nil;
p^.fils:=nil;
new(q);
q^.kind:=STANDARD;
q^.kind:=STANDARD;
if typexp = comment then q^.terminal.kind := VT_STR else
if typexp = int then q^.terminal.kind := VT_ENTIER else
if mot = 'write' then q^.terminal.kind := VT_WRITE else
if mot = 'read' then q^.terminal.kind := VT_READ else
q^.terminal.kind := VT_ID;
q^.non_terminal:= VT_ID;
q^.fils:=p;
q^.frere:=nil;
num_tank:=num_tank+length(mot);
p:=q;
entree.elem[i].ptr_to_tree:=p;
end
else
begin
if mot = '-' then ENTREE.ELEM[i].SYMBOL := VT_MOINS else
if mot = '*' then ENTREE.ELEM[i].SYMBOL := VT_MULTIPLIC else
if mot = '/' then ENTREE.ELEM[i].SYMBOL := VT_DIVISION else
if mot = '+' then ENTREE.ELEM[i].SYMBOL := VT_PLUS else
if mot = ';' then ENTREE.ELEM[i].SYMBOL := VT_SEMI_COLON else
if mot = '.' then ENTREE.ELEM[i].SYMBOL := VT_POINT else
if mot = ',' then ENTREE.ELEM[i].SYMBOL := VT_COMMA else
if mot = '=' then ENTREE.ELEM[i].SYMBOL := VT_EQUALITY else
if mot = '(' then ENTREE.ELEM[i].SYMBOL := VT_PARENTHESE_OUVRANTE els

```

```

if mot = ')' then ENTREE.ELEM[i].SYMBOL := VT_PARENTHESE_FERMANTE else
if mot = ':' then ENTREE.ELEM[i].SYMBOL := VT_COLON else
if mot = '>' then ENTREE.ELEM[i].SYMBOL := VT_GREATER_THEN else
if mot = '<' then ENTREE.ELEM[i].SYMBOL := VT_LESS_THEN else
if mot = ':' then ENTREE.ELEM[i].SYMBOL := VT_ASSIGN else
if mot = '>=' then ENTREE.ELEM[i].SYMBOL := VT_GREATER_OR_EQUAL else
if mot = '<=' then ENTREE.ELEM[i].SYMBOL := VT_LESS_OR_EQUAL else
if mot = '<>' then ENTREE.ELEM[i].SYMBOL := VT_INEQUALITY else
if typexp <> endfile then
writeln('Erreur de syntaxe en ',line_number);
end;
entree.elem[i].line:=line_number;
end;
until (typexp = endfile) or (i >= long_max_pile);
if i >= long_max_pile then continue_compilation := false
else ENTREE.ELEM[i].SYMBOL:=S_ENTREE_SYM;
end;{LEXICA}

```

```

Procedure MESSAGE_ERREUR(message : stlgl; nr_erreur : integer);

{ Cette procédure affiche à l'écran et écrit dans un fichier
 "FICHIER.MSG" le type d'erreur rencontré lors de la compilation.
 "nr_erreur" désigne le numéro de l'erreur et "message" désigne une
 information sur la cause de l'erreur.
}

begin
DETECTER_ERREUR := true;
write('Erreur de syntaxe dans le programme ', fichier);
writeln(' à la ligne ', line_number);
case nr_erreur of
 1 : writeln('Identifiant inconnu !');
 2 : writeln('Variables de type incompatibles !');
 3 : writeln(message, ' est un attribut inconnu !');
 4 : writeln(message, ' est un type d'entité inconnu !');
 5 : writeln(message, ' est un identifiant inconnu !');
 6 : writeln(message, ' est un type d'association inconnu !');
 7 : writeln(message, ' n'appartient pas au type d'entité évalué !');
 8 : writeln('Le texte est trop long !');
 10 : writeln('Le fichier ', message, ' n'appartient à aucune base !');
 11 : writeln(message, 'Type d'association incorrecte !');
end;
repeat until keypressed;
end;

Function PILE_PLEINE : boolean;

begin
PILE_PLEINE := (PILE.TOP = long_max_pile);
end;

Function DE_PILE(n : integer) : ROOT_PTR;

{ Cette fonction prend le n-ième élément de la pile }

begin
DE_PILE := pile.elem[pile.top-n].ptr_to_tree;
end;

Procedure CREER_STANDARD(v : SYMBOLE_GRAMMAIRE; h : PTR;
 var p : PTR);

{ Cette procédure crée un noeud de l'arbre de traduction
 contenant une forme de phrase
}

begin
new(p);
p^.kind := standard;
p^.non_terminal := v;
p^.frere := nil;
p^.fils := h;
ptr_l := p;

```

```

end;

Procedure LIEN_DE_FRERE(a, b : PTR);

{ Cette procédure crée un lien de frère }

begin
a^.frere := b;
end;

Procedure REDUIRE_PILE(n : integer; v : SYMBOLE_GRAMMAIRE; node : PTR);

{ Cette procédure se charge d'effectuer la réduction de la pile.
 Elle dépile la pile de n éléments et met le symbole v sur la
 pile.
}

begin
pile.top:=pile.top-n;
pile.elem[pile.top].symbol:=v;
pile.elem[pile.top].ptr_to_tree:=node;
end;

Procedure CREER_SF_NOEUD(s : SF_CONSTANTE;
 var p : PTR);

{ Cette procédure crée une feuille de l'arbre de traduction.
 Le pointeur p pointe vers le début et la fin du tableau
 qui contient les traductions élémentaires.
}

begin
new(p);
p^.kind := FINAL_STR;
p^.frere := nil;
p^.fils := nil;
p^.str_begin := FS[s].lower;
p^.str_end := FS[s].upper;
end;

Procedure GARNIR_TOPSYM;

{ Cette procédure met à jour le sommet de la pile }

begin
ENTREE.TOP := ENTREE.TOP + 1;
TOPSYM.KIND := ENTREE.ELEM[ENTREE.TOP].SYMBOL;
line_number := ENTREE.ELEM[ENTREE.TOP].LINE;
end;

Procedure SHIFT_TOPSYM;

{ Cette procédure met le contenu du tampon d'entrée "ENTREE" au sommet
 de la pile s'il y a encore assez de place dans celle-ci; sinon,
 l'erreur est notée.
}

```

Ce nouvel élément de la pile pointe vers un noeud contenant la "traduction".

```

}

begin
if not PILE_PLEINE then
begin
PILE.TOP := PILE.TOP + 1;
PILE.ELEM[PILE.TOP].SYMBOL := TOPSYM.KIND;
PILE.ELEM[PILE.TOP].PTR_TO_TREE := ENTREE.ELEM[ENTREE.TOP].PTR_TO_TREE;
end;
end;

```

Procedure PTR\_IN\_HEAP( p : PTR; var q : PTR);

var s : PTR;

```

begin
p:=p^.fils;
new(s);
s^.kind:=TERM_SYMBOL;
s^.terminal.limits.lower:= p^.terminal.limits.lower;
s^.terminal.limits.upper:=p^.terminal.limits.upper;
s^.frere:=nil;
s^.fils:=nil;
new(q);
q^.kind:=STANDARD;
q^.terminal.kind := VT_ID;
q^.non_terminal:= VT_ID;
q^.fils:=p;
q^.frere:=nil;
s:=q;
end;

```

Procedure MOT\_IN\_HEAP(mot : stlgl; var q : PTR);

var k : integer;

```

begin
for k:=0 to length(mot) do TANK.ELEM[num_tank+k]:=mot[k+1];
new(p);
p^.kind:=TERM_SYMBOL;
p^.terminal.limits.lower:=num_tank;
if length(mot) = 1 then
p^.terminal.limits.upper := p^.terminal.limits.lower
else p^.terminal.limits.upper:=num_tank+length(mot)-1;
p^.frere:=nil;
p^.fils:=nil;
new(q);
q^.kind:=STANDARD;
q^.terminal.kind := VT_ID;
q^.non_terminal:= VT_ID;
q^.fils:=p;
q^.frere:=nil;
num_tank:=num_tank+length(mot);

```

```
p:=q;
end;
```

```
Procedure PTR_TO_STRING(p : PTR; var mot : stlgl);
```

```
{ Cette procédure met un élément mot dans le tank. q
 est le pointeur qui pointe vers cet élément.
}
```

```
var i : integer;
```

```
begin
mot := '';
p:=p^.fils;
for i:=p^.terminal.limits.lower to p^.terminal.limits.upper do
mot := mot + TANK.ELEM[i];
line_number := p^.terminal.line;
end;
```

```
Procedure CHERCHER_REFERENCE(entitel,entite2 : stlgl;
 var attribut_com : stlgl;
 var found : boolean);
```

```
{ Cette procedure se charge de trouver l'attribut "attribut_com" du
 type d'entité d'entité cible "entitel" que référence le type d'entité
 origine "entite2". Si la procédure ne trouve pas cet attribut "found"
 est mis à faux.
}
```

```
var ENT : TENTITE;
 ATT : TATTRIBUT;
 PRO : TPROCEDUR;
 SCH : TSHEMA;
 trouve : boolean;
```

```
begin
dbopen('METABASE');
trouve := false;
found := false;
dbfirst(PROCEDUR,PRO);
while dbfound do
 begin
 if PRO.NOM_PROCEDURE = FICHIER then
 begin
 dbfpath(SCH,PRO,-APPARTENANCE);
 while dbfound do
 begin
 dbfpath(ENT,SCH,COLLECTIONNE_E);
 while dbfound and not found do
 begin
 if ENT.NOM_ENTITE = entite2 then
 begin
 found := true;
 dbfpath(ATT,ENT,CARACTERISTIQUE);
 while dbfound and not trouve do
```

```

begin
 if ATT.ID_ATTRIBUT = 'O' then
 begin
 trouve:=true;
 attribut_com := ATT.NOM_ATTRIBUT;
 end
 else dbnpath(ATT,ENT,CARACTERISTIQUE);
 end;
 end
 else dbnpath(ENT,SCH,COLLECTIONNE_E);
 end;
 dbnpath(SCH,PRO,-APPARTENANCE);
 end;
end
else dbnext(PROCEDUR,PRO);
end;
if found and trouve then found := true;
dbclose;
end;

Procedure EXISTE_ENTITE(p : PTR; var mot : stlgl;
 var found : boolean);

{ Cette procédure vérifie si le type d'entité "p" existe dans
 la base de données FICHIER. Si ce n'est pas le cas "found" est
 mis à faux.
}

var ENT : TENTITE;
 PRO : TPROCEDUR;
 SCH : TSHEMA;

begin
 dbopen('METABASE');
 found := false;
 PTR_TO_STRING(p,mot);
 dbfirst(PROCEDUR,PRO);
 while dbfound do
 begin
 if PRO.NOM_PROCEDURE = FICHIER then
 begin
 dbfpath(SCH,PRO,-APPARTENANCE);
 while dbfound do
 begin
 dbfpath(ENT,SCH,COLLECTIONNE_E);
 while dbfound and not found do
 begin
 if ENT.NOM_ENTITE = mot then found := true
 else dbnpath(ENT,SCH,COLLECTIONNE_E);
 end;
 dbnpath(SCH,PRO,-APPARTENANCE);
 end;
 end
 else dbnext(PROCEDUR,PRO);
 end;
 end
 end
 end
 end
 end
 end;
end;

```

```
dbclose;
end;
```

```
Procedure EXISTE_ASSOCIATION(p : PTR; var origine : stlgl;
 var cible : stlgl;
 var found : boolean);
```

```
{ Cette procédure vérifie si le type d'association "p" existe dans
 la base de données FICHIER. Si ce n'est pas le cas "found" est mis
 à faux. "origine" et "cible" sont respectivement l'entité origine et
 cible du type d'association.
}
```

```
var ASS : TASSOCIATION;
 ROL : TROLE;
 PRO : TPROCEDUR;
 SCH : TSHEMA;
 trouve : boolean;
```

```
begin
dbopen('METABASE');
found := false;
trouve := false;
PTR_TO_STRING(p,mot);
dbfirst(PROCEDUR,PRO);
while dbfound and not trouve do
begin
if PRO.NOM_PROCEDURE = FICHIER then
begin
dbfpath(SCH,PRO,-APPARTENANCE);
while dbfound do
begin
dbfpath(ass,sch,collectionne_a);
while dbfound do
begin
if ASS.NOM_ASSOCIATION = mot then
begin
found := true;
dbfpath(ROL,ASS,LIEN);
while dbfound do
begin
origine := ROL.ORIGINE_ROLE;
cible := ROL.CIBLE_ROLE;
dbnpath(ROL,ASS,LIEN);
end;
end;
dbnpath(ass,sch,collectionne_a);
end;
dbnpath(SCH,PRO,APPARTENANCE);
end
end;
dbnext(PROCEDUR,PRO);
end;
dbclose;
end;
```

```

Procedure ATTRIBUT_APPART_ENTITE(entitel, nom_attribut : str;
 var found : boolean);

{ Cette procédure vérifie si l'attribut "nom_attribut" appartient
 au type d'entité "entitel". Si ce n'est pas le cas "found" est mis
 à faux.
}

var ENT : TENTITE;
 ATT : TATTRIBUT;
 PRO : TPROCEDUR;
 SCH : TSHEMA;
 etrouve, atrouve : boolean;

begin
 dbopen('METABASE');
 found:=false;
 etrouve:=false;
 atrouve:=false;
 dbfirst(PROCEDUR,PRO);
 while dbfound do
 begin
 if PRO.NOM_PROCEDURE = FICHER then
 begin
 dbfpath(SCH,PRO,-APPARTENANCE);
 while dbfound do
 begin
 dbfpath(ENT,SCH,COLLECTIONNE_E);
 while dbfound and not etrouve do
 begin
 if ENT.NOM_ENTITE = entitel then
 begin
 dbfpath(ATT,ENT,CARACTERISTIQUE);
 while dbfound and not atrouve do
 begin
 if ATT.NOM_ATTRIBUT = nom_attribut
 then atrouve:=true
 else dbnpath(ATT,ENT,CARACTERISTIQUE);
 end;
 etrouve := true;
 end
 else dbnpath(ENT,SCH,COLLECTIONNE_E);
 end;
 dbnpath(SCH,PRO,-APPARTENANCE);
 end;
 end
 else dbnext(PROCEDUR,PRO);
 end;
 if etrouve and atrouve then found := true;
 end;
 dbclose;
end;

Procedure EXISTE_ATTRIBUT(nom_attribut : stlgl;
 var entitel : stlgl;

```

```

 var found : boolean);

{ Cette procédure vérifie si "nom_attribut" est un attribut de la
 base de données FICHER. Si ce n'est pas le cas "found" est mis
 à faux. "entitel" est le type d'entité associé à cet attribut.
}

var ATT : TATTRIBUT;
 ENT : TENTITE;
 trouve : boolean;
 type_attribut : char;

begin
dbopen('METABASE');
found := false;
trouve := false;
dbfirst(ATTRIBUT,ATT);
while dbfound and not found do
 begin
 if ATT.NOM_ATTRIBUT = nom_attribut then
 begin
 found := true;
 dbfpath(ENT,ATT,-CARACTERISTIQUE);
 case upcase(ATT.TYPE_ATTRIBUT) of
 'S' : typ1 := 'str';
 'I' : typ1 := 'ent';
 end;
 while dbfound and not trouve do
 begin
 entitel := ENT.NOM_ENTITE;
 trouve := true;
 dbnpath(ENT,ATT,-CARACTERISTIQUE);
 end
 end
 else dbnext(ATTRIBUT,ATT);
 end;
 if found and trouve then found := true;
 dbclose;
end;

Procedure TROUVER_VARIA(nom : stlgl; var i : integer;
 var found : boolean);

{ Cette procédure cherche la variable nom dans la table de
 travail. Si cet élément n'appartient pas à cette table
 found est mis à faux. i est le numéro de l'élément dans
 la table.
}

var mot : stlgl;

begin
found := false;
i := 1;
while (found = false) and (i <= num_table) do

```

```

if table[i].name = nom then found := true else i := i + 1;
end;

```

```

Procedure DEPOSER_DS_TABLE(entitel : str;
 mot : str);

```

```

{ Cette procédure met le type d'entite entitel et son surnom
 dans la table de travail.
}

```

```

var num : integer;
 trouve : boolean;

```

```

begin
TROUVER_VARIA(entitel,num,trouve);
if trouve = false then
 begin
 num_table:=num_table + 1;
 table[num_table].name:=entitel;
 table[num_table].typ:=mot;
 end
else MESSAGE_ERREUR('',20);
end;

```

```

Procedure DEPOSER_TABLE(nom : str;
 mot : str);

```

```

{ Cette procédure met la variable et son type dans la table de travail.}

```

```

begin
num_table:=num_table + 1;
table[num_table].name:=nom;
table[num_table].typ:=mot;
end;

```

```

Procedure TROUVER_DS_TABLE2(mot : str; var i : integer;
 var found : boolean);

```

```

{ Cette procédure cherche l'élément mot dans la table de
 travail. Si cet élément n'appartient pas à cette table
 found est mis à faux. i est le numéro de l'élément dans
 la table.
}

```

```

begin
found := false;
i := 1;
while (found = false) and (i <= num_table) do
if table[i].typ = mot then found := true else i := i + 1;
end;

```

```

Procedure SET_TO_TRUE(nom : str);

```

```

var i : integer;
 find : boolean;

```

```

begin
TROUVER_DS_TABLE2(nom,i,find);
table[i].bit := true;
end;

Procedure ACCES_PAR_ASSOCIATION(p ,q, s, r: PTR;
 var attribut_com : stlgl;
 var found, typ : boolean);

var founda, found_common, foundc, find : boolean;
 t, num : integer;
 mot, mot1, mot2, nom : stlgl;
 origine, cible : stlgl;

Procedure RELATION_DS_TABLE(mot, nom : stlgl; var found : boolean);

begin
TROUVER_DS_TABLE2(mot,num,find);
if find = false then
 begin
 table[num].bit := true;
 DEPOSER_DS_TABLE(mot1,mot);
 found := true;
 end
else begin
 if table[num].name = mot1
 then begin
 found := true;
 table[num].bit := true;
 end
 else found := false;
 end;
if found <> false then
 begin
 TROUVER_DS_TABLE2(nom,num,find);
 if find = false then
 begin
 DEPOSER_DS_TABLE(mot2,nom);
 found := true;
 end
 else begin
 if table[num].name = mot2
 then found := true
 else found := false;
 end;
 end;
 end;
end;

begin
typ := true;
founda := false;
foundc := false;
found_common := false;
PTR_TO_STRING(p,mot);

```

```

EXISTE_ASSOCIATION(p,origine,cible,founda);
if founda = true then
begin
PTR_TO_STRING(q,mot);
PTR_TO_STRING(r,mot1);
PTR_TO_STRING(s,mot2);
CHERCHER_REFERENCE(cible,origine,attribut_com,found_common);
if found_common = true then
begin
if (origine = mot) and (mot2 <> cible) then
begin
typ := false;
RELATION_DS_TABLE(mot,cible,found);
end;
if (cible = mot) and (mot2 <> origine) then
begin
typ := true;
RELATION_DS_TABLE(mot,origine,found);
end
end
else MESSAGE_ERREUR('',11);
end
else MESSAGE_ERREUR(mot,6); ;
end;

Procedure ACCES_PAR_CLE(p ,q, s, r: PTR; var entitel : stlgl;
 var found : boolean);

var find, exist, founda : boolean;
 i, num : integer;
 nom, nom1, nom2, nom_entite : stlgl;

begin
founda := false;
PTR_TO_STRING(p,nom2);
PTR_TO_STRING(r,nom1);
PTR_TO_STRING(q,nom);
TROUVER_DS_TABLE2(nom,num,find);
if find = false then
begin
EXISTE_ENTITE(q,nom,exist);
if exist = true then
begin
DEPOSER_DS_TABLE(nom1,nom);
SET_TO_TRUE(nom);
found := true;
end
else
begin
found := false;
MESSAGE_ERREUR(nom,4)
end;
end
else begin
table[num].bit := true;

```

```

 if table[num].name = nom1
 then found := true
 else begin
 found := false;
 MESSAGE_ERREUR(nom2,7)
 end;
 end;
if found = true then
begin
 EXISTE_ATTRIBUT(nom2,nom_entite,founda);
 if founda = false then
 begin
 found := false;
 MESSAGE_ERREUR(nom2,7)
 end
 else
 begin
 if nom_entite <> nom then MESSAGE_ERREUR(nom2,7) else
 found := true;
 end
 end;
end;

Procedure ACCES_SEQUENTIEL(p, q : PTR; var found : boolean);

var i, j, num : integer;
 nom1, nom2 : stlgl;
 exist, find : boolean;

begin
 i:=1;
 j:=1;
 exist := false;
 find:=false;
 PTR_TO_STRING(p,nom2);
 PTR_TO_STRING(q,nom1);
 TROUVER_DS_TABLE2(nom1,num,find);
 if find = false then
 begin
 EXISTE_ENTITE(q,nom1,exist);
 if exist = true then
 begin
 found := true;
 DEPOSER_DS_TABLE(nom2,nom1);
 SET_TO_TRUE(nom2);
 table[num].bit := true;
 end
 else found := false;
 end
 else
 begin
 if (table[num].typ = nom1) and (table[num].name = nom2)
 then
 begin
 found := true;

```

```

 table[num].bit := true;
 end
 else found:=false;
end;
end;

Procedure verifier_arbre;

{ Cette procédure vérifie si dans l'arbre avec racine "p"
 il n'y pas d'éléments de types incompatibles, ou inconnus.
 Si c'est le cas un message d'erreur est notée. Le type "typ1"
 désigne le type de l'expression.
}

var ent, entite, entite2, nom : stlgl;
 find : boolean;

procedure verifier(p:PTR);

var i : integer;

begin
if (p <> nil) then
 begin
 verifier(p^.fils);
 case p^.kind of
 FINAL_STR, TERM_SYMBOL, STANDARDB:
 begin
 case ord(p^.terminal.kind) of
 22 : begin
 if typ1='' then typ1:='str'
 else begin
 if typ1 <> 'str' then MESSAGE_ERREUR(typ1,2)
 else typ1:='str';
 end;
 end;
 50 : begin
 if entite2<>' ' then MESSAGE_ERREUR(entite2,2)
 else
 begin
 if typ1='' then typ1:='ent'
 else begin
 if typ1 <> 'ent' then MESSAGE_ERREUR(typ1,2)
 else typ1:='ent';
 end;
 end;
 end;
 21 : begin
 ent:=' ';
 for i:=p^.terminal.limits.lower to
 p^.terminal.limits.upper
 do ent := ent + tank.elem[i];
 TROUVER_VARIA(ent,i,found);
 if found = false then
 begin

```

```

 if entite2='' then
 begin
 entite2:=ent;
 end
 else
 begin
 EXISTE_ATTRIBUT(ent,nom,find);
 if find = false
 then MESSAGE_ERREUR(ent,3)
 else
 begin
 TROUVER_DS_TABLE2(nom,i,find);
 if find=true then
 begin
 if table[i].name<>entite2 then
 MESSAGE_ERREUR(ent,2)
 end
 end
 else
 begin
 DEPOSER_TABLE(entite2,nom);
 entite2:='';
 end
 end;
 end;
 end;
 end
 else begin
 if (table[i].typ <> 'str') and (table[i].typ <> 'ent')
 then begin
 entite2:=ent;
 entite:=table[i].typ;
 end
 else
 begin
 if typ1 = '' then typ1:=table[i].typ else
 if typ1<> table[i].typ then MESSAGE_ERREUR(typ1,2);
 end;
 end;
 end;
end;

 end;
end;

 end;
verifier(p^.frere);
end;

end;

begin
 nom:='';
 entite2:='';
 typ1:='';
 verifier(p);
end;

Procedure HEAD_SEQ(p : PTR;var q,s :PTR);

{ Cette procédure construit un arbre qui correspond à une

```

```

partie de la traduction de la boucle énumérative, accès
séquentiel.(select <var_entite>
 goto top
 do while .not. eof()
)
}

var ptr_s3, ptr_s4,
 ptr_s0, ptr_s1, ptr_s2 : PTR;

begin
 CREER_SF_NOEUD(SF_SELECT,s);
 CREER_SF_NOEUD(SF_GOTOTOP,ptr_s1);
 CREER_SF_NOEUD(SF_DO,ptr_s2);
 CREER_SF_NOEUD(SF_ENTER,ptr_s3);
 CREER_SF_NOEUD(SF_ENTER,q);
 LIEN_DE_FRERE(s,p);
 LIEN_DE_FRERE(p,ptr_s3);
 LIEN_DE_FRERE(ptr_s3,ptr_s1);
 LIEN_DE_FRERE(ptr_s1,ptr_s2);
 LIEN_DE_FRERE(ptr_s2,q);
end;

Procedure HEAD_SEQ_SEC(s,p : PTR;var q : PTR);

var ptr_s3, ptr_s4,
 ptr_s0, ptr_s1, ptr_s2 : PTR;

begin
 CREER_SF_NOEUD(SF_GOTOTOP,ptr_s1);
 CREER_SF_NOEUD(SF_DO,ptr_s2);
 CREER_SF_NOEUD(SF_ENTER,ptr_s3);
 CREER_SF_NOEUD(SF_ENTER,ptr_s4);
 CREER_SF_NOEUD(SF_ENTER,q);
 LIEN_DE_FRERE(s,p);
 LIEN_DE_FRERE(p,ptr_s3);
 LIEN_DE_FRERE(ptr_s3,ptr_s1);
 LIEN_DE_FRERE(ptr_s1,ptr_s4);
 LIEN_DE_FRERE(ptr_s4,ptr_s2);
 LIEN_DE_FRERE(ptr_s2,q);
end;

Procedure IN_IF(s,p : PTR; var q : PTR);

{ Cette procédure construit un arbre qui correspond à
 if eof()
 exit
 endif
 X <var_entite> = recno()
}

var ptr_s1, ptr_s2, ptr_s3,
 ptr_s7, ptr_s8, ptr_s9 : PTR;

begin

```

```

CREER_SF_NOEUD(SF_IF,ptr_s1);
CREER_SF_NOEUD(SF_EOF,ptr_s2);
CREER_SF_NOEUD(SF_EXIT,ptr_s3);
CREER_SF_NOEUD(SF_ENDIF,ptr_s8);
CREER_SF_NOEUD(SF_X,ptr_s7);
CREER_SF_NOEUD(SF_RECNO,q);
LIEN_DE_FRERE(p,ptr_s1);
LIEN_DE_FRERE(ptr_s1,ptr_s2);
LIEN_DE_FRERE(ptr_s2,ptr_s3);
LIEN_DE_FRERE(ptr_s3,ptr_s8);
LIEN_DE_FRERE(ptr_s8,ptr_s7);
LIEN_DE_FRERE(ptr_s7,s);
LIEN_DE_FRERE(s,q);
end;

```

```

Procedure IN_END(p,q,s : PTR);

```

```

{ Cette procédure construit un arbre qui correspond à
 select <var_entite>
 goto X <var_entite> + 1
 enddo
}

```

```

var ptr_s1, ptr_s2, ptr_s3,
 ptr_s4, ptr_s5, ptr_s6, ptr_s7, ptr_s8, ptr_s9, ptr_s10 : PTR;

```

```

begin
CREER_SF_NOEUD(SF_GOTO,ptr_s2);
CREER_SF_NOEUD(SF_END,ptr_s5);
CREER_SF_NOEUD(SF_SELECT,ptr_s1);
CREER_SF_NOEUD(SF_INC,ptr_s4);
CREER_SF_NOEUD(SF_X,ptr_s3);
CREER_SF_NOEUD(SF_ENTER,ptr_s6);
CREER_SF_NOEUD(SF_ENTER,ptr_s7);
CREER_SF_NOEUD(SF_ENTER,ptr_s8);
CREER_SF_NOEUD(SF_TABUL_M,ptr_s10);
LIEN_DE_FRERE(s,ptr_s10);
LIEN_DE_FRERE(ptr_s10,ptr_s1);
LIEN_DE_FRERE(ptr_s1,p);
LIEN_DE_FRERE(p,ptr_s6);
LIEN_DE_FRERE(ptr_s6,ptr_s2);
LIEN_DE_FRERE(ptr_s2,ptr_s3);
LIEN_DE_FRERE(ptr_s3,q);
LIEN_DE_FRERE(q,ptr_s4);
LIEN_DE_FRERE(ptr_s4,ptr_s7);
LIEN_DE_FRERE(ptr_s7,ptr_s5);
LIEN_DE_FRERE(ptr_s5,ptr_s8);
end;

```

```

Procedure IN_END_SEC(p,q,s : PTR; var r : PTR);

```

```

var ptr_s1, ptr_s2, ptr_s3,
 ptr_s4, ptr_s5, ptr_s6, ptr_s7, ptr_s8, ptr_s9 : PTR;

```

```

begin

```

```

CREER_SF_NOEUD(SF_GOTO,ptr_s2);
CREER_SF_NOEUD(SF_END,ptr_s5);
CREER_SF_NOEUD(SF_SELECT,ptr_s1);
CREER_SF_NOEUD(SF_INC,ptr_s4);
CREER_SF_NOEUD(SF_X,ptr_s3);
CREER_SF_NOEUD(SF_ENTER,ptr_s6);
CREER_SF_NOEUD(SF_ENTER,ptr_s7);
CREER_SF_NOEUD(SF_ENTER,ptr_s8);
LIEN_DE_FRERE(s,ptr_s1);
LIEN_DE_FRERE(ptr_s1,p);
LIEN_DE_FRERE(p,ptr_s6);
LIEN_DE_FRERE(ptr_s6,ptr_s2);
LIEN_DE_FRERE(ptr_s2,ptr_s3);
LIEN_DE_FRERE(ptr_s3,q);
LIEN_DE_FRERE(q,ptr_s4);
LIEN_DE_FRERE(ptr_s4,ptr_s7);
LIEN_DE_FRERE(ptr_s7,ptr_s5);
LIEN_DE_FRERE(ptr_s5,r);
end;

```

```

Procedure CONDITION_ASSOCIATION(q, r : PTR; var s : PTR;
 var entitel : stlgl);

```

```

var nr, nrl, cas : integer;
 find : boolean;
 nom_entite, nom_association, nom_var,nom_varb, nom,
 nom_commun : stlgl;
 mot : stlgl;

```

```

begin
EXISTE_ASSOCIATION(r,origine,cible,found);
PTR_TO_STRING(r,nom_association);
if found = true then
begin
CHERCHER_REFERENCE(cible,origine,nom_commun,found);
if found = true then
begin
PTR_TO_STRING(q,nom_var);
PTR_TO_STRING(s,nom_varb);
if ((nom_var = origine) and (nom_varb = cible))
then cas := 1;
if ((nom_varb = origine) and (nom_var = cible))
then cas := 2;
if ((nom_var = origine) and (nom_varb <> cible))
then cas := 3;
if ((nom_varb <> origine) and (nom_var = cible))
then cas := 4;
if ((nom_var <> origine) and (nom_varb = cible))
then cas := 5;
if ((nom_varb = origine) and (nom_var <> cible))
then cas := 6;
case cas of
1, 2 : begin
entitel := nom_commun;
TROUVER_DS_TABLE2(nom_var,i,find);

```

```

mot:=nom_var;
delete(nom_var,4,length(nom_var));
if find = false then
begin
DEPOSER_DS_TABLE(nom_var,mot);
SET_TO_TRUE(mot);
end;
MOT_IN_HEAP(nom_var,s);
TROUVER_DS_TABLE2(nom_varb,i,find);
mot:=nom_varb;
delete(nom_varb,4,length(nom_varb));
TAMP := nom_varb;
if find = false then
begin
DEPOSER_DS_TABLE(nom_varb,mot);
SET_TO_TRUE(mot);
end;
end;
3, 4 : begin
entitel := nom_commun;
TROUVER_DS_TABLE2(nom_var,i,find);
mot:=nom_var;
delete(nom_var,4,length(nom_var));
TAMP := nom_var;
if find = false then
begin
DEPOSER_DS_TABLE(nom_var,mot);
SET_TO_TRUE(mot);
end;
if cas = 3 then nom_var := cible else
nom_var := origine;
TROUVER_DS_TABLE2(nom_var,i,find);
if find = false then
begin
DEPOSER_DS_TABLE(nom_varb,nom_var);
SET_TO_TRUE(nom_var);
end;
end;
5, 6 : begin
entitel := nom_commun;
MOT_IN_HEAP(nom_var,s);
TROUVER_DS_TABLE2(nom_varb,i,find);
mot:=nom_varb;
delete(nom_varb,4,length(nom_varb));
TAMP := nom_varb;
if find = false then
begin
DEPOSER_DS_TABLE(nom_varb,mot);
SET_TO_TRUE(mot);
end;
if cas = 6 then nom_varb := cible else
nom_varb := origine;
TROUVER_DS_TABLE2(nom_varb,i,find);
if find = false then

```

```

begin
 DEPOSER_DS_TABLE(nom_var,nom_varb);
 SET_TO_TRUE(nom_varb);
end;
end
else MESSAGE_ERREUR('',20);
end;
end
else MESSAGE_ERREUR(nom_association,6);
tamp2:=tamp;
end
else MESSAGE_ERREUR(nom_association,6);
end;

Procedure TROUVER_REGLE_A_REDUIRE(var trouve : boolean;
 var nr_regle: integer);

{ Cette procédure cherche la règle à réduire. nr_regle
 est le numéro de la règle dans le tableaux GRAMMAIRE. trouve
 est mis à faux si cette procédure ne trouve pas de règle. }

var s,t, BORNE_INF, BORNE_SUP : integer;
 arret : boolean;
 i : integer;

begin
i:=1;
s:=PILE.TOP;
t:=long_max_prod;
trouve := false;
while (i<=nb_production) do
 begin
 if (GRAMMAIRE[i].PART_DROITE[t] = PILE.ELEM[s].SYMBOL) then
 begin
 BORNE_INF := i;
 i:=i+1;
 while (GRAMMAIRE[i].PART_DROITE[t] = PILE.ELEM[s].SYMBOL) do
 i:=i+1;
 BORNE_SUP := i;
 end
 else
 i := i + 1;
 end;
 BORNE_SUP := BORNE_SUP - 1;
 nr_regle := BORNE_INF;
 while (nr_regle <= BORNE_SUP) and (not trouve) do
 begin
 arret := false;
 t:=long_max_prod;
 s:=PILE.TOP;
 while not arret do
 begin
 if not arret then
 begin
 t:=t - 1;

```

```
 s:=s - 1;
 end;
 if (GRAMMAIRE[nr_regle].PART_DROITE[t] <>
 PILE.ELEM[s].SYMBOL)
 then arret := true;
 if (GRAMMAIRE[nr_regle].PART_DROITE[t] = VS_NULL) then
 begin
 trouve := true;
 arret := true;
 PILE.ELEM[s+1].SYMBOL:=GRAMMAIRE[nr_regle].PART_GAUCHE;
 end;
 end;
 if not trouve then nr_regle := nr_regle + 1;
end;

end;
```

```

{*****}
{* Le module CONSTRUCTEUR *}
{* *} *}
{* La fonction de ce module est de construire la table d'analyse, la *}
{* table contenant toutes les règles de production et la table con- *}
{* tenant les rubans de traduction. *}
{*****}

```

Procedure CONSTRUC;

```

{ Cette procédure construit la table d'analyse.
}

```

Procedure Build\_SRE;

```

type work_array = array[1..2,1..long_max_pile] of integer;
type_sre = char;

```

```

var shift_table : work_array;
 end_mark_shift : integer;
 reduce_table : work_array;
 end_mark_reduce : integer;
 SRE_TABLE : array[1..100,1..100] of type_sre;

```

```

{*****}

```

```

Procedure RULE_EXAM(line : integer; var init_pos : integer;
 var end_pos : integer);

```

```

begin
 end_pos := long_max_prod;
 init_pos := end_pos;
 while grammair[line].part_droite[init_pos] <> VS_NULL do
 init_pos := init_pos - 1;
 init_pos := init_pos + 1
 end;

```

```

{*****}

```

```

Function NOT_IN_TABLE(table : work_array; X : integer; Y : integer;
 last_column : integer) : boolean;

```

```

var i : integer;
 in_table : boolean;

```

```

begin
 in_table := false;
 i := 1;
 while (in_table = false) and (i <= last_column) do
 begin
 if table[1,i] = X
 then
 if table[2,i] = Y
 then
 in_table := true;

```

```

 i := i + 1;
 end;
 not_in_table := not(in_table);
end;

(*****)

Procedure Search_Shift;

var last_column : integer;
 line : integer;
 init_pos : integer;
 end_pos : integer;
 X : integer;
 Y : integer;
 L0 : integer;
 current_mark : integer;
 i : integer;

begin

 (* dollar shift < variable distinguée > *)

 shift_table [1,1] := ord(S_ENTREE_SYM);
 shift_table [2,1] := ord(VS_SEQUENCE);
 last_column := 1;

 (* Si Y ::= L0 ... Ln-1 alors L0 shift L1 .. *)

 for line := 1 to nb_production do
 begin
 RULE_EXAM(line,init_pos,end_pos);
 while init_pos < end_pos do
 begin
 X := ord(grammaire[line].part_droite[init_pos]);
 Y := ord(grammaire[line].part_droite[init_pos + 1]);
 if not_in_table (shift_table,X,Y,last_column)
 then
 begin
 last_column := last_column + 1;
 shift_table[1,last_column] := X;
 shift_table[2,last_column] := Y;
 end;
 init_pos := init_pos + 1;
 end;
 end;
 end;

 (* Si X shift Y et Y ::= L0 .. LN alors X shift L0 *)

 current_mark := 1;
 while current_mark <> last_column do
 begin
 Y := shift_table[2,current_mark];
 for line := 1 to nb_production do
 begin

```

```

 if ord(grammaire[line].part_gauche) = Y
 then
 begin
 RULE_EXAM(line,init_pos,end_pos);
 X := shift_table[1,current_mark];
 L0 := ord(grammaire[line].part_droite[init_pos]);
 if NOT_IN_TABLE(shift_table,X,L0,last_column)
 then
 begin
 last_column := last_column + 1;
 shift_table[1,last_column] := X;
 shift_table[2,last_column] := L0 ;
 end
 end;
 end;
 current_mark := current_mark + 1;
 end;

(* table finale des shift *)
(* On ne garde que les couples qui ont des symboles terminaux en
 deuxième position *)

end_mark_shift := 0;
for i := 1 to last_column do
begin
 if (shift_table[2,i] >= ord(VT_WRITE)) and
 (shift_table[2,i] <= ord(S_ENTREE_SYM))
 then
 begin
 end_mark_shift := end_mark_shift + 1;
 shift_table[1,end_mark_shift] := shift_table[1,i];
 shift_table[2,end_mark_shift] := shift_table[2,i];
 end;
 end;
end;

(*****)

Procedure Search_Reduce;

var last_column : integer;
 line : integer;
 init_pos : integer;
 end_pos : integer;
 X : integer;
 Y : integer;
 L0 : integer;
 Ln : integer;
 current_mark : integer;
 i : integer;

begin

 (* < variable distinguée > reduce dollar *)

```

```

reduce_table[1,1] := ord(VS_SEQUENCE);
reduce_table[2,1] := ord(S_ENTREE_SYM);
last_column := 1;

(* si X shift Y et X := L0 ... Ln alors Ln reduce Y *)

for i := 1 to end_mark_shift do
begin
 X := shift_table[1,i];
 Y := shift_table[2,i];
 for line := 1 to nb_production do
 begin
 if ord(grammaire[line].part_gauche) = X
 then
 begin
 Ln := ord(grammaire[line].part_droite[long_max_prod]);
 if NOT_IN_TABLE(reduce_table,Ln,Y,last_column)
 then
 begin
 last_column := last_column + 1;
 reduce_table[1,last_column] := Ln;
 reduce_table[2,last_column] := Y;
 end;
 end;
 end;
 end;
 end;
 end;

end;

(* Si X reduce Y et Y ::= L0 ... Ln alors X reduce L0 *)
(* Si X reduce Y et X ::= L0 ... Ln alors Ln reduce Y *)

current_mark := 1;
while current_mark <> last_column do
begin
 X := reduce_table[1,current_mark];
 Y := reduce_table[2,current_mark];
 for line := 1 to nb_production do
 begin
 if ord(grammaire[line].part_gauche) = Y
 then
 begin
 RULE_EXAM(line,init_pos,end_pos);
 L0 := ord(grammaire[line].part_droite[init_pos]);
 if NOT_IN_TABLE(reduce_table,X,L0,last_column)
 then
 begin
 last_column := last_column + 1;
 reduce_table[1,last_column] := X;
 reduce_table[2,last_column] := L0;
 end;
 end;
 end
 else
 begin
 if ord(grammaire[line].part_gauche) = X
 then
 begin

```

```

Ln := ord(grammaire[line].part_droite[long_max_prod]);
if NOT_IN_TABLE(reduce_table,Ln,Y,last_column)
then
 begin
 last_column := last_column + 1;
 reduce_table[1,last_column] := Ln;
 reduce_table[2,last_column] := Y;
 end;
end
end;
current_mark := current_mark + 1;
end;

(* table finale des reduce *)
(* on ne garde que les couples quie ont des symboles terminaux en
deuxième position *)

end_mark_reduce := 0;
for i := 1 to last_column do
 begin
 if (reduce_table[2,i] >= ord(VT_WRITE)) and
 (reduce_table[2,i] <= ord(S_ENTREE_SYM))
 then
 begin
 end_mark_reduce := end_mark_reduce + 1;
 reduce_table [1,end_mark_reduce] := reduce_table[1,i];
 reduce_table [2,end_mark_reduce] := reduce_table[2,i];
 end;
 end;
 end;
end;

(*****)

Procedure SAVE_in_FILE;

var i, j : integer;
 f : text;

begin
 assign(f,'SRTFILE.DTA');
 rewrite(f);
 for i := ord(VS_INSTRUCTION) to ord(S_ENTREE_SYM) do
 begin
 for j := ord(VT_WRITE) to ord(S_ENTREE_SYM) do
 begin
 write(f,SRE_TABLE[i,j]);
 end;
 writeln(f);
 end;
 end;
 close(f);
end;

(*****)

```

```

Procedure CREATE_SRE_TABLE;
var i,j : integer;
begin
 for i:= ord(VS_INSTRUCTION) to ord(S_ENTREE_SYM) do
 for j:=ord(VT_WRITE) to ord(S_ENTREE_SYM) do
 SRE_TABLE[i,j] := 'E';

 (* mettre les shift *)

 for i := 1 to end_mark_shift do
 SRE_TABLE[shift_table[1,i],shift_table[2,i]] := 'S';

 (* mettre les reduce *)

 for i := 1 to end_mark_reduce do
 SRE_TABLE[reduce_table[1,i],reduce_table[2,i]] := 'R';
 end;

 (*=====*)
begin
 search_shift;
 Search_reduce;
 Create_SRE_table;
 SAVE_in_FILE;
end;

Procedure BUILD_PRODUCTION;

{ Cette procédure lit le fichier GRAMFILE contenant la grammaire et la
 met dans le tableau GRAMMAIRE.
}

var prod : stlgl;
 t : SYMBOLE_GRAMMAIRE;
 p, ok : integer;
 j, i : integer;

begin
 for j:=1 to NB_PRODUCTION do
 begin
 repeat
 gettoken(GRAMFILE,PROD,typexp);
 until (typexp <> space) and (typexp <> endlene);
 t:=VS_NULL;
 val(prod,p,ok);
 while ord(t) <> p do
 begin
 t:=succ(t);
 end;
 if ord(t) = p then
 begin
 GRAMMAIRE[j].NUMERO_PROD := j;

```

```

GRAMMAIRE[j].PART_GAUCHE:=t;
for i:=0 to long_max_prod do
begin
repeat
gettoken(GRAMFILE,PROD,typexp);
until (typexp <> space) and (typexp <> endl);
t:=VS_NULL;
val(prod,p,ok);
while ord(t) <> p do
begin
t:=succ(t);
end;
if ord(t) = p then
GRAMMAIRE[j].PART_DROITE[i]:=t;
end;
end;
end;
end;

Procedure BUILD_FS;

{ Cette procédure lit le fichier FSFILE contenant les rubans de
traduction et les met dans le tableau FS.
}

var t : SF_CONSTANTE;
 n : integer;
 inp : stlgl;
 k : integer;

begin
k:=1;
n:=1;
t:=SF_PLUS;
repeat
mot := '';
inp := '';
repeat
repeat
gettoken(FSFILE,inp,typexp);
if inp = 'not' then inp:='.'+inp+'.';
if (inp <> '$') and (typexp <> endl) then mot:=mot + inp;
if typexp = endfile then exit;
until (typexp = endl) or (inp = '$');
until typexp <> endl;
if (typexp <> endfile) or (typexp <> endl) then
begin
FS[t].lower:=n;
if length(mot) = 1 then FS[t].upper:=FS[t].lower else
FS[t].upper:=n + length(mot)-1;
for k:= 1 to length(mot) do TANK.ELEM[n+k-1]:=mot[k];
n := n + length(mot);
t := succ(t);
end;
until t = SF_NULL;

```

```

end;

Procedure LECT_SRE;

{ Cette procédure lit le fichier SRTFILE contenant la table d'analyse
 et la met dans le tableau SRT.
}

var ch : char;
 i,j : integer;
 s : gram_ter_sym;
 t : SYMBOLE_GRAMMAIRE;

begin {BUILD_SRE}
ASSIGN(SRTFILE,'SRTFILE.DTA');
{$i-}reset(SRTFILE){$i+};
if ioresult <> 0 then writeln('Unknown SRE filename!');
t:=VS_INSTRUCTION;
while not eof(SRTFILE) do
begin
 s:=VT_WRITE;
 while not eoln(SRTFILE) do
 begin
 read(SRTFILE,ch);
 if ch = 'E' then SRT[t,s]:=ERRORACT;
 if ch = 'S' then SRT[t,s]:=SHIFTACT;
 if ch = 'R' then SRT[t,s]:=REDUCEACT;
 s:=succ(s);
 end;
 readln(SRTFILE);
t:=succ(t);
end;
close(SRTFILE);
end;

begin {corps du programme}
BUILD_PRODUCTION;
BUILD_FS;
BUILD_SRE;
LECT_SRE;
end;

```

```

{*****}
{* Le module TRADUCTEUR *}
{* *} *}
{* La fonction de ce module est de réaliser l'analyse syntaxique *}
{* et de construire l'arbre de traduction. *}
{*****}

```

Procedure ROUT\_SEM\_6;

```

var ptr_r0,
 ptr_l : PTR;

```

```

begin
ptr_r0 := de_pile(0);
p:=ptr_r0;
ptr_r0:=p^.fils;
CREER_STANDARD(VS_INSTRUCTION,ptr_r0,ptr_l);
REDUIRE_PILE(0,VS_INSTRUCTION,ptr_l);
end;

```

Procedure ROUT\_SEM\_7;

```

var ptr_r1, ptr_r2,
 ptr_s0, ptr_s1,
 ptr_l : PTR;

```

```

begin
ptr_r1 := de_pile(1);
ptr_r2 := de_pile(2);
LIEN_DE_FRERE(ptr_r2,ptr_r1);
CREER_STANDARD(VS_SEQUENCE,ptr_r2,ptr_l);
REDUIRE_PILE(2,VS_SEQUENCE,ptr_l);
p:=ptr_l;
end;

```

Procedure ROUT\_SEM\_8;

```

var ptr_r1,
 ptr_s0, ptr_s1,
 ptr_l : PTR;

```

```

begin
ptr_r1 := de_pile(1);
p:=ptr_r1;
ptr_r1:=p^.fils;
CREER_STANDARD(VS_SEQUENCE,ptr_r1,ptr_l);
REDUIRE_PILE(1,VS_SEQUENCE,ptr_l);
p:=ptr_l;
end;

```

Procedure ROUT\_SEM\_9;

```

var ptr_r1,
 ptr_r2,
 ptr_l : PTR;

```

```

begin
ptr_r1 := de_pile(1);
ptr_r2 := de_pile(2);
LIEN_DE_FRERE(ptr_r2,ptr_r1);
CREER_STANDARD(VS_SEQ,ptr_r2,ptr_l);
REDUIRE_PILE(2,VS_SEQ,ptr_l);
end;

```

Procedure ROUT\_SEM\_10;

```

var ptr_r1,
 ptr_l : PTR;

```

```

begin
ptr_r1 := de_pile(1);
p:=ptr_r1;
ptr_r1:=p^.fils;
CREER_STANDARD(VS_SEQ,ptr_r1,ptr_l);
REDUIRE_PILE(1,VS_SEQ,ptr_l);
end;

```

Procedure ROUT\_SEM\_11;

```

var ptr_r0, ptr_r2,
 ptr_s0, ptr_s1,
 ptr_l : PTR;
 found : boolean;
 i : integer;
 entitel : stlgl;

```

```

begin
ptr_r0 := de_pile(0);
ptr_r2 := de_pile(2);
CREER_SF_NOEUD(SF_ASSIGN,ptr_s0);
CREER_SF_NOEUD(SF_ENTER,ptr_s1);
LIEN_DE_FRERE(ptr_r2,ptr_s0);
LIEN_DE_FRERE(ptr_s0,ptr_r0);
LIEN_DE_FRERE(ptr_r0,ptr_s1);
p:=ptr_r0;
verifier_arbre;
PTR_TO_STRING(ptr_r2,entitel);
TROUVER_VARIA(entitel,i,found);
if found = false then DEPOSER_TABLE(entitel,typ1)
else if table[i].typ <> typ1 then MESSAGE_ERREUR(typ1,2);
CREER_STANDARD(VS_ASSIGNMENT,ptr_r2,ptr_l);
REDUIRE_PILE(2,VS_ASSIGNMENT,ptr_l);
end;

```

Procedure ROUT\_SEM\_12;

```

var ptr_r0, ptr_r1, ptr_r2,
 ptr_s0, ptr_s1,
 ptr_l : PTR;

```

```

begin
ptr_r0 := de_pile(0);
ptr_r1 := de_pile(1);
ptr_r2 := de_pile(2);
CREER_SF_NOEUD(SF_ENTER,ptr_s1);
LIEN_DE_FRERE(ptr_r2,ptr_r1);
LIEN_DE_FRERE(ptr_r1,ptr_r0);
LIEN_DE_FRERE(ptr_r0,ptr_s1);
p:=ptr_r2;
verifier_arbre;
CREER_STANDARD(VS_CONDITION,ptr_r2,ptr_l);
REDUIRE_PILE(2,VS_CONDITION,ptr_l);
end;

```

Procedure ROUT\_SEM\_13;

```
var ptr_s0, ptr_l : PTR;
```

```

begin
CREER_SF_NOEUD(SF_ASSIGN,ptr_s0);
CREER_STANDARD(VT_REL_OP,ptr_s0,ptr_l);
REDUIRE_PILE(0,VT_REL_OP,ptr_l);
end;

```

Procedure ROUT\_SEM\_14;

```
var ptr_s0, ptr_l : PTR;
```

```

begin
CREER_SF_NOEUD(SF_INEQUALITY,ptr_s0);
CREER_STANDARD(VT_REL_OP,ptr_s0,ptr_l);
REDUIRE_PILE(0,VT_REL_OP,ptr_l);
end;

```

Procedure ROUT\_SEM\_15;

```
var ptr_s0, ptr_l : PTR;
```

```

begin
CREER_SF_NOEUD(SF_LESS_THEN,ptr_s0);
CREER_STANDARD(VT_REL_OP,ptr_s0,ptr_l);
REDUIRE_PILE(0,VT_REL_OP,ptr_l);
end;

```

Procedure ROUT\_SEM\_16;

```
var ptr_s0, ptr_l : PTR;
```

```

begin
CREER_SF_NOEUD(SF_GREATER_THEN,ptr_s0);
CREER_STANDARD(VT_REL_OP,ptr_s0,ptr_l);
REDUIRE_PILE(0,VT_REL_OP,ptr_l);
end;

```

Procedure ROUT\_SEM\_17;

```
var ptr_s0, ptr_l : PTR;
```

```
begin
CREER_SF_NOEUD(SF_LESS_OR_EQUAL,ptr_s0);
CREER_STANDARD(VT_REL_OP,ptr_s0,ptr_l);
REDUIRE_PILE(0,VT_REL_OP,ptr_l);
end;
```

```
Procedure ROUT_SEM_18;
```

```
var ptr_s0, ptr_l : PTR;
```

```
begin
CREER_SF_NOEUD(SF_GREATER_OR_EQUAL,ptr_s0);
CREER_STANDARD(VT_REL_OP,ptr_s0,ptr_l);
REDUIRE_PILE(0,VT_REL_OP,ptr_l);
end;
```

```
Procedure ROUT_SEM_19;
```

```
var ptr_r0, ptr_r2,
 ptr_s0,
 ptr_l : PTR;
```

```
begin
ptr_r0 := de_pile(0);
ptr_r2 := de_pile(2);
CREER_SF_NOEUD(SF_PLUS,ptr_s0);
LIEN_DE_FRERE(ptr_r2,ptr_s0);
LIEN_DE_FRERE(ptr_s0,ptr_r0);
CREER_STANDARD(VS_EXPR,ptr_r2,ptr_l);
REDUIRE_PILE(2,VS_EXPR,ptr_l);
end;
```

```
Procedure ROUT_SEM_20;
```

```
var ptr_r0, ptr_r2,
 ptr_s0,
 ptr_l : PTR;
```

```
begin
ptr_r0 := de_pile(0);
ptr_r2 := de_pile(2);
CREER_SF_NOEUD(SF_MOINS,ptr_s0);
LIEN_DE_FRERE(ptr_r2,ptr_s0);
LIEN_DE_FRERE(ptr_s0,ptr_r0);
CREER_STANDARD(VS_EXPR,ptr_r2,ptr_l);
REDUIRE_PILE(2,VS_EXPR,ptr_l);
end;
```

```
Procedure ROUT_SEM_21;
```

```
var ptr_r0,
 ptr_l : PTR;
```

```

begin
ptr_r0 := de_pile(0);
p:=ptr_r0;
ptr_r0:=p^.fils;
CREER_STANDARD(VS_EXPR,ptr_r0,ptr_l);
REDUIRE_PILE(0,VS_EXPR,ptr_l);
end;

```

Procedure ROUT\_SEM\_22;

```

var ptr_r0, ptr_s0,
 ptr_r2,
 ptr_l : PTR;

```

```

begin
ptr_r0 := de_pile(0);
ptr_r2 := de_pile(2);
CREER_SF_NOEUD(SF_MULTIPLIC,ptr_s0);
LIEN_DE_FRERE(ptr_r2,ptr_s0);
LIEN_DE_FRERE(ptr_s0,ptr_r0);
CREER_STANDARD(VS_TERM,ptr_r2,ptr_l);
REDUIRE_PILE(2,VS_TERM,ptr_l);
end;

```

Procedure ROUT\_SEM\_23;

```

var ptr_r0, ptr_s0,
 ptr_r2,
 ptr_l : PTR;

```

```

begin
ptr_r0 := de_pile(0);
ptr_r2 := de_pile(2);
CREER_SF_NOEUD(SF_DIVISION,ptr_s0);
LIEN_DE_FRERE(ptr_r2,ptr_s0);
LIEN_DE_FRERE(ptr_s0,ptr_r0);
CREER_STANDARD(VS_TERM,ptr_r2,ptr_l);
REDUIRE_PILE(2,VS_TERM,ptr_l);
end;

```

Procedure ROUT\_SEM\_24;

```

var ptr_r0,
 ptr_l : PTR;

```

```

begin
ptr_r0 := de_pile(0);
p:=ptr_r0;
ptr_r0:=p^.fils;
CREER_STANDARD(VS_TERM,ptr_r0,ptr_l);
REDUIRE_PILE(0,VS_TERM,ptr_l);
end;

```

Procedure ROUT\_SEM\_25;

```
var ptr_r0,
 ptr_l : PTR;
```

```
begin
ptr_r0 := de_pile(0);
p:=ptr_r0;
ptr_r0:=p^.fils;
CREER_STANDARD(VS_FACTOR,ptr_r0,ptr_l);
REDUIRE_PILE(0,VS_FACTOR,ptr_l);
end;
```

```
Procedure ROUT_SEM_27;
```

```
var ptr_r2, ptr_r4, ptr_r7, ptr_r9,
 ptr_s0, ptr_s3, ptr_s2, ptr_s4, ptr_s1, ptr_s6,ptr_s7, ptr_s8,
 ptr_s9, ptr_s10, ptr_s11, ptr_s12, ptr_s13, ptr_s14, ptr_s15,
 ptr_s16,ptr_s18, ptr_l ,p , q, s, t, v: PTR;
i : integer;
find, exist : boolean;
nom : stlgl;
nom_attribut, nom_entite, nom_type : stlgl;
```

```
begin
ptr_r2 := de_pile(1);
ptr_r4 := de_pile(3);
ptr_r7 := de_pile(6);
PTR_TO_STRING(ptr_r7,TAMP1);
PTR_IN_HEAP(ptr_r7,ptr_tamp);
TROUVER_DS_TABLE2(tamp1,i,find);
if find = false then
begin
nom:=tamp1;
EXISTE_ENTITE(ptr_r7,tamp1,exist);
if exist = false then MESSAGE_ERREUR(tamp1,4)
else
begin
PTR_TO_STRING(ptr_r4,nom_attribut);
EXISTE_ATTRIBUT(nom_attribut,nom_entite,find);
if find = false then MESSAGE_ERREUR(nom_attribut,3) else
if tamp1 <> nom_entite then MESSAGE_ERREUR(nom_attribut,7) else
begin
nom_type := typ1;
p := ptr_r2;
verifier_arbre;
if nom_type <> typ1 then MESSAGE_ERREUR('',2)
else
begin
SET_TO_TRUE(nom);
delete(nom,4,length(nom));
DEPOSER_DS_TABLE(nom,tamp1);
tamp2:=nom;
end;
end;
end;
end;
```

```

 end
 else
 begin
 SET_TO_TRUE(tamp1);
 nom := table[i].name;
 exist := true;
 end;
if exist = true then
begin
CREER_SF_NOEUD(SF_TABUL_P,ptr_s1);
CREER_SF_NOEUD(SF_ASSIGN,ptr_s2);
CREER_SF_NOEUD(SF_ENTER,ptr_s3);
CREER_SF_NOEUD(SF_ENTER,ptr_s4);
LIEN_DE_FRERE(ptr_s1,ptr_r4);
LIEN_DE_FRERE(ptr_r4,ptr_s2);
LIEN_DE_FRERE(ptr_s2,ptr_r2);
CREER_STANDARD(VS_CONDITION_LDA,ptr_r4,ptr_l);
REDUIRE_PILE(6,VS_CONDITION_LDA,ptr_l);
end;
end;

```

Procedure ROUT\_SEM\_28;

```

var ptr_r1, ptr_r3, ptr_r5, ptr_r7,
 ptr_s0, ptr_s1, ptr_s2, ptr_s3, ptr_s4, ptr_s5, ptr_s6, ptr_s7,
 ptr_l : PTR;
 i, j:integer;
 mot:stlgl;
 find, exist : boolean;

```

```

begin
con_ass := true;
ptr_r1 := de_pile(1);
ptr_r3 := de_pile(3);
ptr_r5 := de_pile(5);
ptr_r7 := de_pile(7);
CREER_SF_NOEUD(SF_SELECT,ptr_s0);
CREER_SF_NOEUD(SF_ASSIGN,ptr_s2);
CREER_SF_NOEUD(SF_ENTER,ptr_s3);
CREER_SF_NOEUD(SF_ENTER,ptr_s7);
CREER_SF_NOEUD(SF_ARROW,ptr_s4);
CONDITION_ASSOCIATION(ptr_r5,ptr_r3,ptr_r1,entitel);
PTR_TO_STRING(ptr_r5,tamp1);
PTR_TO_STRING(ptr_r1,tamp);
MOT_IN_HEAP(entitel,ptr_s5);
MOT_IN_HEAP(entitel,ptr_s6);
LIEN_DE_FRERE(ptr_s5,ptr_s2);
LIEN_DE_FRERE(ptr_s2,ptr_r1);
LIEN_DE_FRERE(ptr_r1,ptr_s4);
LIEN_DE_FRERE(ptr_s4,ptr_s6);
CREER_STANDARD(VS_CONDITION_LDA,ptr_s5,ptr_l);
REDUIRE_PILE(5,VS_CONDITION_LDA,ptr_l);
end;

```

Procedure ROUT\_SEM\_29;

```

var ptr_r1, ptr_r3,
 ptr_s0, ptr_s1, ptr_s2, ptr_s3,
 ptr_l : PTR;
nom_attribut : stlgl;
find : boolean;
i : integer;

begin
ptr_r1 := de_pile(1);
ptr_r3 := de_pile(3);
PTR_TO_STRING(ptr_r1,nom_attribut);
TROUVER_VARIA(nom_attribut,i,find);
if find = false then
begin
MESSAGE_ERREUR(nom_attribut,5);
end
else
begin
if table[i].typ = 'str' then CREER_SF_NOEUD(SF_READB,ptr_s0)
 else CREER_SF_NOEUD(SF_INPUT,ptr_s0);
CREER_SF_NOEUD(SF_READA,ptr_s1);
CREER_SF_NOEUD(SF_ENTER,ptr_s2);
LIEN_DE_FRERE(ptr_s0,ptr_r3);
LIEN_DE_FRERE(ptr_r3,ptr_s1);
LIEN_DE_FRERE(ptr_s1,ptr_r1);
LIEN_DE_FRERE(ptr_r1,ptr_s2);
CREER_STANDARD(VS_READ,ptr_s0,ptr_l);
REDUIRE_PILE(5,VS_READ,ptr_l);
end;
end;

Procedure ROUT_SEM_30;

var ptr_r1, ptr_r6, ptr_r5, ptr_r8, ptr_r10, ptr_r4,
 ptr_s0, ptr_s1, ptr_s2, ptr_s3, ptr_s4, ptr_s5, ptr_s6, ptr_s7,
 ptr_s8, ptr_s9, ptr_s10, ptr_s11, ptr_s12, ptr_s13, ptr_s14,
 ptr_s15, ptr_s16, ptr_s17, ptr_s18, ptr_s19, ptr_s20, ptr_s21,
 ptr_l ,p , q , s , t , v: PTR;
tt,vv, ptr_r4b, ptr_s23,ptr_s30,ptr_s31,ptr_s5b,ptr_s22, qq,
ppp, pp, qq, ptr_s00,pppp, ptr_s32, ptr_s33 : PTR;

begin
ptr_r1 := de_pile(4);
MOT_IN_HEAP(tamp,pp);
MOT_IN_HEAP(tamp,ppp);
CREER_SF_NOEUD(SF_ASSIGN,ptr_s17);
CREER_SF_NOEUD(SF_ARROW,ptr_s18);
CREER_SF_NOEUD(SF_ENTER,ptr_s33);
LIEN_DE_FRERE(pp,ptr_s18);
LIEN_DE_FRERE(ptr_s18,ptr_r1);
LIEN_DE_FRERE(ptr_r1,ptr_s33);
CREER_STANDARD(VS_EXPR,pp,ptr_l);
REDUIRE_PILE(4,VS_EXPR,ptr_l);
end;

```

```
Procedure ROUT_SEM_31;
```

```
var ptr_r1, ptr_r4, ptr_s0, ptr_l : PTR;
```

```
begin
ptr_r1 := de_pile(1);
ptr_r4 := de_pile(4);
CREER_SF_NOEUD(SF_ARROW,ptr_s0);
LIEN_DE_FRERE(ptr_r1,ptr_s0);
LIEN_DE_FRERE(ptr_s0,ptr_r4);
CREER_STANDARD(VS_FACTOR,ptr_r1,ptr_l);
REDUIRE_PILE(4,VS_FACTOR,ptr_l);
end;
```

```
Procedure ROUT_SEM_32;
```

```
var ptr_r1,
 ptr_s0, ptr_s1,
 ptr_l : PTR;
```

```
begin
ptr_r1 := de_pile(1);
p:=ptr_r1;
verifier_arbre;
CREER_SF_NOEUD(SF_WRITE,ptr_s0);
CREER_SF_NOEUD(SF_ENTER,ptr_s1);
LIEN_DE_FRERE(ptr_s0,ptr_r1);
LIEN_DE_FRERE(ptr_r1,ptr_s1);
CREER_STANDARD(VS_WRITE,ptr_s0,ptr_l);
REDUIRE_PILE(3,VS_WRITE,ptr_l);
end;
```

```
Procedure ROUT_SEM_33;
```

```
var ptr_r1,
 ptr_s0 , ptr_s1, ptr_s2,
 ptr_l : PTR;
```

```
begin
ptr_r1 := de_pile(1);
p:=ptr_r1;
verifier_arbre;
CREER_SF_NOEUD(SF_WRITE,ptr_s0);
CREER_SF_NOEUD(SF_WRITE,ptr_s1);
CREER_SF_NOEUD(SF_ENTER,ptr_s2);
LIEN_DE_FRERE(ptr_s0,ptr_s1);
LIEN_DE_FRERE(ptr_s1,ptr_r1);
LIEN_DE_FRERE(ptr_r1,ptr_s2);
CREER_STANDARD(VS_WRITELN,ptr_s0,ptr_l);
REDUIRE_PILE(3,VS_WRITELN,ptr_l);
end;
```

```
Procedure ROUT_SEM_34;
```

```
var ptr_r0, ptr_r1, ptr_r2, ptr_s0, ptr_s1, ptr_l : PTR;
```

```
begin
ptr_r1 := de_pile(1);
CREER_SF_NOEUD(SF_PARENTHESE_OUVRANTE,ptr_s0);
CREER_SF_NOEUD(SF_PARENTHESE_FERMANTE,ptr_s1);
LIEN_DE_FRERE(ptr_s0,ptr_r1);
LIEN_DE_FRERE(ptr_r1,ptr_s1);
CREER_STANDARD(VS_FACTOR,ptr_s0,ptr_l);
REDUIRE_PILE(2,VS_FACTOR,ptr_l);
end;
```

```
Procedure ROUT_SEM_35;
```

```
var ptr_r1, ptr_r3, ptr_r4, ptr_r5,
 ptr_s0, ptr_s1, ptr_s2, ptr_s3, ptr_s4, ptr_s5, ptr_s6,
 ptr_l : PTR;
```

```
begin
ptr_r1 := de_pile(1);
ptr_r3 := de_pile(3);
CREER_SF_NOEUD(SF_WHILE,ptr_s1);
CREER_SF_NOEUD(SF_ENDDO,ptr_s2);
CREER_SF_NOEUD(SF_ENTER,ptr_s3);
CREER_SF_NOEUD(SF_ENTER,ptr_s4);
CREER_SF_NOEUD(SF_TABUL_P,ptr_s5);
CREER_SF_NOEUD(SF_TABUL_M,ptr_s6);
LIEN_DE_FRERE(ptr_s1,ptr_r3);
LIEN_DE_FRERE(ptr_r3,ptr_s5);
LIEN_DE_FRERE(ptr_s5,ptr_r1);
LIEN_DE_FRERE(ptr_r1,ptr_s6);
LIEN_DE_FRERE(ptr_s6,ptr_s3);
LIEN_DE_FRERE(ptr_s3,ptr_s2);
LIEN_DE_FRERE(ptr_s2,ptr_s4);
CREER_STANDARD(VS_WHILE,ptr_s1,ptr_l);
REDUIRE_PILE(4,VS_WHILE,ptr_l);
end;
```

```
Procedure ROUT_SEM_36;
```

```
var ptr_r1, ptr_r3, ptr_r5,
 ptr_s0, ptr_s1, ptr_s2, ptr_s3, ptr_s4, ptr_s5, ptr_s6, ptr_s7,
 ptr_s8, ptr_l : PTR;
```

```
begin
ptr_r1 := de_pile(1);
ptr_r3 := de_pile(3);
ptr_r5 := de_pile(5);
CREER_SF_NOEUD(SF_IF,ptr_s0);
CREER_SF_NOEUD(SF_ELSE,ptr_s1);
CREER_SF_NOEUD(SF_ENDIF,ptr_s2);
CREER_SF_NOEUD(SF_ENTER,ptr_s3);
CREER_SF_NOEUD(SF_ENTER,ptr_s4);
CREER_SF_NOEUD(SF_TABUL_P,ptr_s5);
CREER_SF_NOEUD(SF_TABUL_M,ptr_s6);
```

```

CREER_SF_NOEUD(SF_TABUL_P,ptr_s7);
CREER_SF_NOEUD(SF_TABUL_M,ptr_s8);
LIEN_DE_FRERE(ptr_s0,ptr_r5);
LIEN_DE_FRERE(ptr_r5,ptr_s7);
LIEN_DE_FRERE(ptr_s7,ptr_r3);
LIEN_DE_FRERE(ptr_r3,ptr_s8);
LIEN_DE_FRERE(ptr_s8,ptr_s1);
LIEN_DE_FRERE(ptr_s1,ptr_s3);
LIEN_DE_FRERE(ptr_s3,ptr_s5);
LIEN_DE_FRERE(ptr_s5,ptr_r1);
LIEN_DE_FRERE(ptr_r1,ptr_s6);
LIEN_DE_FRERE(ptr_s6,ptr_s2);
LIEN_DE_FRERE(ptr_s2,ptr_s4);
CREER_STANDARD(VS_IF,ptr_s0,ptr_l);
REDUIRE_PILE(6,VS_IF,ptr_l);
end;

```

Procedure ROUT\_SEM\_37;

```

var ptr_r1, ptr_r3, ptr_s0,
 ptr_s1, ptr_s2, ptr_s3, ptr_s4,
 ptr_l : PTR;

```

```

begin
ptr_r1 := de_pile(1);
ptr_r3 := de_pile(3);
CREER_SF_NOEUD(SF_IF,ptr_s0);
CREER_SF_NOEUD(SF_ENDIF,ptr_s1);
CREER_SF_NOEUD(SF_ENTER,ptr_s2);
CREER_SF_NOEUD(SF_TABUL_P,ptr_s3);
CREER_SF_NOEUD(SF_TABUL_M,ptr_s4);
LIEN_DE_FRERE(ptr_s0,ptr_r3);
LIEN_DE_FRERE(ptr_r3,ptr_s3);
LIEN_DE_FRERE(ptr_s3,ptr_r1);
LIEN_DE_FRERE(ptr_r1,ptr_s4);
LIEN_DE_FRERE(ptr_s4,ptr_s1);
LIEN_DE_FRERE(ptr_s1,ptr_s2);
CREER_STANDARD(VS_IF,ptr_s0,ptr_l);
REDUIRE_PILE(4,VS_IF,ptr_l);
end;

```

Procedure ROUT\_SEM\_38;

```

var ptr_r1, ptr_r3, ptr_s0,
 ptr_s1, ptr_s2, ptr_s3, ptr_s4, ptr_s5, ptr_s6, ptr_s7, ptr_s8,
 ptr_s9, ptr_s10, pp,ppp,p,pppp,ptr_l,ptr_s12, ptr_s11 : PTR;

```

```

begin
ptr_r1 := de_pile(1);
ptr_r3 := de_pile(3);
MOT_IN_HEAP(tamp2,p);
MOT_IN_HEAP(tamp2,pp);
MOT_IN_HEAP(tamp2,ppp);
MOT_IN_HEAP(tamp2,pppp);
CREER_SF_NOEUD(SF_ENTER,ptr_s0);

```

```

CREER_SF_NOEUD(SF_COUNT,ptr_s2);
CREER_SF_NOEUD(SF_N1,ptr_s3);
CREER_SF_NOEUD(SF_ENTER,ptr_s6);
CREER_SF_NOEUD(SF_IF,ptr_s7);
CREER_SF_NOEUD(SF_N,ptr_s8);
CREER_SF_NOEUD(SF_TABUL_P,ptr_s9);
CREER_SF_NOEUD(SF_TABUL_M,ptr_s10);
CREER_SF_NOEUD(SF_ENDIF,ptr_s11);
HEAD_SEQ(p,ptr_s0,ptr_s1);
LIEN_DE_FRERE(ptr_s0,ptr_s2);
LIEN_DE_FRERE(ptr_s2,ptr_r3);
LIEN_DE_FRERE(ptr_r3,ptr_s3);
IN_IF(pppp,ptr_s3,ptr_s5);
IN_END_SEC(pp,ppp,ptr_s5,ptr_s6);
LIEN_DE_FRERE(ptr_s6,ptr_s7);
LIEN_DE_FRERE(ptr_s7,ptr_s8);
LIEN_DE_FRERE(ptr_s8,ptr_s9);
LIEN_DE_FRERE(ptr_s9,ptr_r1);
LIEN_DE_FRERE(ptr_r1,ptr_s10);
LIEN_DE_FRERE(ptr_s10,ptr_s11);
CREER_STANDARD(VS_IF,ptr_s1,ptr_l1);
REDUIRE_PILE(4,VS_IF,ptr_l1);
end;

```

Procedure ROUT\_SEM\_41;

```

var ptr_r2, ptr_r4, ptr_r7, ptr_r9,
 ptr_s0, ptr_s3, ptr_s2, ptr_s4, ptr_s6,ptr_s1, ptr_s5,ptr_s7, ptr_s8,
 ptr_s9, ptr_s10, ptr_s11, ptr_s12, ptr_s13, ptr_s14, ptr_s15,ptr_s16,
 ptr_s18, ptr_l ,p , q, s, t, v: PTR;
 find, exist : boolean;
 i:integer;
 nom, mot :stlgl;

begin
ptr_r2 := de_pile(0);
ptr_r4 := de_pile(2);
CREER_SF_NOEUD(SF_SELECT,ptr_s0);
CREER_SF_NOEUD(SF_LOCATE1,ptr_s5);
CREER_SF_NOEUD(SF_ENTER,ptr_s4);
CREER_SF_NOEUD(SF_ENTER,ptr_s6);
PTR_TO_STRING(ptr_r4,nom);
TROUVER_DS_TABLE2(tamp1,i,find);
if find = false then
 begin
 nom:=tamp1;
 MOT_IN_HEAP(tamp1,ptr_r7);
 EXISTE_ENTITE(ptr_r7,tamp1,exist);
 if exist = false
 then MESSAGE_ERREUR(tamp1,4)
 else
 begin
 delete(nom,4,length(nom));
 DEPOSER_DS_TABLE(nom,tamp1);
 SET_TO_TRUE(tamp1);
 end
 end
 end

```

```

 end;
 end
 else SET_TO_TRUE(tamp1);
if con_ass = true then
 begin
 con_ass := false;
 TROUVER_VARIA(tamp,i,find);
 if (find = false) then MESSAGE_ERREUR(tamp,5);
 if (tamp=nom) then MESSAGE_ERREUR('',20);
 end;
LIEN_DE_FRERE(ptr_s6,ptr_s0);
LIEN_DE_FRERE(ptr_s0,ptr_r4);
LIEN_DE_FRERE(ptr_r4,ptr_s4);
LIEN_DE_FRERE(ptr_s4,ptr_s5);
LIEN_DE_FRERE(ptr_s5,ptr_r2);
CREER_STANDARD(VS_ASSIGNATION_ENTITE,ptr_s6,ptr_l);
REDUIRE_PILE(2,VS_ASSIGNATION_ENTITE,ptr_l);
end;

```

Procedure ROUT\_SEM\_42;

```

var ptr_r1, ptr_r6, ptr_r5, ptr_r9, ptr_r11, ptr_r4,
 ptr_s0, ptr_s3, ptr_s2, ptr_s4, ptr_s5, ptr_s6,ptr_s7, ptr_s8,
 ptr_s9, ptr_s10, ptr_s11, ptr_s12, ptr_s13, ptr_s14, ptr_s15,
 ptr_s16, ptr_s18, ptr_s19, ptr_s20, ptr_l ,w , q, s, t, v: PTR;
 nom_type : stlgl;

begin
ptr_r1 := de_pile(1);
ptr_r4 := de_pile(4);
ptr_r6 := de_pile(6);
ptr_r9 := de_pile(9);
ptr_r11 := de_pile(11);
ptr_r5 := de_pile(5);
ACCES_PAR_CLE(ptr_r6,ptr_r9,ptr_r4,ptr_r11,entitel,found);
if found = true then
 begin
 found := false;
 nom_type := typ1;
 p:=ptr_r4;
 verifier_arbre;
 if nom_type <> typ1 then MESSAGE_ERREUR('',7)
 else
 begin
 found := true;
 PTR_IN_HEAP(ptr_r11,w);
 PTR_IN_HEAP(ptr_r11,q);
 PTR_IN_HEAP(ptr_r11,s);
 CREER_SF_NOEUD(SF_SELECT,ptr_s0);
 CREER_SF_NOEUD(SF_END,ptr_s4);
 CREER_SF_NOEUD(SF_LOCATE1,ptr_s5);
 CREER_SF_NOEUD(SF_ENDIF,ptr_s10);
 CREER_SF_NOEUD(SF_ARROW,ptr_s18);
 CREER_SF_NOEUD(SF_ENTER,ptr_s6);
 CREER_SF_NOEUD(SF_ENTER,ptr_s16);
 end
 end
 end
end;

```

```

CREER_SF_NOEUD(SF_TABUL_P,ptr_s19);
CREER_SF_NOEUD(SF_TABUL_M,ptr_s20);
HEAD_SEQ(ptr_r11,ptr_s2,ptr_s0);
LIEN_DE_FRERE(ptr_s2,ptr_s5);
LIEN_DE_FRERE(ptr_s5,ptr_r6);
LIEN_DE_FRERE(ptr_r6,ptr_r5);
LIEN_DE_FRERE(ptr_r5,ptr_r4);
LIEN_DE_FRERE(ptr_r4,ptr_s6);
IN_IF(s,ptr_s6,ptr_s16);
LIEN_DE_FRERE(ptr_s16,ptr_s19);
LIEN_DE_FRERE(ptr_s19,ptr_r1);
IN_END(w,q,ptr_r1);
CREER_STANDARD(VS_FOR,ptr_s0,ptr_l);
REDUIRE_PILE(12,VS_FOR,ptr_l);
end;
end;
end;

```

Procedure ROUT\_SEM\_43;

```

var ptr_r1, ptr_r6, ptr_r5, ptr_r8, ptr_r10, ptr_r4,
 ptr_s0, ptr_s1, ptr_s2, ptr_s3, ptr_s4, ptr_s5, ptr_s6,ptr_s7,
 ptr_s8, ptr_s9, ptr_s10, ptr_s11, ptr_s12, ptr_s13, ptr_s14,
 ptr_s15, ptr_s16, ptr_s17, ptr_s18, ptr_s19, ptr_s20, ptr_s21,
 ptr_l ,p , q, s, t, v: PTR;
 tt,vv, ptr_s22, ptr_s23 : PTR;

begin
ptr_r1 := de_pile(1);
ptr_r4 := de_pile(4);
ptr_r6 := de_pile(6);
ptr_r8 := de_pile(8);
ptr_r10 := de_pile(10);
ACCES_PAR_ASSOCIATION(ptr_r6,ptr_r8,ptr_r4,ptr_r10,entitel,found,yes);
if found = true then
begin
PTR_IN_HEAP(ptr_r10,p);
PTR_IN_HEAP(ptr_r10,q);
PTR_IN_HEAP(ptr_r10,s);
MOT_IN_HEAP(entitel,t);
MOT_IN_HEAP(entitel,v);
CREER_SF_NOEUD(SF_SELECT,ptr_s0);
CREER_SF_NOEUD(SF_DO,ptr_s2);
CREER_SF_NOEUD(SF_LOCATE1,ptr_s5);
CREER_SF_NOEUD(SF_IF,ptr_s6);
CREER_SF_NOEUD(SF_ENDIF,ptr_s10);
CREER_SF_NOEUD(SF_RECNO,ptr_s16);
CREER_SF_NOEUD(SF_ASSIGN,ptr_s17);
CREER_SF_NOEUD(SF_ARROW,ptr_s18);
CREER_SF_NOEUD(SF_LOCATE2,ptr_s19);
CREER_SF_NOEUD(SF_NOT,ptr_s20);
CREER_SF_NOEUD(SF_EOF,ptr_s21);
CREER_SF_NOEUD(SF_ENTER,ptr_s11);
CREER_SF_NOEUD(SF_ENTER,ptr_s12);
CREER_SF_NOEUD(SF_ENTER,ptr_s13);

```

```

CREER_SF_NOEUD(SF_ENTER,ptr_s14);
CREER_SF_NOEUD(SF_ENTER,ptr_s15);
CREER_SF_NOEUD(SF_TABUL_P,ptr_s22);
CREER_SF_NOEUD(SF_TABUL_M,ptr_s23);
if yes = true then
begin
HEAD_SEQ(ptr_r10,ptr_s2,ptr_s0);
LIEN_DE_FRERE(ptr_s2,ptr_s5);
LIEN_DE_FRERE(ptr_s5,t);
LIEN_DE_FRERE(t,ptr_s17);
LIEN_DE_FRERE(ptr_s17,ptr_r4);
LIEN_DE_FRERE(ptr_r4,ptr_s18);
LIEN_DE_FRERE(ptr_s18,v);
LIEN_DE_FRERE(v,ptr_s11);
IN_IF(s,ptr_s11,ptr_s16);
LIEN_DE_FRERE(ptr_s16,ptr_s22);
LIEN_DE_FRERE(ptr_s22,ptr_r1);
IN_END(p,q,ptr_r1);
end
else
begin
MOT_IN_HEAP(entitel,tt);
MOT_IN_HEAP(entitel,vv);
LIEN_DE_FRERE(ptr_s0,ptr_r10);
LIEN_DE_FRERE(ptr_r10,ptr_s12);
LIEN_DE_FRERE(ptr_s12,ptr_s19);
LIEN_DE_FRERE(ptr_s19,tt);
LIEN_DE_FRERE(tt,ptr_s17);
LIEN_DE_FRERE(ptr_s17,ptr_r4);
LIEN_DE_FRERE(ptr_r4,ptr_s18);
LIEN_DE_FRERE(ptr_s18,vv);
LIEN_DE_FRERE(vv,ptr_s13);
LIEN_DE_FRERE(ptr_s13,ptr_s6);
LIEN_DE_FRERE(ptr_s6,ptr_s20);
LIEN_DE_FRERE(ptr_s20,ptr_s21);
LIEN_DE_FRERE(ptr_s21,ptr_s14);
LIEN_DE_FRERE(ptr_s14,ptr_s22);
LIEN_DE_FRERE(ptr_s22,ptr_r1);
LIEN_DE_FRERE(ptr_r1,ptr_s23);
LIEN_DE_FRERE(ptr_s23,ptr_s10);
LIEN_DE_FRERE(ptr_s10,ptr_s15);
end;
CREER_STANDARD(VS_FOR,ptr_s0,ptr_l);
REDUIRE_PILE(11,VS_FOR,ptr_l);
end
else MESSAGE_ERREUR('',20);
end;

```

Procedure ROUT\_SEM\_44;

```

var ptr_r1, ptr_r5, ptr_r3,
 ptr_s0, ptr_s3, ptr_s4, ptr_s5, ptr_s6, ptr_s7,
 ptr_s8, ptr_s9,
 ptr_l : PTR;
 noml : str;

```

```
intab : boolean;
```

```
begin
intab := false;
ptr_r1 := de_pile(1);
ptr_r5 := de_pile(5);
ptr_r3 := de_pile(3);
PTR_IN_HEAP(ptr_r5,q);
ACCES_SEQUENTIEL(ptr_r5,ptr_r3,found);
if found = true then
begin
CREER_SF_NOEUD(SF_SKIP,ptr_s3);
CREER_SF_NOEUD(SF_END,ptr_s4);
CREER_SF_NOEUD(SF_SELECT,ptr_s5);
CREER_SF_NOEUD(SF_TABUL_P,ptr_s6);
CREER_SF_NOEUD(SF_TABUL_M,ptr_s7);
CREER_SF_NOEUD(SF_ENTER,ptr_s9);
HEAD_SEQ(ptr_r5,ptr_s8,ptr_s0);
LIEN_DE_FRERE(ptr_s8,ptr_s6);
LIEN_DE_FRERE(ptr_s6,ptr_r1);
LIEN_DE_FRERE(ptr_r1,ptr_s7);
LIEN_DE_FRERE(ptr_s7,ptr_s5);
LIEN_DE_FRERE(ptr_s5,q);
LIEN_DE_FRERE(q,ptr_s9);
LIEN_DE_FRERE(ptr_s9,ptr_s3);
LIEN_DE_FRERE(ptr_s3,ptr_s4);
CREER_STANDARD(VS_FOR,ptr_s0,ptr_l);
REDUIRE_PILE(6,VS_FOR,ptr_l);
end
else MESSAGE_ERREUR('',20);
end;
```

```
Procedure ESSAYER_DE_REDUIRE;
```

```
var NR : integer;
 trouve_regle , arret : boolean;
```

```
begin
TROUVER_REGLE_A_REDUIRE(trouve_regle,NR);
if trouve_regle = true then
begin
case GRAMMAIRE[NR].NUMERO_PROD OF
1, 2, 3, 4,
5, 6 : ROUT_SEM_6;
7 : ROUT_SEM_7;
8 : ROUT_SEM_8;
9 : ROUT_SEM_9;
10 : ROUT_SEM_10;
11 : ROUT_SEM_11;
12 : ROUT_SEM_12;
13 : ROUT_SEM_13;
14 : ROUT_SEM_14;
15 : ROUT_SEM_15;
16 : ROUT_SEM_16;
```

```
17 : ROUT_SEM_17;
18 : ROUT_SEM_18;
19 : ROUT_SEM_19;
20 : ROUT_SEM_20;
21 : ROUT_SEM_21;
22 : ROUT_SEM_22;
23 : ROUT_SEM_23;
24 : ROUT_SEM_24;
25 : ROUT_SEM_25;
26 : ROUT_SEM_25;
27 : ROUT_SEM_27;
28 : ROUT_SEM_28;
29 : ROUT_SEM_29;
30 : ROUT_SEM_30;
31 : ROUT_SEM_31;
32 : ROUT_SEM_32;
33 : ROUT_SEM_33;
34 : ROUT_SEM_34;
35 : ROUT_SEM_35;
36 : ROUT_SEM_36;
37 : ROUT_SEM_37;
38 : ROUT_SEM_38;
39 : ROUT_SEM_6;
40 : ROUT_SEM_25;
41 : ROUT_SEM_41;
42 : ROUT_SEM_42;
43 : ROUT_SEM_43;
44 : ROUT_SEM_44;
45 : ROUT_SEM_6;

 end;
end
else MESSAGE_ERREUR(' ',20);
end;
```



```

write(lister,tab,TANK.ELEM[i]);
enter:=false;
end
else write(lister,TANK.ELEM[i])

end;

end;
end;
end;
walk(p^.frere);
end;
end;

begin
tab := '';
walk(p);
end;

begin
j:=1;
assign(LISTER,RESULTAT);
rewrite(LISTER);
writeln(lister,'close databases');
if (table[i].bit = true) and (table[i].typ <> 'str') and
(table[i].typ <> 'ent') then
begin
writeln(lister,'select ',j);
writeln(lister,'use ',table[i].typ,' alias ',table[i].name);
j := j + 1;
end;
WALK_THROUGH;
writeln(lister);
writeln(lister,'close databases');
close(LISTER);
writeln;
writeln('La compilation a réussie. ');
writeln('Le texte dBase se trouve dans ',RESULTAT);
repeat until keyressed;
end;

```

```

Procedure INIT(var continue_compilation : boolean);

{ Cette procédure vérifie si les fichiers GRAMFILE.DTA et
 FSFILE.DTA sont des fichiers existants. Si ce n'est pas
 le cas "continue_compilation" est mis à faux.
 Elle initialise aussi la table de travail.
}

var i : integer;

begin
 for i:=1 to max_table do
 begin
 table[i].name:='';
 table[i].typ:='';
 table[i].origine:='';
 table[i].bit:=false;
 end;
 con_ass:=false;
 enter := false;
 continue_compilation := true;
 typ1:='';
 assign(GRAMFILE,'GRAMFILE.DTA');
 {$i-}reset(GRAMFILE){$i+};
 if ioresult <> 0 then
 begin
 continue_compilation := false;
 writeln('Le fichier GRAMFILE.DTA qui contient la grammaire');
 writeln(' est introuvable !');
 end
 else
 begin
 assign(FSFILE,'FSFILE.DTA');
 {$i-}reset(FSFILE){$i+};
 if ioresult <> 0 then
 begin
 continue_compilation := false;
 writeln('Le fichier FSFILE.DTA qui contient les rubans');
 writeln(' de traduction est introuvable !');
 end
 end;
 end;
end;

```

Procedure compil;

```

{*****}
{* Ce module effectue la génération dBase *}
{*****}

begin
INIT(continue_compilation);
if continue_compilation <> false then
begin
 LEXICA;
 if continue_compilation <> false then
 begin
 CONSTRUC;
 VAR_DISTINGUEE := false;
 DETECTER_ERREUR := false;
 PILE.TOP := 0;
 ENTREE.TOP := 0;
 NUM_TABLE := 0;
 PILE.ELEM[0].SYMBOL := S_ENTREE_SYM;
 GARNIR_TOPSYM;
 while (VAR_DISTINGUEE = false) and (DETECTER_ERREUR = false) do
 begin
 T:=SRT[PILE.ELEM[PILE.TOP].SYMBOL,TOPSYM.KIND];
 Case T of
 ERRORACT : begin
 MESSAGE_ERREUR('',20);
 end;
 SHIFTACT : begin
 SHIFT_TOPSYM;
 GARNIR_TOPSYM;
 end;
 REDUCEACT : begin
 ESSAYER_DE_REDUIRE;
 if PILE.ELEM[PILE.TOP].SYMBOL = VS_SEQUENCE then
 VAR_DISTINGUEE := true;
 end;
 end;
 end;
 end;
 if DETECTER_ERREUR = true
 then MESSAGE_ERREUR('',10)
 else if VAR_DISTINGUEE = true then TRADUCTION_DS_FICHER;
 end
 else MESSAGE_ERREUR('',8);
 end;
end;
end;

```

```
(*****
(* Ce module permet l'introduction d'une base de données en projet. *)
(*****)
```

```
Procedure CHARGEUR;
```

```
type
 chaine=string[200];

var attsuiv,continuer,cont,fin,asssuiv:boolean;
 sch:Tschema;
 ent:Tentite;
 ass:Tassociation;
 att:Tattribut;
 rol:Trole;
 pro:Tprocedur;
 entsuiv:boolean;
 base:string[20];
```

```
Function Majuscule(ch:chaine):chaine;
```

```
var i:integer;

begin
 for i:=1 to length(ch) do
 ch[i]:=upcase(ch[i]);
 majuscule:=ch;
 end;
```

```
Procedure ECRAN_BAS;
```

```
begin
 write('_____');
 writeln('_____');
 write('Base de données : Type d''Entité :');
 writeln('_____');
 write('Type d''Assoc. : Attribut :');
 writeln('_____');
 write('Messages :');
 writeln('_____');
 write('_____');
 writeln('_____');
end;
```

```
Procedure ECRAN_HAUT;
```

```
begin
 write('_____');
 writeln('_____');
 write('_____ Gestion des Bases de données');
 writeln('_____');
 write('_____');
 writeln('_____');
 writeln;
end;
```

```
Procédure SaisieAttribut;
```

```
var y : integer;
```

```
begin
 repeat
 cont:=true;
 clrscr;
 ECRAN_HAUT;
 write('_____');
 writeln('_____');
 write('|_____');
 writeln('|_____');
 write('|_____');
 writeln('|_____');
 gotoxy(1,20);
 ECRAN_BAS;
 gotoxy(23,21);
 write(sch.nom_schema);
 gotoxy(48,21);
 write('Type d''Entité : ');
 write(ent.nom_entite);
 gotoxy(1,10);
 writeln(' NOM IDENTIFIANT ');
 writeln(' _____ _____ ');
 writeln;
 writeln(' Description');
 write('_____');
 write('_____');
 write('_____');
 writeln('_____');
 writeln('Type Pascal : Int,Real,St,Ch,Bool');
 writeln(' _____ longueur : _____');
 y:=11;
 gotoxy(3,y);
 readln(att.nom_attribut);
 att.nom_attribut:=majuscule(att.nom_attribut);
 if att.nom_attribut='' then
 begin
 continuer:=false;
 cont:=false;
 attsuiv:=true;
 fin:=false;
 end
 else
 begin
 repeat
 gotoxy(37,y);
 readln(att.id_attribut);
 att.id_attribut := upcase(att.id_attribut)
 until att.id_attribut in ['O','N'];
 gotoxy(3,y+3);
 readln(att.description_attribut);
 att.description_attribut:=majuscule(att.description_attribut);
```

```

 gotoxy(1,y+7);
 readln(att.type_attribut);
 if upcase(att.type_attribut)='S' then
 begin
 gotoxy(43,y+7);
 readln(att.longueur_attribut);
 end;
 dbcreate(attribut,att);
 dbinsert(att,ent,caracteristique);
 if dbstatus = 0 then
 begin
 gotoxy(14,23);
 writeln('L''attribut ',att.nom_attribut,' est inséré!');
 delay(1000);
 end
 else
 begin
 gotoxy(14,30);
 writeln('L''attribut ',att.nom_attribut,'n'' est pas inséré!');
 delay(1000);
 end;
 end;
until cont=false;
end;

Procedure Saisie_Tentite;

var ch:char;
 y : integer;

begin
 entsuiv:=true;
 attsuiv:=true;
 if positionne=false then
 begin
 gotoxy(30,11);
 writeln('Base de données non positionnée!');
 end
 else
 repeat
 clrscr;
 ECRAN_HAUT;
 write('_____');
 writeln('');
 write('|_____');
 writeln('Création d''un Type d''Entité');
 write('|');
 writeln('');
 write('_____');
 writeln('');
 gotoxy(1,20);
 ECRAN_BAS;
 gotoxy(23,21);
 write(sch.nom_schema);
 gotoxy(1,10);
 writeln(' NOM');
 writeln(' _____');
 until cont=false;
 end;
end;

```

```

writeln;
writeln(' Description');
write('_____');
write('_____');
write('_____');
writeln('_____');
write(' F1 : EXIT');
y:=11;
gotoxy(3,y);
readln(ent.nom_entite);
if ent.nom_entite = '' then entsuiv:=false
else
 begin
 ent.nom_entite:=majuscule(ent.nom_entite);
 gotoxy(3,14);
 readln(ent.description_entite);
 writeln(ent.nom_entite);
 dbcreate(entite,ent);
 dbinsert(ent,sch,collectionne_e);
 if dbstatus = 0 then
 begin
 gotoxy(14,23);
 writeln('L''entité ',ent.nom_entite,' est insérée!');
 delay(1000);
 end
 else
 begin
 gotoxy(14,30);
 writeln('L''entité ',ent.nom_entite,'n'' est pas insérée!');
 delay(1000);
 end;
 repeat
 Saisieattribut;
 until attsuiv=true;
 end;
 until entsuiv=false;
end;

Procedure EcranGestion(var chx:char);

var y:integer;
 ch:string[10];

begin
 clrscr;
 ECRAN_HAUT;
 write('_____');
 writeln('_____');
 write('|_____');
 writeln('|_____');
 write('|_____');
 writeln('|_____');
 gotoxy(6,8);
 write('=====');
 writeln('=====');

```

```

write(' || Commandes à exécuter (2 Initiales) ');
writeln(' || ');
write(' || ');
writeln(' || ');
write(' || Créer Type d''Entité ');
writeln(' || ');
write(' || Type d''Association ');
writeln(' || ');
write(' || Base de données ');
writeln(' || ');
write(' || Procedure ');
writeln(' || ');
write(' || F1 = Quitter gestion ');
writeln(' || ');
write(' || ');
writeln('=====');
write('=====');
writeln('-----');
write(' Commande : ');
writeln('-----');
write('-----');
writeln('-----');
gotoxy(1,20);
ECRAN_BAS;
repeat
 gotoxy(18,18);
 ch:='';
 y:=18;
 chx:=readkey;
 chx:=upcase(chx);
 if (chx <> 'P') and (chx <> 'C') and (chx <> #59) then
 begin
 sound(440);
 delay(500);
 gotoxy(16,23);
 writeln('Caractère non autorisé!');
 delay(500);
 gotoxy(16,23);
 clreol;
 nosound;
 gotoxy(78,23);
 writeln(' ');
 end;
 if chx in ['C','P'] then write(chx);
until chx in ['C','P',#59];
if (chx <> #59) then
begin
ch:=chx;
y:=y+1;
repeat
 gotoxy(y,18);
 chx:=readkey;
 chx:=upcase(chx);
 if (chx <> 'E') and (chx <> 'B') and (chx <> 'A')
 and (chx <> 'P') and (chx <> #59) then

```

```

begin
 sound(440);
 delay(500);
 gotoxy(16,23);
 writeln('Caractère non autorisé!');
 delay(500);
 gotoxy(16,23);
 clreol;
 nosound;
 gotoxy(78,23);
 writeln('|');
end;
if chx in ['E','B','A','P'] then write(chx);
ch:=ch+chx;
until chx in ['E','B','A','P',#59];
end;
y:=18;
if ch='PB' then chx:='P';
if ch='PE' then chx:='Q';
if ch='CB' then chx:='B';
if ch='CE' then chx:='E';
if ch='CA' then chx:='A';
if ch='CP' then chx:='C';
end; { procedure ecran }

```

Procedure origine\_cible;

var trouve : boolean;

```

begin
 clrscr;
 ECRAN_HAUT;
 write('_____');
 writeln('_____');
 write('_____');
 writeln('_____');
 write('_____');
 writeln('_____');
 writeln(' Origine');
 writeln('_____');
 writeln;
 writeln(' Cible');
 writeln('_____');
 writeln;
 gotoxy(1,20);
 ECRAN_BAS;
 gotoxy(23,21);
 write(sch.nom_schema);
 gotoxy(23,22);
 write(ass.nom_association);
 gotoxy(3,9);
 readln(rol.origine_role);
 rol.origine_role := majuscule(rol.origine_role);
 gotoxy(3,12);
 readln(rol.cible_role);

```

```

rol.cible_role := majuscule(rol.cible_role);
dbcreate(rol,rol);
dbinsert(rol,ass,lien);
if dbstatus <> 0 then writeln('Not ok');
dbfirst(entite,ent);
trouve:=false;
while dbfound and trouve do
 begin
 if ENT.NOM_ENTITE = rol.cible_role then
 begin
 trouve:=true;
 dbinsert(rol,ent,joue);
 if dbstatus <> 0 then writeln('Not OK');
 end
 else dbnext(entite,ent);
 end;
trouve:=false;
while dbfound and trouve do
 begin
 if ENT.NOM_ENTITE = rol.origine_role then
 begin
 trouve:=true;
 dbinsert(rol,ent,joue);
 if dbstatus <> 0 then writeln('Not OK');
 end
 else dbnext(entite,ent);
 end;
end;

```

Procedure Saisie\_Tassociation;

```

var continuer : boolean;
 y : integer;

```

```

begin
continuer:=true;
repeat
clrscr;
ECRAN_HAUT;
write('_____');
writeln('_____');
write('|_____Création d''un Type d''Association');
writeln('|_____');
write('_____');
writeln('_____');
gotoxy(1,20);
write('_____');
writeln('_____');
write('| Base de données : ');
write(sch.nom_schema);
gotoxy(48,21);
writeln('Type d''Entité : |');
write('| Type d''Assoc. : |');
writeln('|_____');
write('| Messages : |');

```

```

writeln(' |');
write('_____');
writeln('_____');
gotoxy(1,10);
writeln(' NOM');
writeln('_____');
writeln;
writeln(' Description');
writeln('_____');
writeln;
y:=11;
gotoxy(3,y);
readln(ass.nom_association);
ass.nom_association := majuscule(ass.nom_association);
if ass.nom_association='' then continuer:=false
else
begin
gotoxy(3,y+3);
readln(ass.description_associat);
dbcreate(association,ass);
dbinsert(ass,sch,collectionne_a);
if dbstatus = 0 then
begin
gotoxy(14,23);
writeln('L''association ',ass.nom_association,' est insérée!');
delay(1000);
end
else
begin
gotoxy(14,30);
writeln('L''association ',ass.nom_association,'n'' est pas insérée!');
delay(1000);
end;
writeln;
origine_cible;
end;
until continuer=false;
end;

Procédure saisie_base;

var continuer,fin:boolean;
 getal,y:integer;

begin
clrscr;
ECRAN_HAUT;
write('_____');
writeln('_____');
write('|_____Création d''une Base de données');
writeln(' |');
write('_____');
writeln('_____');
writeln(' NOM');
write('_____');

```

```

writeln;
gotoxy(1,19);
ECRAN_BAS;
gotoxy(3,9);
readln(sch.nom_schema);
sch.nom_schema:=majuscule(sch.nom_schema);;
dbcreate(schema,sch);
case dbstatus of
 0 : begin
 gotoxy(16,22);
 writeln('La Base de données ',sch.nom_schema,' est crée!');
 positionne:=true;
 delay(1000);
 end;
 2 : begin
 gotoxy(11,30);
 writeln('Base de données existe déjà!');
 delay(1000);
 end;
end;
end;

```

Procedure saisie\_proc;

```

var continuer,fin:boolean;
 getal,y:integer;
 trouve : boolean;

```

```

begin
 clrscr;
 ECRAN_HAUT;
 write(' _____');
 writeln(' _____');
 write(' | Intoduction d''une procédure ');
 writeln(' | ');
 write(' _____');
 writeln(' _____');
 writeln(' NOM');
 writeln(' _____');
 writeln;
 writeln(' NOM DE LA BASE');
 write(' _____');
 writeln;
 gotoxy(1,19);
 ECRAN_BAS;
 gotoxy(3,9);
 readln(pro.nom_procedure);
 gotoxy(3,12);
 readln(base);
 base := CAPITAL(base);
 trouve := false;
 dbfirst(schema,sch);
 while dbfound and not trouve do
 begin
 if sch.nom_schema = base then

```

```

begin
pro.nom_procedure:=majuscule(pro.nom_procedure);;
dbcreate(procedur,pro);
dbinsert(pro,sch,appartenance);
writeln(dbstatus);
case dbstatus of
0 : begin
 gotoxy(16,22);
 writeln('La procédure ',pro.nom_procedure,' est insérée!');
 positionne:=true;
 delay(1000);
 end
 else
 begin
 gotoxy(11,30);
 writeln('La procédure ',pro.nom_procedure,' n''est pas insérée!');
 delay(1000);
 end;
end;
end;
dbnext(schema,sch);
end;
end;

begin
repeat
 EcranGestion(chx);
 case chx of
 'E': saisie_Tentite;
 'A': saisie_Tassociation;
 'B': saisie_base;
 'C': saisie_proc;
 end;
until chx=#59;
end;

```

```
(*****)
(* Ce module permet de produire un rapport détaillé de la base des *)
(* spécifications. *)
(*****)
```

```
Procedure rapport;
```

```
type y = record
 entite : array[1..15] of record
 nom : string[20];
 description : string[194];
 attribut : array[1..10] of record
 nom : string[20];
 description : string[194];
 type_at : char;
 longueur : integer;
 end;
 nbre_at : integer;
 end;
 nbre_ent : integer;
end;
z = array[1..10] of integer;
q = record
 association : array[1..15] of record
 nom : string[20];
 description : string[194];
 origine : string[20];
 cible : string[20];
 end;
 nbre_ass : integer;
end;
```

```
var ent : Tentite;
 ass : Tassociation;
 att : Tattribut;
 sch : Tschema;
 rol : Trole;
 pro : Tprocedur;
 nbre_ent, nbre_ass, choix : integer;
 nbre_at, i, j, n : integer;
 x : y;
 p : q;
 base, entit, associat, typ, nom:string[20];
 nbr_at : z;
 tp : string[20];
 trouver_base : boolean;
```

```
Procedure sortie;
```

```
var i, j : integer;
```

```
begin
 clrscr;
 write('Vers quel organe désirez-vous orienter le rapport');
 writeln(' de la base de données?');
```

```

write(' 1');
write(' : Imprimante');
writeln;
write(' 2');
write(' : Ecran');
writeln;
repeat
 writeln('Quel est votre choix (1 ou 2) ?');
 readln(choix);
until choix in [1..2];
clrscr;
case choix of
 2 : begin
 writeln(' RAPPORT SUR UNE BASE DE DONNEES ');
 writeln('*****');
 writeln;
 writeln('NOM : ',sch.nom_schema);
 writeln;
 writeln('Liste des types d''Entités');
 writeln('=====');
 writeln;
 for i:=1 to nbre_ent-1 do
 begin
 entit:=x.entite[i].nom;
 writeln(i, ' : ',entit);
 writeln(' Description: ');
 writeln(' Attributs:');
 for j:=1 to x.entite[i].nbre_at-1 do
 begin
 write(' ');
 write(x.entite[i].attribut[j].nom);
 tp:=x.entite[i].attribut[j].type_at;
 if upcase(tp[1])='S' then
 begin
 tp:='S' + 'tring[';
 write(' : ',tp);
 writeln(x.entite[i].attribut[j].longueur,']');
 end
 else
 begin
 if upcase(tp[1])='I' then tp:='I' + 'nteger';
 writeln(' : ',tp);
 end;
 end;
 end;
 writeln;
 writeln(' Description Attribut:');
 for j:=1 to x.entite[i].nbre_at-1 do
 begin
 write(' ');
 write(x.entite[i].attribut[j].nom,' : ');
 writeln(x.entite[i].attribut[j].description);
 end;
 repeat until keypressed;
 end;
 writeln;
 end;
end;

```

```

writeln('Liste des Types d''Association');
writeln('=====');
writeln;
for i:=1 to nbre_ass-1 do
begin
 associat:=p.association[i].nom;
 writeln(i,' : ',associat);
 writeln(' Description: ');
 write(' Origine : ');
 writeln(p.association[i].origine);
 write(' Cible : ');
 writeln(p.association[i].cible);
 writeln;
 repeat until keypressed;
end;
end;
1 : begin
 writeln(lst,' RAPPORT SUR UNE BASE DE DONNEES ');
 writeln(lst,'*****');
 writeln(lst);
 writeln(lst,'NOM : ',sch.nom_schema);
 writeln(lst);
 writeln(lst,'Liste des types d''Entités');
 writeln(lst,'=====');
 writeln(lst);
 for i:=1 to nbre_ent-1 do
begin
 entit:=x.entite[i].nom;
 writeln(lst,i,' : ',entit);
 writeln(lst,' Description: ');
 writeln(lst,' Attributs:');
 for j:=1 to x.entite[i].nbre_at-1 do
begin
 write(lst,' ');
 write(lst,x.entite[i].attribut[j].nom);
 tp:=x.entite[i].attribut[j].type_at;
 if upcase(tp[1])='S' then
begin
 tp:='S' + 'tring[';
 write(lst,' : ',tp);
 writeln(lst,x.entite[i].attribut[j].longueur,']');
end
else
begin
 if upcase(tp[1])='I' then tp:='I' + 'nteger';
 writeln(lst,' : ',tp);
end;
end;
writeln(lst);
writeln(lst,' Description Attribut:');
 for j:=1 to x.entite[i].nbre_at-1 do
begin
 write(lst,' ');
 write(lst,x.entite[i].attribut[j].nom,' : ');
 writeln(lst,x.entite[i].attribut[j].description);

```

```

end;

end;
writeln(lst);
writeln(lst,'Liste des Types d''Association');
writeln(lst,'=====');
writeln(lst);
for i:=1 to nbre_ass-1 do
begin
 associat:=p.association[i].nom;
 writeln(lst,i,' : ',associat);
 writeln(lst,' Description: ');
 write(lst,' Origine : ');
 writeln(lst,p.association[i].origine);
 write(lst,' Cible : ');
 writeln(lst,p.association[i].cible);
 writeln(lst);
end;

end;

end;
end;

begin
clrscr;
trouver_base := false;
dbopen('METABASE');
gotoxy(15,14);
write('Nom de la base de données : ');
gotoxy(43,14);
readln(base);
base:=capital(base);
dbfirst(schema,sch);
while dbfound and not trouver_base do
begin
 if sch.nom_schema = base then trouver_base := true
 else dbnext(schema,sch);
end;
if trouver_base = false then writeln('Base n''existe pas !')
else
begin
dbfpath(ent,sch,collectionne_e);
 nbre_ent:=1;
 while dbfound do
begin
 nbre_at:=1;
 x.entite[nbre_ent].nom:=ent.nom_entite;
 dbfpath(att,ent,caracteristique);
 while dbfound do
begin
 x.entite[nbre_ent].attribut[nbre_at].nom:=att.nom_attribut;
 x.entite[nbre_ent].attribut[nbre_at].description:=
att.description_attribut;
 x.entite[nbre_ent].attribut[nbre_at].type_at:=att.type_attribut;
 x.entite[nbre_ent].attribut[nbre_at].longueur:=
att.longueur_attribut;

```

```

 dbnpath(att,ent,caracteristique);
 nbre_at:=nbre_at+1;
 end;
 nbr_at[nbre_at]:=nbre_at;
 x.entite[nbre_ent].nbre_at:=nbre_at;
 nbre_ent:=nbre_ent+1;
 dbnpath(ent,sch,collectionne_e);
 end;
dbfpath(ass,sch,collectionne_a);
nbre_ass:=1;
while dbfound do
 begin
 p.association[nbre_ass].nom:=ass.nom_association;
 dbfpath(rol,ass,lien);
 while dbfound do
 begin
 p.association[nbre_ass].origine:=rol.origine_role;
 p.association[nbre_ass].cible:=rol.cible_role;
 dbnpath(rol,ass,lien);
 nbre_ass:=nbre_ass+1;
 end;
 dbnpath(ass,sch,collectionne_a);
 end;
 sortie;
end;
dbclose;
end;

```

```
(*****)
(* Cette procédure affiche à l'écran le menu principal de l'atelier. *)
(*****)
```

```
Procedure EcranInit(var k : char);
```

```
begin
 clrscr;
 write(' ');
 writeln(' ');
 write(' ');
 writeln(' ');
 write(' ');
 writeln(' ');
 gotoxy(17,12);
 writeln(' ');
 writeln(' ');
 writeln(' ');
 writeln(' ');
 writeln(' ');
 writeln(' ');
 repeat
 gotoxy(60,24);
 writeln('Votre choix : ');
 gotoxy(74,24);
 readln(k);
 k:=upcase(k);
 until k in ['1','2','3','4','5'];
end;
```

```
(*****)
(* Ce module contrôle la présentation des informations sur écran ainsi *)
(* que le dialogue avec l'utilisateur. Il coordonne l'activation du *)
(* module CHARGEUR, du module RAPPORT et du module COMPIL. *)
(*****)
```

```
{ $R- } {Range checking off}
{ $B+ } {Boolean complete evaluation on}
{ $S+ } {Stack checking on}
{ $I+ } {I/O checking on}
{ $N- } {No numeric coprocessor}
{ $M 65500,16384,655360 } {Turbo 3 default stack and heap}
```

Program atelier;

Uses

```
 Crt,
 Printer,
 turbo3;
```

```
(* $I METABASE.TYP *)
(* $I DECL_COM.TYP *)
(* $I DECL_COM.VAR *)
(* $I DBMS.PAS *)
(* $I INIT_COM.PAS *)
(* $I LEXICA.PAS *)
(* $I UTILIT.PAS *)
(* $I CONSTRUC.PAS *)
(* $I TRADUCT.PAS *)
(* $I PARCOUR.PAS *)
(* $I COMPIL.PAS *)
(* $I DECL_DOC.VAR *)
(* $I CHARGEUR.PAS *)
(* $I RAPPORT.PAS *)
(* $I ECRAN_IN.PAS *)
```

```
begin
dbopen('METABASE');
textbackground(9);
positionne:=false;
repeat
EcranInit(chx);
case chx of
 '1' : CHARGEUR;
 '3' : RAPPORT;
 '4' : COMPIL;
 '5' : clrscr;
end;
until chx = '5';
dbclose;
end.
```

## **ANNEXE 1**

\*\*\*\*\*

## NOM : GESTION

## Liste des types d'Entités

=====

## 1 : CLIENT

Description:

Attributs:

NCLI : String[4]

NOM : String[20]

ADRESSE : String[30]

LOCALITE : String[40]

Description Attribut:

NCLI : NCLI représente le numéro identifiant le client

NOM : NOM représente le nom du client

ADRESSE : ADRESSE représente l'adresse du client

LOCALITE : LOCALITE représente la localité de résidence du client

## 2 : PRODUIT

Description:

Attributs:

NPRO : String[5]

LIBELLE : String[40]

PRIX : Integer

QSTOCK : Integer

Description Attribut:

NPRO : NPRO représente le numéro identifiant un produit

LIBELLE : LIBELLE donne le nom conventionnel du produit

PRIX : PRIX indique le prix à l'unité du produit

QSTOCK : QSTOCK indique la quantité restant en stock

## 3 : COMMANDE

Description:

Attributs:

NCOM : String[4]

DATE : String[8]

Description Attribut:

NCOM : NCOM représente le numéro identifiant une commande

DATE : DATE représente la date à laquelle la commande a été passée

## 4 : LIGNECOM

Description:

Attributs:

QCOM : Integer

Description Attribut:

QCOM : QCOM représente la quantité commandée

## Liste des Types d'Association

=====

## 1 : CC

Description:

Origine : CLIENT

Cible : COMMANDE

## 2 : CL

Description:

Origine : COMMANDE

Cible : LIGNECOM

87

3 : PL

Description:

Origine : PRODUIT

Cible : LIGNECOM

## **ANNEXE 2**

## Les règles de la grammaire

```

1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 17
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 9
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 8
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 11
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 12
3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 7 1 19
3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 19
7 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 7 1 20
7 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 20
2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 21 23 4
10 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 4 47 4
47 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 35
47 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 41
47 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 42
47 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 43
47 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 44
47 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 45
4 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 4 24 5
4 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 4 25 5
4 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 5
5 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 5 26 6
5 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 5 27 6
5 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 6
6 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 21
6 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 22
14 0 0 0 0 0 0 0 0 0 0 0 0 4 28 36 4 47 4 29
14 0 0 0 0 0 0 0 0 0 0 0 0 0 4 28 4 36 4 29
8 0 0 0 0 0 0 0 0 0 0 0 0 0 30 28 4 34 4 29
4 0 0 0 0 0 0 0 0 0 0 0 0 0 4 28 36 14 29
6 0 0 0 0 0 0 0 0 0 0 0 0 0 4 28 36 4 29
11 0 0 0 0 0 0 0 0 0 0 0 0 0 0 18 28 4 29
12 0 0 0 0 0 0 0 0 0 0 0 0 0 0 46 28 4 29
6 0 0 0 0 0 0 0 0 0 0 0 0 0 0 28 4 29
16 0 0 0 0 0 0 0 0 0 0 0 0 0 48 10 38 7 39
17 0 0 0 0 0 0 0 0 0 0 0 0 37 10 38 7 40 7 39
17 0 0 0 0 0 0 0 0 0 0 0 0 0 37 10 38 7 39
17 0 0 0 0 0 0 0 0 0 0 0 0 0 37 14 38 7 39
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 16
6 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 50
13 0 0 0 0 0 0 0 0 0 0 0 0 0 0 21 23 14
9 0 0 0 0 0 31 21 23 4 28 36 4 47 4 29 33 7 32
9 0 0 0 0 0 31 21 23 4 28 4 36 4 29 33 7 32
9 0 0 0 0 0 0 0 0 0 0 31 21 23 21 33 7 32
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 13

```

## **ANNEXE 3**

## La table d'analyse générée

```

ESSEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEE
EREEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEE
EEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEER
ERREESSEESSEEESSSEREESSSSSESEEEEE
ERREERSSSRREEERRREREERRRRREREEEEE
ERREERRRRRREEERRREREERRRRREREEEEE
SESEEEEEEESSSEEESESESEEEEESESEEE
EREEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEE
EREEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEE
EEEEEEEEEEEEEEEEEEEESEEEEEEEEEEEEE
EREEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEE
EREEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEE
EREEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEE
EREEEEEEEESEEEEEEEEESEEEEEEEEEEEEE
EEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEE
EREEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEE
EREEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEE
EEEEEEEESEEEEEEEEEEEEEEEEEEEEEEEEE
EEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEER
REEEEEEEEEERRREERERREEEEEEREREEE
ERREESRRRRREEESRRREREERRRRREREEEEE
ERREERRRRRREEERRREREERRRRREREEEEE
EESSEEEEESEEEEEEEEEEEEEEEEEEEEESE
EESSEEEEESEEEEEEEEEEEEEEEEEEEEESE
EESSEEEEESEEEEEEEEEEEEEEEEEEEEESE
EESSEEEEESEEEEEEEEEEEEEEEEEEEEESE
EESSEEEEESEEEEEEEEEEEEEEEEEEEEESE
EESSEEEEESEEEEEEEEEEEEEEEEEEEEESE
ERREERRRRRREEESRRREREERRRRREREEEEE
EEEEEEEESEEEEEEEEEEEEEEEEEEEEEEEEE
EESEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEE
EREEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEE
SESEEEEEEESSSEEESESEEEEESESEEE
EESSEEEEESEEEEEEEEEEEEEEEEEEEEESE
EERREEEEEEREEEEEEEEEEEEEEEEEEEEER
EESSEEEEESEEEEEEEEEEEEEEEEEEEEESE
EESSEEEEESEEEEEEEEEEEEEEEEEEEEESE
SESEEEEEEESSSEEESESEEEEESESEEE
EREEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEE
SESEEEEEEESSSEEESESEEEEESESEEE
EERREEEEEEREEEEEEEEEEEEEEEEEEEEER
EERREEEEEEREEEEEEEEEEEEEEEEEEEEER
EERREEEEEEREEEEEEEEEEEEEEEEEEEEER
EERREEEEEEREEEEEEEEEEEEEEEEEEEEER
EEEEEEEESEEEEEEEEEEEEEEEEEEEEEEEEE
EESSEEEEESEEEEEEEEEEEEEEEEEEEEESE
EESSEEEEESEEEEEEEEEEEEEEEEEEEEESE
EEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEE
ERREERRRRRREEERRREREERRRRREREEEEE
SESEEEEEEESSSEEESESEEEEESESEEE

```

## **ANNEXE 4**

## Le fichier des rubans de traduction

```

+$
-$
*$
/$
?$
=$
accept $
to $
select $
goto top~$
skip~$
enddo~$
do while not eof()~$
use $
alias $
->$
goto $
X$
+ 1$
if $
eof()~$
 exit~$
endif~$
locate next 10000 for $
locate for $
= recno()~$
not$
N <> 0~$
to N~$
N <> 0~$
($
)$
else $
do while $
<>$
<$
>$
<=$
>=$
enddo $
~$
count for $
input $
^$
#$
V$

```