Ecole Privée des Sciences Informatiques
de Bordeaux
3 rue du Docteur Gabriel Perri
33000 Bordeaux

Société SynOptis
Technopole Izarbel
64210 Bidart

REALISATION D'UN INTERPRETEUR ET DE SON EDITEUR POUR UN LOGICIEL D'OPTIMISATION

Anthony CALLEGARO

Développement effectué d'Avril à Septembre 2002

Maître de stage
Fabien RODES

Promo 2002

RESTRICTIONS ET RESERVES DE CONFIDENTIALITE

Les informations contenues dans ce document peuvent être modifiées sans préavis et ne constituent aucun engagement de la part de SynOptis et de ses ayants droit.

Les logiciels décrits dans ce document sont fournis par SynOptis en vertu d'une licence SynOptis et ne peuvent être utilisés que dans le cadre défini par les termes de ce contrat.

Aucune partie de ce manuel ne peut être reproduite ou transmise par quelque moyen que ce soit, électronique, mécanique, manuel ou optique ou par tout procédé sans la permission écrite de SynOptis et de ses ayants droit.

SynOptis et ses ayants droit revendiquent le droit de propriété sur ces programmes, ce document et leur documentation en tant qu'ouvrage inédit, leurs révisions ayant déjà fait l'objet d'une licence à la date indiquée lors du précédent avis.

La revendication du copyright n'implique pas l'abandon des autres droits de SynOptis et de ses autres ayants droit.

Toutes les marques sont des marques déposées de SynOptis à l'exception de toutes les autres marques qui sont des marques déposées de leur propriétaire respectif.

Ce document contient des informations confidentielles sur le projet d'entreprise et les produits de SynOptis.

Ce document reste la propriété de SynOptis. Les informations qu'il contient sont fournies exclusivement dans le seul cadre du stage. Son objectif est de servir au processus d'évaluation de la soutenance devant le jury par les personnes habilitées à le faire. Toutes les pages de ce document sont soumises à cette restriction.

Toute autre utilisation n'est possible qu'avec l'autorisation préalable et écrite de :

SynOptis, Technopole Izarbel, 64210 Bidart.

REMERCIEMENTS

Je tiens à adresser mais plus sincères remerciements, au personnel de SynOptis pour m'avoir accueilli au sein de leur société, et tout particulièrement M. Fabien Rodes, le dirigeant, qui fut mon principal contact avec l'entreprise

Je remercie également les employés M. Jean-Marc Elissalde, et M. Olivier Datt qui m'ont fourni, à de nombreuses reprises, les réponses à mes questions concernant l'intégration de mon travail au progiciel SynOptis.

Enfin je remercie les stagiaires Ismaël, Johann, Lionel et Yannick, pour avoir corroborer à l'ambiance si sympathique de la société, et pour m'avoir aidé lorsque je bloquais dans mon développement.

SOMMAIRE

INTRODUCTION

Avant propos

Le document que vous vous apprêtez à lire, se veut être plus qu'un simple rapport de stage, et tente d'être une véritable étude sur la **réalisation d'un langage de programmation**, offrant, même aux néophytes, une très grande puissance, en terme de possibilités de calcul. Ce mémoire synthétise les différentes études et recherches que j'ai effectué, pour la réalisation de ce projet, durant mon stage de fin d'études. Ce stage s'inscrit dans la formation de *Cycle Supérieur d'Ingénierie Informatique – Spécialité Génie Logiciel* dispensée à l'Ecole Privée des Sciences Informatiques (E.P.S.I.) de Bordeaux. D'une durée de six mois, ce stage s'est déroulé d'Avril à Septembre 2002 au sein de la société SynOptis.

Présentation de l'Entreprise

La société SynOptis est basée à Bidart, entre Biarritz et Bayonne, dans les Pyrénées Atlantiques (64), au sein de la technopole Izarbel ; autrement nommé I.D.L.S. (Institut Du Logiciel et des Systèmes).

L'Institut Du Logiciel et des Systèmes

Mis en place en 1985 par la Chambre de Commerce et d'Industrie de Bayonne, l'I.D.L.S. est un organisme de formation supérieure, de recherche industrielle, de création d'activités et de transfert de technologies au service des entreprises.

La vocation de ce centre spécialisé en Informatique Industrielle est de favoriser le développement économique local, le transfert de compétences et l'introduction des nouvelles technologies dans les entreprises, en s'appuyant sur la formation supérieure.

L'I.D.L.S. s'est doté d'un organisme spécialisé en transfert de technologie, l'association Innovation - Logiciel - Système (I.L.S.). Cette association loi 1901 regroupe la Chambre de Commerce et d'Industrie de Bayonne Pays Basque et des représentants du secteur industriel.

Les principales activités du département de développement technologique sont :

- La formation supérieure en informatique

- L'incubateur technologique
- L'E.S.T.I.A. (Ecole Supérieure des Technologies Supérieures Avancées)

Formation supérieure en informatique

Cette formation de troisième cycle comprend 5 filières :

- Gestion et Ingénierie des Projets et Systèmes Industriels : diplôme homologué au niveau 2. Il se déroule sur 1 an et est ouvert aux titulaires de bac+2 expérimentés (3 ans minimum).
- DESS Systèmes de Production Industriels Automatisés : délivré par les universités de Bordeaux I et de Pau – Pays de l'Adour, ouvert aux titulaires de maîtrises scientifiques et technologiques.
- Master of Sciences CAD/CAM/CAE (CAO/CFAO) : délivré par l'université de Cranfield en Grande Bretagne. Formation bilingue anglais – français destinée à des bac+5 désireux d'approfondir leurs connaissances ou d'acquérir une double compétence.
- Ingénieur en Organisation et Gestion Industrielle par l'apprentissage (homologation au niveau 1 de la filière en juillet 96) : cette formation se déroule sur deux ans par l'apprentissage au sein du Centre de Formation pour Apprentis Ingénieurs de la CCI, en collaboration avec l'école d'ingénieurs de Bilbao en Espagne.
- DESS Ergonomie du Logiciel et Informatique Avancée : en collaboration avec l'école polytechnique de Montréal au Canada.

Pépinière et incubateur technologique

L'incubateur est un espace accueillant des porteurs de projets innovants qui visent la création de leur entreprise et bénéficient d'un cycle spécialisé de « formation – action » pour préparer leur projet.

La pépinière technologique « Laminak » abrite de jeunes entreprises à fort potentiel technologiques dont les activités ont un lien avec les domaines de compétence de l'IDLS.

Lors de la première promotion en décembre 96, sur 7 projets, 5 ont conduit à des créations d'entreprises.

L'Ecole Supérieure des Technologies Industrielles Avancées

En octobre 1996, la commission des titres d'ingénieurs donne naissance à l'Ecole Supérieure des Technologies Industrielles Avancées (E.S.T.I.A.). La durée du cycle est de 3 ans. Cette école a contribué à la création d'un pôle technologique autour de l'IDLS en attirant sur place des entreprises de haute technologie tout en favorisant, grâce à son impact local, la compétitivité des entreprises de la circonscription. La cible de l'ESTIA concerne plus précisément les PMI demandeuses de cadres de haut niveau susceptibles de concevoir et de mettre en place des projets liés aux applications de l'informatique et de la gestion industrielle.

La Société SynOptis

Issu de l'incubateur, la société SynOptis fut créée en novembre 1996 par M. Fabien Rodes, ancien élève de l'E.P.S.I. Bordeaux, et M. Fauchet. Ces deux personnes sont toujours associés et employés dans la société, mais c'est Fabien Rodes, qui en est devenu l'unique dirigeant. Depuis 1996 et son lancement SynOptis a prouvé son potentiel et est désormais hébergée dans l'hôtel d'entreprises de l'I.D.L.S.

La société SynOptis oriente principalement son activité sur les progiciels d'optimisation et d'aide à la décision pour des problèmes connus et applicables à diverses industries. Le produit phare de la société est sans contexte le progiciel SynOptis, qui est prévu pour planifier et optimiser les circuits de collecte des ordures ménagères dans de grandes agglomérations ou des communautés urbaines en tenant compte de multiples facteurs (variations saisonnières, temps de parcours, disposition du réseau routier, ...). Il s'agit aussi de minimiser les ressources matérielles et financières nécessaires afin d'affecter les ressources restantes à d'autres tâches. L'intérêt est tout autant économique que technique. Le progiciel a été mis en exploitation sur différents sites comme le district du B.A.B. (Bayonne, Anglet, Biarritz), Lyon, Rennes, ou bien encore Valenciennes.

Ce projet donne lieu à divers travaux dans de nombreux domaines de l'informatique (programmation objet, génie logiciel, base de données, compilation, informatique embarquée, intelligence artificielle, etc.) et des mathématiques (statistique, modélisation stochastique, recherche opérationnelle,…).

Le marché visé par SynOptis est en premier lieu celui des collectivités locales qui sont

demandeuses de tels systèmes. Cette demande a, de plus, été amplifiée par l'apparition du système de la collecte sélective (papier, plastique…) qui nécessite une réorganisation quasi totale du système de collecte des ordures ménagères.

Ce sont en premier lieu les communautés urbaines françaises qui ont été demandeuses du progiciel, mais on assiste à un début d'internationalisation du produit. La concurrence est pour l'instant relativement restreinte (environ 3 concurrents au niveau mondial).

La société emploie à ce jour, six personnes à plein temps, dont quatre développeurs (un mathématicien, et trois informaticiens), les autres étant employés au coté commercial et au secrétariat, plus une personne à mi-temps, qui s'occupe du graphisme de la société, du site Internet, et de la création de la documentation.

La société SynOptis intervenant à tous les stades de la mise en place du projet, toute l'équipe participe à la mise en place du produit chez le client :

- Tout d'abord le commercial et l'ingénieur-conseil définissent avec le client ses besoins et aident à la mise en place des solutions adaptées. Ils vont ainsi interagir avec l'équipe de développement pour fournir une solution en adéquation avec les besoins spécifiques de chaque client.
- Ensuite toute l'équipe doit assurer l'intégration du progiciel chez le client. La société SynOptis fonctionne alors comme un véritable bureau d'études. Il s'agit d'intégrer le progiciel au système informatique existant et de le mettre en œuvre en assurant la continuité avec le système de gestion précédent (nécessite la récupération de données sous plusieurs formats et parfois des développements spécifiques).
- L'intégration est ensuite suivie par une étape de formation chez le client. Cette étape est réalisée par un membre du personnel détaché quelques jours chez le client, afin d'enseigner les fonctionnalités du progiciel.

La partie recherche et développement est assurée par un noyau dur de quatre personnes. Celles-ci vont, en collaboration avec l'équipe commerciale, définir et mettre en place des solutions pour mettre en œuvre les nouvelles fonctionnalités du progiciel, souvent en réponse à un besoin d'un client. Cette équipe va aussi être chargée de la maintenance de l'application et du maintien de la cohérence de celle-ci.

Présentation du progiciel SynOptis

Au-delà d'un outil d'aide à la décision, le progiciel offre également des fonctionnalités que l'on pourrait comparer à celles d'un Système d'Informations Géographiques (S.I.G.).

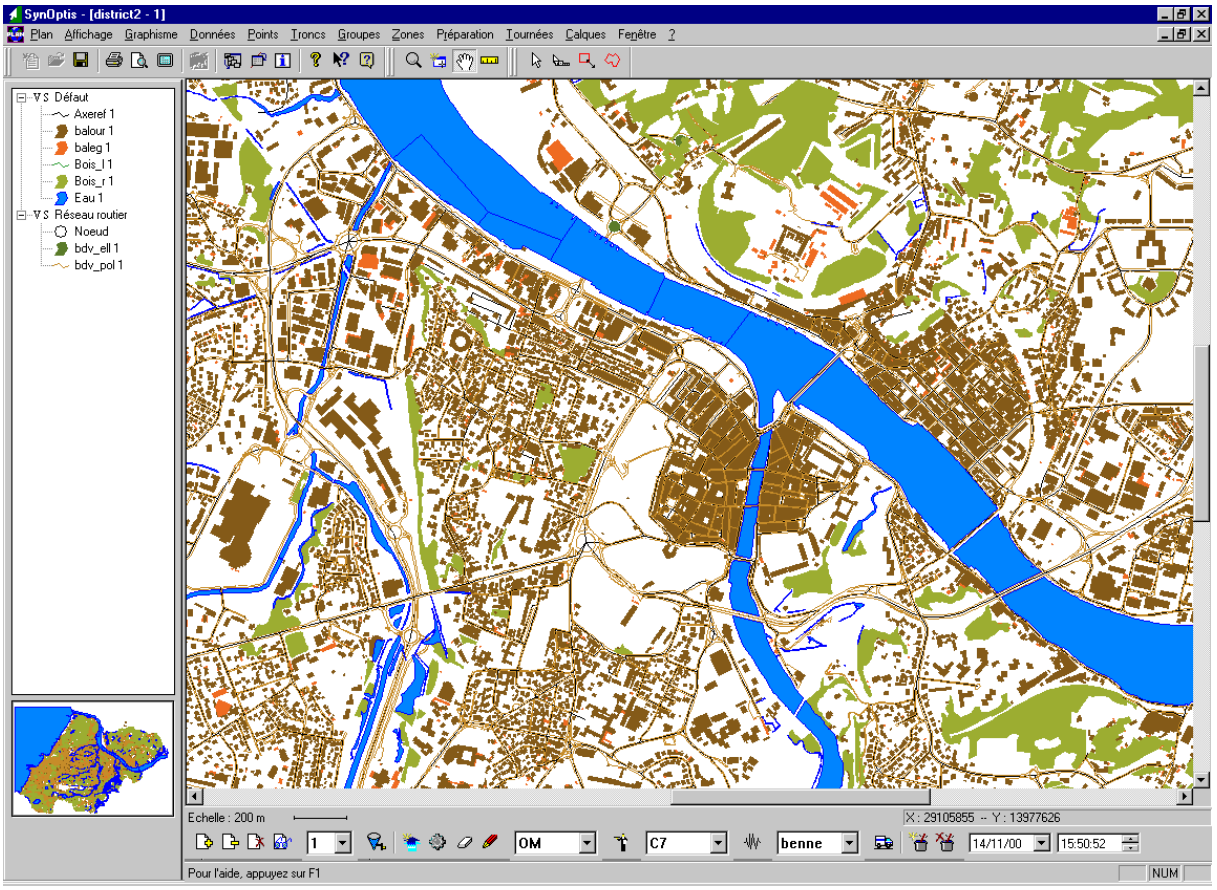


Figure 1 : Aperçu du logiciel

Les S.I.G.

Les S.I.G. sont des systèmes de base de données mêlant des données graphiques et leurs données géographiques associées. Les données graphiques illustrent sous forme de plan ou de carte une localisation géographique précise; les données graphiques peuvent être stockées sous forme de points ou de vecteurs. Dans tous les cas, la cartographie résultante de ces données peut être des points, des lignes, ou des surfaces. Les données géographiques associées caractérisent chaque élément graphique de la carte et peuvent représenter des informations diverses (noms de rues, type d'habitation pour un plan urbain, ou voies d'eau, voies de chemin de fer pour une cartographie de terrain).

Le principe de fonctionnement des S.I.G. est élémentaire : on superpose différentes

couches d'informations graphiques (les voies d'eau en bleu, les routes en noir, les rails de chemin de fer en rouge, etc.). Il est ensuite facile de modifier l'affichage de la carte en donnant divers ordres de priorité entre les couches.

Les S.I.G. sont des systèmes de base de données complexes, dont la gestion reste toutefois difficile.

Le « Formulateur »

Il serait trop long de détailler ici toutes les fonctionnalités qu'offrent le progiciel, je m'attacherai donc à ne vous présenter que la partie me concernant.

SynOptis est, on l'a dit, un progiciel d'optimisation et d'aide à la décision, il permet, par exemple, de modéliser et de visualiser dans le S.I.G. les tournées des collectes des ordures ménagères, générées dynamiquement par le progiciel ou saisies manuellement par l'utilisateur.

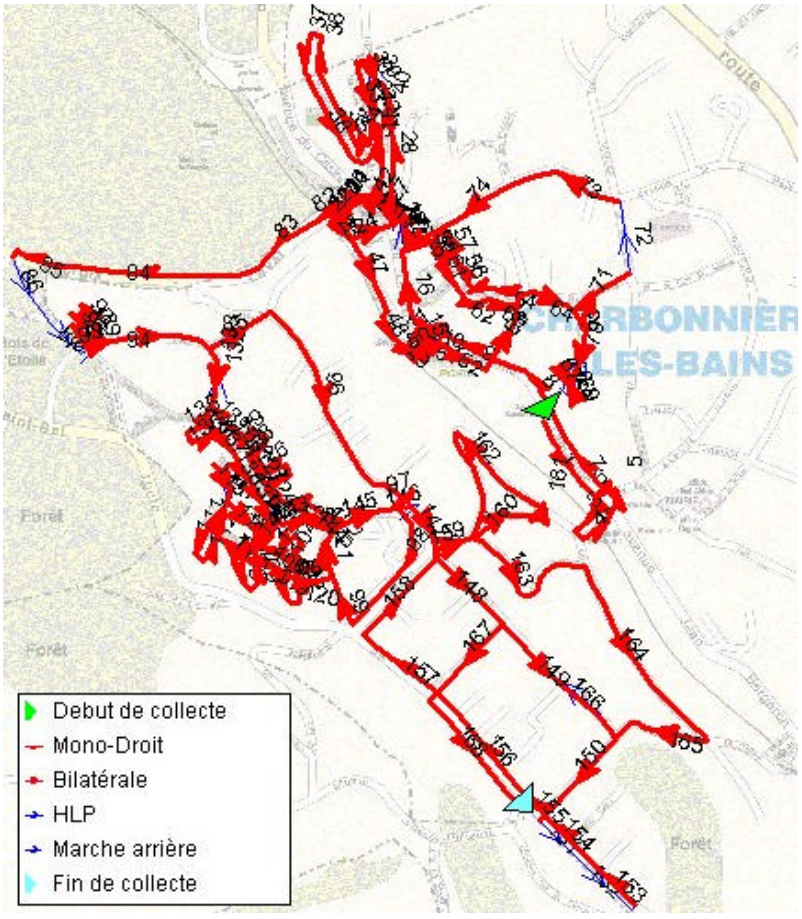


Figure 2 : Exemple de tournée avec un fond de carte

A partir de cette tournée, il est alors possible de visualiser la feuille de route qui sera imprimée pour les hommes de terrain, présentant leurs itinéraires, les rues collectées, et d'autres informations calculées comme le poids prévisionnel d'ordures collectées, les temps de passage et les durées des collectes, ou le volume d'ordures. Tous ces renseignements sont donnés à partir de formules de calcul.

Or, pour un champ spécifique de la base de données, il existe une multitude de méthodes, et donc de formules de calculs différentes ; à cela deux principales raisons :

- Premièrement, chaque utilisateur a une approche différente du problème, et plutôt que d'imposer sa méthode, le progiciel cherche à s'adapter à celle de l'utilisateur.
- Deuxièmement, les formules s'inscrivent dans des unités, ou référentiels différents (par exemple le poids peut s'exprimer en tonne/kilomètre, en tonne/heure, ou encore selon les habitants,…).

Cela fait donc un nombre très variable de formules de calcul possibles, pour un seul champ donné. Il est donc impossible de fixer une formule précise pour un champ sans amputer la flexibilité et la puissance de l'outil.

Pour arriver à palier à cette multiplicité, le progiciel SynOptis s'est doté dès ses débuts d'un « Formulateur ». Ce formulateur est un interpréteur d'un mini langage de programmation interne, qui permet de rentrer ses propres formules, pour calculer chaque champ de la base de données (par exemple le poids à prélever sur la tournée).

Problématique

Le formulateur actuel faisait état de nombreux bugs ou lacunes, que je détaillerai plus bas, qui ont poussé la société SynOptis à lancer un projet de refonte. Ce projet a pour principaux buts :

- De trouver un métacompilateur, rapide et puissant qui permette une bonne intégration dans le logiciel de développement utilisé dans la société SynOptis (Visual C++ 6.0)
- D'implémenter le plus possible l'interpréteur en langage orienté objet, pour la

réutilisation du module dans différents endroits du logiciel.

- De pouvoir récupérer les formules directement en mémoire, pour éviter de passer par le disque dur comme le fait le formulateur actuel.
- D'accroître les fonctionnalités du module existant, en augmentant sa puissance[1].
- De créer un langage simple et puissant qui permettent aux utilisateurs, mêmes néophytes, de pouvoir saisir leurs propres formules de calculs des champs de la base de donnée. Ceci implique de trouver un moyen pour simplifier des notions abstraites comme les relations entre les fichiers de la base de données, les variables et leur types, etc.

En bref, le projet est de réaliser un interpréteur reconnaissant un langage de programmation simple et puissant permettant à n'importe quel utilisateur de saisir ses propres formules de calculs, des champs de la base de données, pour accroître les fonctionnalités et rendre plus flexible le progiciel d'optimisation qu'est SynOptis.

Vous comprenez aisément que la réalisation d'un module comme le formulateur s'apparente beaucoup plus à la création d'un langage de programmation, qu'à un simple développement classique, vu la multitude de cas qu'il faut gérer. Ce projet est pour moi la continuité logique de ma formation à l'école E.P.S.I., puisqu'il rentre tout à fait dans le cadre de ma spécialité (Génie Logiciel et Compilation), et qu'il me permet de me lancer dans une véritable étude théorique de la compilation.

Etat de l'art

L'objectif de ce projet est, on l'a dit, la réalisation d'un interpréteur[2]. Pour cela seront utilisés les principes de compilation suivants : l'analyse lexicale et syntaxique, mais vu qu'il s'agit d'un interpréteur (qui évalue les expressions au fur et à mesure qu'il analyse les formules), il ne s'agit pas d'implémenter toutes les phases d'un compilateur, que je détaillerai

[1] : Ici on parle de puissance en terme de fonctionnalités de l'outil, c'est à dire en terme de degrés de liberté pour l'utilisateur, qui peut effectuer toutes les opérations sur les données qu'il souhaite sans limitation.

[2] : Les notions évoquées ici (interpréteur, compilation,…) seront, rassurez vous, explicitées dans le développement mais vous pouvez déjà en consulter les définitions dans le glossaire.

plus tard.

Ce projet n'est pas destiné à devenir un énième langage de programmation pour initiés (cf. Annexe 1), mais se veut être débarrassé de tous les concepts qui rendent la programmation inaccessible au plus grand nombre. L'outil que je vous présente n'ira pas se poser en concurrent du C ou de Java, mais essaye d'être le plus adapté, et le plus puissant pour son utilisation à l'intérieur du progiciel SynOptis.

C'est pourquoi, malgré la prolifération des langages déjà existant (cf. Annexe 1), sans compter ceux développés pour un usage personnel, aucun ne se trouve adapter à l'intégration dans le progiciel ; ce qui confirme l'utilité de ce développement.

Enoncé des grandes lignes du mémoire

De part la relative complexité et l'étendu du domaine dans lequel j'ai officié, la part de l'étude théorique prend une place très importante dans la conception et la mise en œuvre du projet, et ceux malgré les cours de compilation que j'ai suivi au sein de l'E.P.S.I. En effet, la compilation est un domaine ancien, puisqu'il date des débuts de l'informatique[3], et donc qui a énormément évolué vers des techniques très au point. Mais pour réaliser ce projet il me faut revenir dans le temps, vers des considérations moins modernes, et surtout vers des langages plus anciens qui sont de ce fait plus allégés, et donc plus proche de ce que je cherche à réaliser. Il est difficile, par exemple, de s'inspirer d'un langage comme Java qui est entièrement orienté objet, alors que je recherche à implémenter un langage simple, itératif, structuré mais sans sous-programme (procédures et fonctions).

Donc, mon travail de recherches a du s'orienter vers d'anciennes connaissances, et c'est ce qui constituent la première partie de ce mémoire. Une sorte de retour aux sources, vers les « pionniers » de l'informatique qui ont crées les premiers langages de programmation, avec l'énorme avantage, de profiter des outils de développement récents et stabilisés. La première partie donc, détaillera la compilation, avec, pour vous situer dans un domaine qui ne vous est peut être pas familier, sa définition ainsi que celle des différents

[3] : ANDREW FERGUSON [2000],« The History of Computer Programming Languages. »

concepts qui la composent. Puis je justifierai, le choix que j'ai fait en vous exposant, les apports de cette technique, les énormes avantages qu'elles représentent, mais je n'oublierai pas de vous faire part de ses inconvénients.

La seconde partie présentera les différentes études nécessaires, celles qui constituent la partie génie logiciel et gestion de projet de mon développement. Cette seconde partie, outre les études préalables, exposera les phases de réalisations et de résultats, pour présenter le travail de développement, ses différentes phases (développement de l'interpréteur, et intégration dans le progiciel, etc.) ainsi que les différents résultats obtenus.

Enfin, la conclusion sera le moment pour moi de faire le bilan critique et objectif, bien sûr, de mon travail sur ce projet de six mois, sur lequel je me suis beaucoup démené en recherches et en approfondissement de connaissances. Un projet dont le développement fut prenant et extrêmement enrichissant, et qui j'espère vous passionnera autant que je l'ai été.

PREMIERE PARTIE – LA COMPILATION

I. DEFINITION

1.1 Introduction

« Selon une définition simplifiée, un compilateur est un programme qui lit un programme écrit dans un premier langage – le langage source – et le traduit en un programme équivalent écrit dans un autre langage – le langage cible »[4], telle est la définition que donne Aho, Séthi et Ullman en introduction de leur livre, qui est devenu la bible référence de la compilation, tant ces trois personnes sont des sommités dans ce domaine. C'est donc « simplement » une traduction d'un langage vers un autre, du source (ou code) qui est écrit en langage intelligible par l'homme, vers la cible qui est une suite d'instructions, bas niveau (le langage machine) compréhensible par l'ordinateur. Tel est l'objectif de la compilation : rendre la création de programme, et plus généralement l'exécution de tâches automatisées, plus simple, et c'est également l'objectif que je me suis donné en acceptant la réalisation de ce projet.

D'après ces objectifs, on comprends aisément que la compilation devait avoir un destin commun avec l'informatique, allant chercher ses origines à la même époque, et l'accompagnant jusqu'à aujourd'hui, en étant une des principales raison de son développement. En effet, sans compilation, il n'y aurait pas de langage de programmation, si ce n'est le langage machine, qui reste totalement inabordable même pour des programmeurs expérimentés, et qui cantonnerais la programmation à des petits logiciels de traitement très basiques. Donc, sans langage de programmation, l'ordinateur n'aura pas connu le développement, ni l'engouement extraordinaire qu'il vient de connaître. Je vous propose donc, de faire un petit voyage dans le temps et de remonter l'histoire de l'informatique et de la compilation, pour comprendre comment ce domaine si obscur et méconnu est en fait devenu le facteur essentiel à toute création sur ordinateur.

---

[4] : ALFRED AHO, RAVI SETHI et JEFFREY ULLMAN [1989], « COMPILATEURS Principes, techniques et outils » p15

1.2 Historique[5]

L'histoire de l'informatique et de la compilation a débuté grâce à la passion d'un homme, dont les travaux ont posé les bases de l'informatique moderne, Charles Babbage, un mathématicien qui dès 1834 avait, avec sa machine analytique mécanique, inventé le principe de fonctionnement des ordinateurs tels qu'ils ont été architecturés, par la suite. Il avait surtout crée le principe de programmation, puisque sa machine était composée d'un magasin (mémoire), d'un moulin (unité de calcul), d'une unité de contrôle, de cartes de variables, et de cartes de nombres. Cette découverte lui a valu d'endosser la paternité des ordinateurs actuels. D'ailleurs, beaucoup de personnes lui rendent encore hommage, et on peut même trouver des émulateurs qui reproduisent le fonctionnement de sa machine révolutionnaire (http://www.fourmilab.ch/babbage/applet.html), qui la première permis de programmer le comportement de la machine[6].

Cette invention va apporter beaucoup, notamment à la programmation, car malgré qu'on ne parle pas encore de compilation, le langage machine est bien présent ; certes il s'apparente plus à de l'assembleur, mais il est déjà possible de stocker des valeurs dans des variables. L'évolution importante suivante fut la conception de la notion d'algorithme, par A. Turing qui démontra, en 1936 que tout problème pouvant être mis sous forme d'algorithme, serait résolu par sa machine virtuelle.

Par la suite, on a longtemps cru que le premier ordinateur « véritable » était l'ENIAC, qui en 1946 servait à l'armée américaine à calculer les tables de tir de balistique. Mais on a appris récemment que les anglais avaient conçu, durant la seconde guerre mondiale, le Colossus, qui dès 1943 décryptait les messages codés de l'armée allemande. Cet ordinateur gardé secret à la demande de Winston Churchill, a permis de savoir que les allemands avaient bien cru que le débarquement se ferait dans le Pas de Calais.

Mais on est bien loin de la programmation, ces monstres de 30 tonnes sont, en effet,

---

[5] : FRANÇOIS GUILLIER, [2002], « Histoire le l'informatique », et ANDREW FERGUSON, ibid.

[6] : Depuis Falcon en 1728, et Jacquard en 1806, il existait déjà des machines à tisser, programmées par des cartes perforées en bois, mais la machine analytique permettait carrément d'automatiser des calculs.

programmés physiquement et manuellement, par des commutateurs (l'ENIAC en possédait plus de 6000). Bref, la mise en œuvre d'un calcul demandait plusieurs journées et de solides connaissances, pour l'implémenter sur les circuits.

La révolution des langages va être amenée par celui qui est considéré comme un des plus grands génies de ce siècle, avec Einstein (avec lequel il a travaillé) : John Von Neumann. En effet, en 1945 Von Neumann énonce les principes des ordinateurs modernes, à savoir :

- Les machines doivent être simplifiées au niveau matériel et c'est le programme logiciel qui sera compliqué (pour supprimer les commutateurs manuels)
- Deuxièmement, il postula que les machines devaient pouvoir effectuer des sauts conditionnels (SI <condition> ALORS <expression> SINON <expression>).

Avec ces deux concepts, Von Neumann crée l'informatique moderne ; son architecture est en effet celle encore utilisée dans les processeurs de nos jours, et il créa, par la même, les langages de programmation. L'avancé de Von Neumann[7], fut gigantesque et de nombreuses découvertes partirent de ses concepts.

La progression que connut le domaine, par la suite, fut exponentielle. En 1946 sort le premier langage de programmation en binaire, le Short Code, puis vient l'Assembleur en 1947. Le premier compilateur apparaît en 1951, le A0 de Grace Murray Hopper. Cette femme engagée dans la marine américaine, est considérée comme la mère des langages de programmation. En 1957 viendra le FORTRAN, abréviation de FORmular TRANslator, qui fut le premier langage de programmation reconnu, et qui reste toujours utilisé pour la manipulation de formules mathématiques.

Suivront le LISP en 1959, et le COBOL de G. M. Hopper, puis toute une multitude de langages n'apportant pas beaucoup de nouveautés (cf. Annexe 1). Parmi ceux-ci citons l'ALGOL qui n'a pas été beaucoup utilisé mais qui a servi d'exemple pour le Pascal et le C. L'ALGOL a surtout été le premier dont la grammaire était sous forme BNF (Backus-Naur Form) qui est un précepte très important de la compilation.

---

[7] : La contribution de Von Neumann à l'informatique a été beaucoup moins médiatique et reconnue dans l'histoire que son autre terrible invention : La Bombe A

Enfin autre grande étape de l'histoire de la compilation, en 1969 apparaît UNIX, et avec lui le C en 1972. Pour implémenter le C, on utilise une grammaire BNF, et deux outils qui apparaîtront en 1975 : Lex (un analyseur lexicale) et Yacc (un analyseur syntaxique), qui vont révolutionner le monde de la compilation. Lex et Yacc forment ensemble un métacompilateur (compilateur de compilateur), c'est-à-dire un outil qui génère un langage de programmation. Il n'existe pas de compilateur pour créer Lex & Yacc, mais bizarrement leur code source est écrit en C et en assembleur. Pour finir, avec l'arrivée du monde libre (Linux en 1991) et la licence GNU (GNU's Not Unix, *GNU n'est pas Unix*) GPL (General Public License) ont été développés Flex et Bison, les équivalents libres de Lex et Yacc, qui sont la propriété des laboratoires Bell AT&T.

Depuis il fut crée de nombreux métacompilateurs qui se veulent différents et innovants, mais le standard reste Flex & Bison, pour de nombreuses raisons que j'exposerai plus tard. En tout cas, tous réalisent des compilateurs qui ont un principe de fonctionnement relativement similaire.

1.3 Principes de fonctionnement

Pour comprendre le fonctionnement d'un compilateur, il suffit de le comparer aux langages humains. Lorsque l'on parle, où que l'on écrit, nous nous plions à plusieurs règles :

- des règles lexicales qui définissent les mots, la ponctuation, les symboles que l'on peut utiliser et qui en assurent l'orthographe,
- des règles syntaxiques qui s'assurent que l'on place les mots dans un enchaînement correct, par exemple : sujet – verbe – complément,
- des règles sémantiques qui s'assurent de la cohérence de la phrase, c'est-à-dire que son sens soit correct.

De plus, le compilateur, on l'a dit, est un traducteur d'un langage défini vers le langage que comprend la machine. Par conséquent, cette phase de traduction se compare également tout à fait à celle que l'on opère sur le langage humain :

- le compilateur traduit ce qu'il a lu,
- il optimise ce qu'il a traduit, en enlevant des erreurs de traduction ou de sens, pour une meilleure compréhension de l'interlocuteur (ici la machine),

- enfin il restitue le message au destinataire pour qu'il exécute ce que l'expéditeur lui a demandé.

Donc, on peut résumer tout le processus de compilation, par le schéma ci-dessous qui montre bien les phases successives d'analyses, ainsi que le résultat de chaque phase (en gras italique à coté des flèches). On peut tout à fait faire la comparaison de ce croquis avec le schéma définissant la communication humaine.

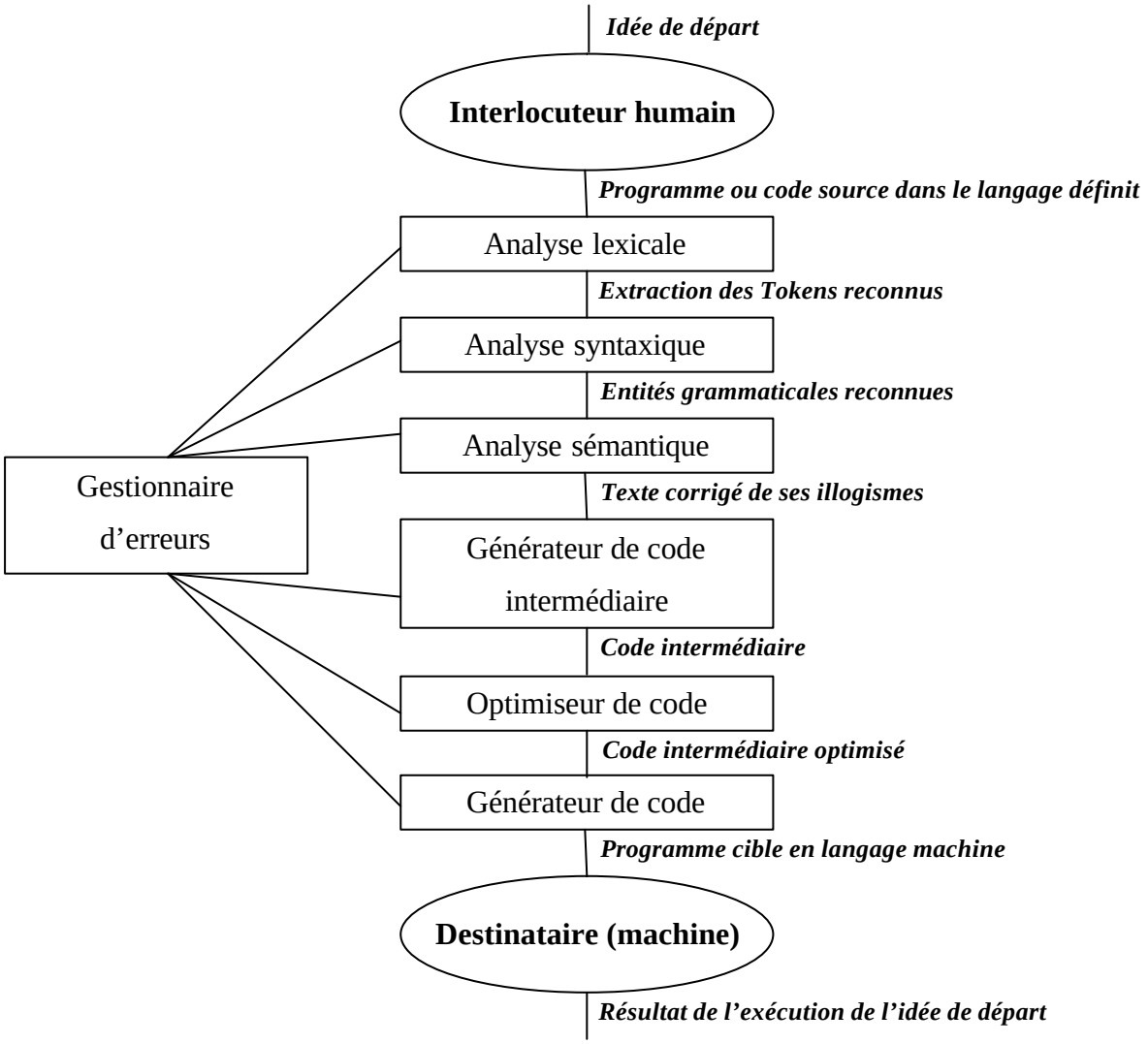


Figure 3 : Phases d'un compilateur

Ce sont toutes ces phases, qu'il faut implémenter, à l'aide d'un métacompilateur, pour réaliser un compilateur complet, ce qui s'avère être un développement très coûteux, en temps, notamment. L'autre solution en compilation est de ne faire qu'un interpréteur du langage.

L'interpréteur ne réalise en fait que les premières étapes d'analyse, à savoir : l'analyse lexicale, l'analyse syntaxique, l'analyse sémantique, ainsi que la gestion des erreurs. Quant à l'exécution du code source, il se fait au fur et à mesure des différentes analyses. C'est-à-dire que l'interpréteur ne fait pas de génération, ni d'optimisation de code.

Dans tous les cas (compilateur ou interpréteur), on définit pour le langage de programmation, comme pour une langue oral, un lexique (vocabulaire), et des règles de syntaxe, que l'on appelle communément grammaire. Tous ces principes peuvent paraître flous, c'est pourquoi je vais vous détailler les concepts qui forment la compilation, ci-dessous.

II. CONCEPTS ET OUTILS

« L'étude des grammaires » et « la théorie des automates » sont les deux principes de bases dont l'utilisation dans l'informatique, et notamment la compilation, a permis de faciliter la relation homme-machine, en simplifiant les communications entre eux. C'est sur ces deux grands concepts que se fonde la partie analytique de la compilation, à savoir l'analyse lexicale, et l'analyse syntaxique.

2.1 Les Grammaires

La grammaire est l'élément de base d'un langage, elle constitue ce qui demande certainement le plus de réflexion, car une seule règle rajoutée peut la rendre instable et ambiguë, ce qui oblige à la repenser entièrement.

2.1.1 Définition

Une grammaire est un ensemble de règles qui définissent les spécificités d'un langage. Par exemple, une des grammaires les plus simples est celle qui reconnaît les expressions arithmétiques :

$E \rightarrow E + T$	règle 1	$E \rightarrow E + T$
$E \rightarrow E - T$	règle 2	$\mid E - T$
$E \rightarrow T$	règle 3	$\mid T$
$T \rightarrow T * F$	règle 4	$T \rightarrow T * F$
$T \rightarrow T / F$	règle 5	$\mid T / F$
$T \rightarrow F$	règle 6	$\mid F$
$F \rightarrow (E)$	règle 7	$F \rightarrow (E)$
$F \rightarrow$ **entier**	règle 8	$\mid$ **entier**

Figure 4 : Grammaires équivalentes définissant les opérations arithmétiques[8]

[8] : Ces deux grammaires misent sous forme sont totalement équivalentes, le « | » est utilisé pour éviter de répéter le non terminal et la flèche (« $E \rightarrow$ »)

Scientifiquement parlant une grammaire non contextuelle ou BNF, est un ensemble comprenant :

- un ensemble de non terminaux : il s'agit de l'ensemble de tous les symboles qui se trouvent au moins une fois à gauche de la flèche, ici « *E*, *T*, *F* »,
- un ensemble de terminaux : il s'agit de l'ensemble de tous les symboles qui ne sont jamais à gauche d'une flèche, ici « +, -, *, /, (, **)**, **entier** »,
- d'un axiome : il s'agit du premier non terminal de la grammaire, ici « *E* »
- et d'un ensemble de règles de production, que l'on écrit souvent sous forme BNF comme ci-dessus.

Pour essayer de vous expliquer le plus simplement possible, la grammaire définit des règles exactement comme si on écrivait pour notre langue française :

phrase → *sujet verbe complément* **point**
sujet → *pronom*
| *nom*
nom → **nom commun**
| **nom propre**
pronom → **je**
| **tu**
…

Figure 5 : Exemple de grammaire pour la langue française

Cela signifie qu'une phrase doit se composer de cette manière, pour être reconnue par le langage, évidemment pour un langage humain, il est beaucoup plus difficile de créer une grammaire, à cause de toutes les exceptions qui existent. Donc dans notre exemple plus haut, le « *E* » peut se composer de « *E* + *T* », on dit que l'on dérive « *E* » en « *E* + *T* ». Quand on réussit à dériver jusqu'à ce que l'on obtienne le même texte que l'on cherche à évaluer, cela signifie que le texte est conforme aux règles de grammaires que l'on a définie, on appelle cette méthode : analyse descendante. Pour mieux comprendre voici un léger exemple :

On cherche à savoir si « (4 + 5) ∗ 3 » est une expression arithmétique correcte, pour cela on va essayer de dériver à partir de l'axiome de la grammaire.

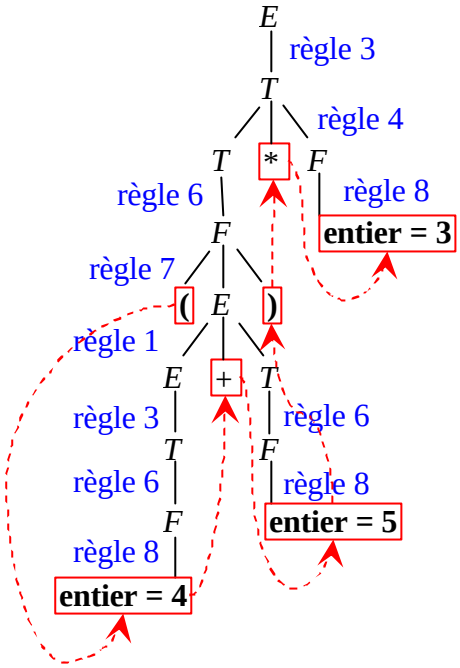


Figure 6 : Dérivation (analyse descendante) de (4 + 5) * 3

Ici «(4 + 5) ∗ 3 » est une opération arithmétique correcte car elle est reconnue par la grammaire quand on l'a dérive. En effet, si on parcourt en profondeur le contenu des cases rouges, selon le trajet des flèches, on obtient « **(** », « **4** », « + », « **5** », « **)** », « * », « **3** ».

Si on prend pour autre exemple « 2 + (4 * 5 » l'expression n'est évidemment pas correcte car il manque une parenthèse fermante. Si on essaye tout de même de la dériver, on obtient l'arbre d'analyse suivant :

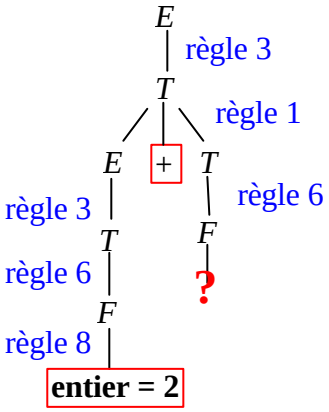


Figure 7 : Dérivation de 2 + (4 + 5

Ici arrivé à l'état avant le point d'interrogation, on ne trouve pas de règle à partir de F qui commence par une parenthèse ouvrante, mais qui ne se ferme pas, c'est un échec. En

conséquence le mot n'appartient pas à la grammaire.

Voici donc en résumé la définition d'une grammaire, de sa mise en forme et surtout de la méthode qu'elle se sert pour vérifier l'appartenance d'une expression au langage qu'elle décrit. Nous verrons plus tard qu'il existe une autre méthode que l'on nomme analyse ascendante, car contrairement à celle que nous venons de voir, qui tente de reconnaître l'expression en partant de l'axiome qu'elle dérive, l'analyse ascendante part de l'expression à reconnaître et cherche par réduction successive à rencontrer l'axiome. L'analyse ascendante sera détaillée dans la partie dédiée à l'analyse syntaxique.

La plus grosse difficulté lors de la conception de la grammaire est surtout d'éviter les ambiguïtés, qui peuvent causer parfois de graves défaillances dans la reconnaissance du langage.

2.1.2 Ambiguïté de la grammaire

Une grammaire est dite ambiguë si, pour une même phrase, elle donne plusieurs arbres de dérivation. Ce cas se produit si on a mal structuré les règles de production, de la grammaire, ou des fois sur des cas particuliers, comme sur le problème du « sinon en suspens » (cf. exemple plus bas).

Le problème de l'ambiguïté réside dans le fait que l'analyseur syntaxique qui reconnaît la grammaire, ne sait pas ce qu'il doit faire, vu qu'il a plusieurs actions à exécuter. Souvent les analyseurs syntaxiques choisissent une action par défaut dans cette situation. Dans la grammaire du C, il existe une ambiguïté dite du «sinon en suspens », or yacc (ou bison) avec lequel est fait la grammaire du C, applique la bonne solution.

Ce genre de comportement n'est pas vraiment acceptable, c'est pourquoi on essaye d'éviter toute ambiguïté dans la conception de la grammaire, car il n'est pas évident que l'analyseur syntaxique prendra toujours la bonne décision, et qu'ainsi il reconnaisse exactement, le langage qu'on lui a défini.

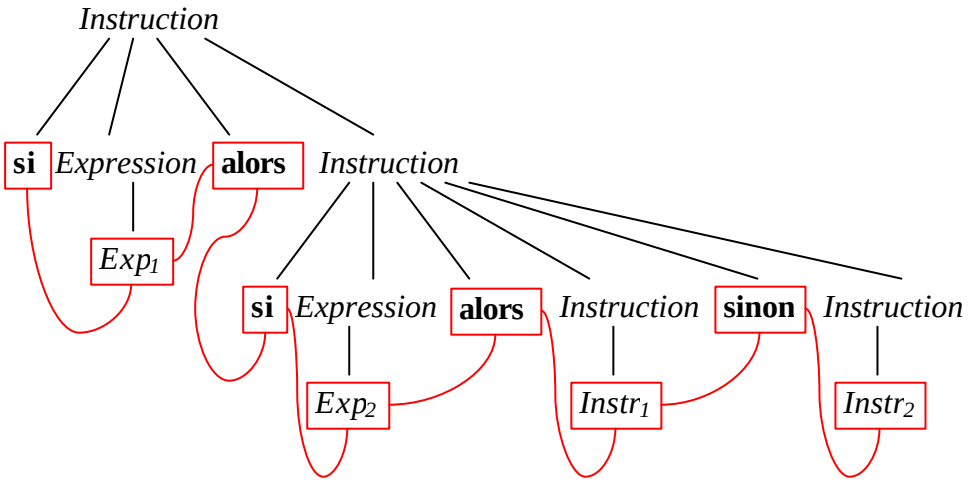
Par exemple, voici un extrait de la grammaire du C, en l'occurrence il s'agit d'un extrait du code source de gcc (compilateur C de linux), que j'ai traduit pour être plus compréhensible, et qui illustre le problème du « sinon en suspens » :

Instruction → **si** *expression* **alors** *instruction*
| **si** *expression* **alors** *instruction* **sinon** *instruction*
| **autre**

Figure 8 : Grammaire ambiguë du "sinon en suspens"

Ici «autre » est mis pour n'importe quel autre instruction. Cette grammaire répond au besoin des langages de proposer des instructions conditionnelles, qui n'ont pas obligatoirement un «sinon ». Or, si on prend l'exemple suivant : «**si** Exp_1 **alors si** Exp_2 **alors** $Instr_1$ **sinon** $Instr_2$ », cette phrase a plusieurs arbres de dérivation (voir figure ci-dessous) :

Première possibilité :



Deuxième possibilité :

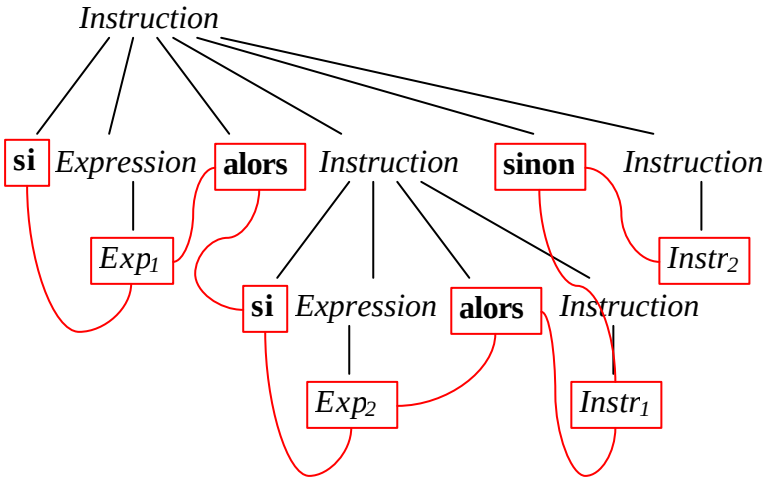


Figure 9 : Deux arbres d'analyse pour une grammaire ambiguë

Ici le parcours en profondeur des deux arbres d'analyse, donne la même phrase : « **si** », « *Exp*$_1$ », « **alors** », « **si** », « *Exp*$_2$ », « **alors** », « *Instr*$_1$ », « **sinon** », « *Instr*$_2$ » ; on est donc bien en présence d'une ambiguïté, c'est-à-dire que la phrase est syntaxiquement correcte, mais que l'analyseur syntaxique n'est pas capable de faire la différence entre les deux possibilités que l'utilisateur aurait voulu différencier, ce qui peut s'avérer être très grave dans l'interprétation de la phrase. En effet dans le premier cas, le « **sinon** » est rattaché au second « **si** », et dans le second cas, le « **sinon** » est rattaché au premier « **si** ». Ainsi vous comprendrez l'importance d'éviter absolument les ambiguïtés dans la grammaire, si on veut réaliser un langage fonctionnel et stable[9].

2.2 Les automates

Issu des mathématiques, comme souvent, la théorie des automates est la méthode qui a permis les meilleurs résultats au niveau de l'analyse lexicale, ou bien encore, dans la reconnaissance vocale.

2.2.1 Définition

Simplement, un automate est un graphe, ayant comme arc les lettres de l'alphabet, qui définissent le vocabulaire, et un ensemble d'état, qui composent les sommets du graphe. Il y a également des états de départs et finaux. Enfin une fonction associe les sommets selon la lettre (ou symbole) qui les relient. Voici un exemple simple d'automate :

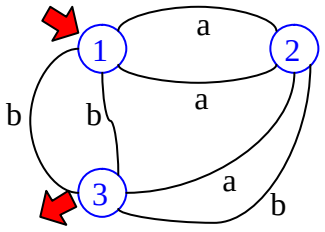


Figure 10 : Exemple d'automate simple

[9] : Le même exemple est repris dans le livre d'ALFRED AHO, RAVI SETHI et JEFFREY ULLMAN [1989], ibid p200-201

Sur le schéma ci-dessus, l'alphabet est composé par « a » et « b », donc le langage ne reconnaîtra que des mots contenant « a » ou « b » ou les deux. Ensuite, l'ensemble des états comprend « 1 », « 2 », « 3 » ; « 1 » étant un état de départ, et « 3 » étant un état final. Enfin, la fonction de transition que l'on nomme souvent δ sera définie dans cet exemple par δ(1, a) = 2, à savoir de l'état 1 par la lettre « a », on arrive à l'état 2. On note également pour faire plus simple (1, a, 2).

Que veulent dire toutes ces définitions ? Et bien, pour mieux comprendre prenons un exemple : le mot « baab » appartient-il au vocabulaire de notre langage ?

Caractère en entrée	Sommet en cours	Fonction d	Action
® baab	Départ : (1)		On démarre de l'état de départ
→ **b**aab	(1)	((1), **b**, (3))	Depuis (1) par **b** on va dans l'état (3)
→ **a**ab	(3)	((3), **a**, (2))	Depuis (3) par **a** on va dans l'état (2)
→ **a**b	(2)	((2), **a**, (1))	Depuis (2) par **a** on va dans l'état (1)
→ **b**	(1)	((1), **b**, (3))	Depuis (1) par **b** on va dans l'état (3)
→ **Vide**	(3) est un état final		On arrive sur un état final, et il n'y a plus de caractère en entrée donc le mot est reconnu.

Figure 11: Reconnaissance du mot "baab" par un automate

Comme commenté dans la dernière ligne, le mot est reconnu si, et seulement si, on est sur un état final (ici 3) et qu'il ne reste plus de caractère en entrée. Par exemple, un mot comme « ba » n'est pas reconnu car quand le flux d'entrée est vide on est sur le sommet 2 qui n'est pas un état final. « ab » par exemple n'appartient pas non plus au vocabulaire car depuis l'état 2 il n'y a pas d'arc qui parte avec comme lettre « b ».

Voici la méthode qu'utilisent les analyseurs lexicaux pour reconnaître le flux d'entrée, qu'on lui fournit que ce soit dans un compilateur (interpréteur), ou bien dans un outil de

reconnaissance vocale, pour découper l'entrée en unité lexicale, plus simple à manipuler par l'analyseur syntaxique.

Mais comme pour les grammaires, il n'est guère facile d'écrire les règles lexicales d'un langage, et les erreurs de conception sont fréquentes. Ici on ne parlera pas d'ambiguïté mais de déterminisme de l'automate.

2.2.2 Déterminisme de l'automate

On dit qu'un automate est déterministe s'il possède un seul état d'entrée, et si d'un même état, deux arcs ne partent pas avec la même lettre, à savoir : voici le même automate que tout à l'heure mais non déterministe (les éléments responsables sont signifiés en rouge) :

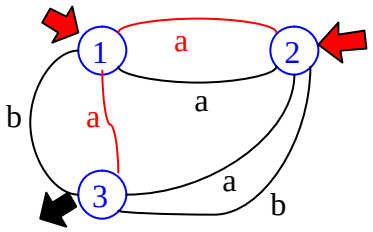


Figure 12 : Un automate non déterministe

Ici l'automate n'est pas déterministe, car l'ensemble des sommets de départ contient plus d'un seul élément, mais également, parce que du sommet 1, il y a deux arcs «a » qui partent.

Dans le cas d'une grammaire ambiguë, l'analyseur syntaxique, choisissait arbitrairement une solution ; c'est également ce que fait l'analyseur lexical dans le cas où il se trouverait en présence de règles définissant un automate non déterministe. En général, les analyseurs lexicaux choisissent la règle la plus longue, et en cas d'égalité, ils prennent la première déclarée. Il faut faire très attention, au fait que ce ne soit pas toujours la première règle qui l'emporte, car cela peut devenir une grande cause d'erreurs.

Il existe bien entendu des méthodes pour rendre un automate non déterministe, en son contraire, tout comme il existe des méthodes pour rendre une grammaire non ambiguë, mais je n'en parlerai pas, car je ne veux pas vous embrouiller avec des méthodes compliquées et

peu intéressantes pour un non initié. Sachez seulement qu'il existe une multitude de solutions pour accélérer les traitements sur les automates, et sur les grammaires, mais qu'heureusement les métacompilateurs, comme nous le verrons plus tard, évitent de réimplémenter ces algorithmes très coûteux en temps.

Après avoir vu ces deux théories de base, intéressons nous aux concepts qui se basent dessus pour réaliser la partie analytique du compilateur.

2.3 L'analyse lexicale

Concept fondamental de la compilation, l'analyse lexicale est la première étape de la reconnaissance d'un langage.

2.3.1 Définition

En compilation l'analyse lexicale se charge de découper le flux d'entrée (fichier pour un compilateur, ou clavier pour un interpréteur) en unité lexicale, c'est-à-dire en « mots » ou « tokens », qui sont les plus petites parties exploitables par l'analyseur syntaxique avec lequel il collabore. C'est dans cette partie, que l'on définit le vocabulaire du langage, à savoir les mots, et symboles qu'il reconnaît. L'analyse lexicale est également très utilisée en reconnaissance vocale, où il a pour rôle de découper les phrases en phonèmes.

L'analyseur lexical est un automate composé de règles, qui définissent les mots et les symboles, reconnus par la syntaxe. Son principe de fonctionnement est assez simple :

- il analyse le flux d'entrée jusqu'à trouver une concordance entre le mot et une règle définissant une unité lexicale, grâce à son automate, et la méthode que l'on a vu précédemment,
- une fois le mot reconnu, l'analyseur lexical, renvoie à l'analyseur syntaxique, le token (unité lexicale),
- l'analyseur syntaxique essaye d'avancer dans sa reconnaissance du langage, puis redemande la prochaine unité lexicale en entrée.

C'est pourquoi on définit souvent l'analyse lexicale comme une sous-routine de

l'analyse syntaxique, mais bien que l'analyse syntaxique puisse se passer d'analyse lexicale, cette dernière la simplifie grandement.

En outre, l'analyse lexicale permet de réaliser des actions quasiment impossibles avec l'analyse syntaxique, comme la vérification de l'existence d'une variable, ou son initialisation.

En résumé, l'analyse lexicale a pour but de lire le flux d'entrée et de produire comme résultat, une suite d'unités lexicales que l'analyseur syntaxique va exploiter. Nous allons maintenant explorer plus en détail son mode de fonctionnement.

2.3.2 Utilité de la méthode et fonctionnement

Il est vrai qu'il est tout à fait possible de créer un langage où la partie d'analyse syntaxique regrouperait également l'analyse lexicale. Mais le découpage du programme source en unité lexicale, présente de nombreux avantages :

- la simplification de la conception, en est sans doute le plus important, en effet il est beaucoup plus simple de supprimer du texte en entrée, les espaces, les tabulations, ou les commentaires dans la phase d'analyse lexicale, et éviter ainsi d'encombrer l'analyse syntaxique avec la considération des blancs.
- L'efficacité du compilateur se trouve bien entendu accrue par la diminution des traitements dans la partie analyse syntaxique,
- de plus la portabilité du compilateur est d'autant plus importante que les problèmes de caractères spéciaux, spécifiques au matériel (clavier), se confinent à l'analyseur lexical, et sont beaucoup moins difficiles à maintenir.
- Enfin le regroupement de type de tokens entre eux, allègent le code de l'analyseur syntaxique, à savoir que les caractères «espace », «tabulation », et « retour chariot » par exemple, retourneront tous le token « blanc » et ne seront pas pris en compte. Cela évite donc d'avoir à répéter le code trois fois dans l'analyseur syntaxique.

Vous comprendrez donc, qu'il est fort utile d'utiliser l'analyse lexicale, car elle clarifie énormément le code source du langage et, de plus son fonctionnement est relativement simple. Vous vous en doutez, il ne faut pas saisir un par un, tous les mots reconnus par le

langage, car cela serait totalement impossible, en ce qui concerne notamment, les identificateurs des variables. L'analyseur lexical, a donc recours à des symboles réservés, qui définissent, la position et la répétition des lettres, un peu comme l'étoile « * » sous environnement Windows. En effet, il est possible d'effectuer, sous Windows (ou sur d'autres systèmes, heureusement), des recherches partielles sur les mots, en connaissant des bribes, et en remplaçant les parties inconnues par «* ».Ainsi la recherche de « m*.doc » donnera tous les fichiers dont le nom commence par «m » et se finit par « .doc ». Dans les analyseurs lexicaux, c'est quasiment la même chose mais il existe beaucoup plus d'options de traitement de la chaîne (cf. Annexe 2 : Les expressions régulières de Lex[10]).

Une fois les règles d'analyse lexicales saisies, il faut programmer son comportement, c'est-à-dire associer des actions à la reconnaissance d'un mot. L'action basique, et la plus répandue, consiste à renvoyer l'unité lexicale, correspondant à ceux que l'on vient de reconnaître à l'analyseur syntaxique. Mais ce serait cantonner l'analyseur lexical au rôle de sous-routine, que j'évoquais plus haut. En effet, ce dernier prend souvent en charge la gestion de la table des symboles, à savoir la liste des variables définies par l'utilisateur, et d'autres fonctionnalités comme une partie de la gestion des erreurs, mais aussi, la comptabilisation des lignes et des colonnes, pour mieux expliciter les erreurs renvoyées. Voici un léger exemple de règles lexicales écrites en lex :

blancs	[\t\n]+	®	Signifie que tous les espaces « », tabulation « \t », et retour chariot « \n » seront des blancs.
lettre	[A-Za-z]	®	Englobe toutes les lettres minuscule et majuscule.
chaîne	{lettre}+	®	Une chaîne est une répétition de lettres
chiffre	[0-9]	®	Reconnaît les chiffres seuls.
entier	{chiffre}+	®	Reconnaît les nombres entiers, à savoir une suite de chiffres continu

[10] : Lex est en effet un générateur d'analyseur lexical, dont on parlera plus longuement plus bas, mais c'est un bon exemple de syntaxe d'analyseurs lexicaux, car il permet une multitude d'opérations pour filtrer, les caractères en entrée.

{blancs}	{}	®	Signifie que tous les blancs seront ignorés (pas d'action associée).
{chaîne}	{action ; return(CHAINE) ;}	®	On exécute l'action voulue et on renvoie l'unité lexicale CHAINE.
{entier}	{action ; return(ENTIER) ;}	®	Idem, on exécute une action définie et on renvoie ENTIER.
.	{renvoyer une erreur ;}	®	Si on ne reconnaît aucun mot (symbolisé par le point) alors on renvoie une erreur.

Figure 13 : Exemple de règles lexicales et des actions associées

Voici donc, un très léger exemple pour bien comprendre les rouages de l'analyse lexicale. Tout d'abord, on définit des types pour éviter de répéter les règles à chaque fois (dans le premier cadre). Puis une fois ces « types » définis on leur associe une action. Littéralement la seconde ligne dans le cadre ci-dessus signifie : « Si l'on rencontre une suite de caractère minuscule ou majuscule, alors on exécute une action spécifique, puis on renvoie à l'analyseur syntaxique que l'on a rencontré une CHAINE ».

Comment l'analyseur syntaxique utilise ces unités lexicales ? C'est ce que nous allons voir dans la prochaine partie.

2.4 L'analyse syntaxique

Intervenant en collaboration avec l'analyse lexicale, l'analyse syntaxique est certainement la phase la plus importante de la compilation.

2.4.1 Définition

Les définitions sur l'analyse syntaxique sont pléthore dans les livres ou bien sur Internet mais toutes semblent être différentes. Certaines prétendent que l'analyse syntaxique

englobe toute la partie analytique de la compilation (lexicale et syntaxique), et nomment cette sous-phase analyse sémantique, alors que d'autres l'appellent simplement analyse syntaxique. Ne préférant pas rentrer dans un débat inutile, et ne voulant pas causer de problème de compréhension de votre part, je précise donc que je nomme « partie analytique » la phase qui regroupe l'analyse lexicale et l'analyse syntaxique, et utilise indifféremment les termes « syntaxique » ou «sémantique » pour la seconde sous-phase, me ralliant prudemment au point de vue d'Aho, Sethi et Ullman, qui font autorité dans le domaine de la compilation.

L'analyse syntaxique est donc la phase qui reconnaît si le texte source est conforme aux règles de la grammaire définie. Pour exécuter cette tache, l'analyseur syntaxique reçoit les unités lexicales (tokens) de l'analyseur lexical, exécute un algorithme de reconnaissance (cf. 2.1 Les Grammaires), puis demande le token suivant, jusqu'à reconnaître le source. Durant l'analyse il est courant de rencontrer des erreurs, mais il n'est pas acceptable que l'analyseur s'arrête dès la première erreur, car il faut qu'il réussisse à fournir le plus possible d'informations sur les erreurs qui suivent. Par conséquent, le code de gestion d'erreurs doit être capable de se récupérer après un problème dans le processus d'analyse.

Ensuite, dans le cas d'un interpréteur, le code sera exécuté au fur et à mesure de la reconnaissance ; ou bien dans un compilateur, l'analyse produit des informations pour la génération de code qui doit suivre. En tout les cas, une action (code source) est associée à chaque règle, il est également possible d'associer plusieurs blocs de code pour une seule règle.

Il existe ensuite différents modes de fonctionnement, mais les analyseurs syntaxiques les plus performants, sont ceux qui fonctionnent par analyse ascendante.

2.4.2 Utilité de la méthode et mode de fonctionnement

Il est extrêmement important d'utiliser les grammaires pour formaliser les règles de productions de l'analyseur. De nos jours la plupart des métacompilateurs, ne proposent pas d'autres moyens de modélisation. Quelques avantages de cette formulation :

- La grammaire donne une très grande précision de la syntaxe tout en restant facile à comprendre
- Suivant les métacompilateurs utilisés, il est possible de trouver les ambiguïtés de

la grammaire, au moment de la génération de l'analyseur syntaxique.

- Le langage ainsi construit est beaucoup plus évolutif.

Le mode de fonctionnement de l'analyseur est également crucial au niveau des performances, car comme cité ci-dessus, les analyseurs par ascendance sont les plus véloces.

L'analyse par ascendance est exactement l'inverse de l'analyse descendante (d'où son nom) vu au chapitre 2.1, à savoir qu'ici on part des feuilles de l'arbre pour essayer de reconnaître l'axiome par décalage ou réduction. Un décalage est l'action de rajouter le token suivant, à l'expression en cours de reconnaissance, alors que la réduction s'opère lorsque l'expression est identique à la partie droite d'une règle de production de la grammaire. Pour mieux comprendre, reprenons l'exemple de la figure 6 : reconnaître « (4 + 5) * 3) » avec la grammaire « ETF » de la figure 4. En utilisant l'analyse ascendante nous obtenons, le schéma suivant :

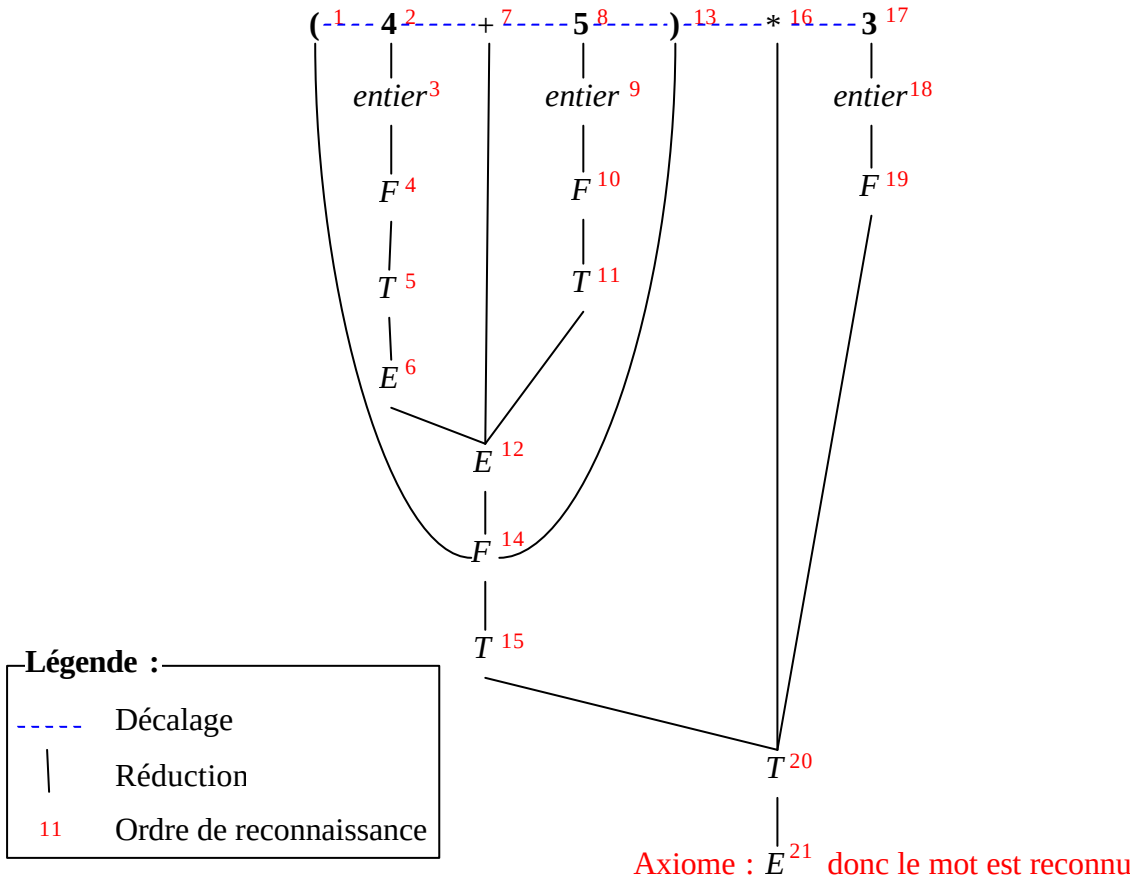


Figure 14 : Analyse ascendante de « (4 + 5) * 3

Le schéma ci-dessus se lit de haut en bas, car cela le rend plus explicite sous cette forme, même s'il est vrai que cette figure ne rend pas vraiment compte, visuellement, du caractère ascendant de l'analyse. Les numérotations en rouge témoignent plus de cet état ; en les suivants on détermine l'enchaînement suivant :

- l'analyse démarre des feuilles du premier token,
- l'analyseur réduit jusqu'à être bloqué,
- ensuite il décale, c'est-à-dire qu'il demande le prochain token en entrée à l'analyseur lexical,
- puis il retente de nouvelles réductions, jusqu'à obtenir l'axiome de la grammaire seul, et une entrée vide, ou bien de tomber sur une erreur.

Cette méthode ascendante est surtout utilisée dans les métacompilateurs, qui génèrent des analyseurs lexicaux ayant ce mode de fonctionnement. Sinon, pour une analyse manuelle sur papier, c'est la méthode d'analyse descendante qui prime.

En ce qui concerne les actions associées aux règles de grammaire, elle se situent généralement en fin de règles, mais elles peuvent également se placer entre deux tokens de la règle, pour être exécutées lorsque ces tokens seront reconnus. En effet, les actions de fin ou intermédiaire sont utilisées par l'analyseur syntaxique comme des composantes de la règle, et donc se retrouvent dans l'arbre d'analyse, et s'exécutent lorsqu'elles sont reconnues. Par exemple, soit la syntaxe suivante :

règle$_1$ ® *règle*$_2$ **point_virgule** {action de fin$_1$;}
règle$_2$ ® *partie*$_1$ {action intermédiaire ;} *partie*$_2$ {action de fin$_2$;}

On aura l'arbre d'analyse suivant :

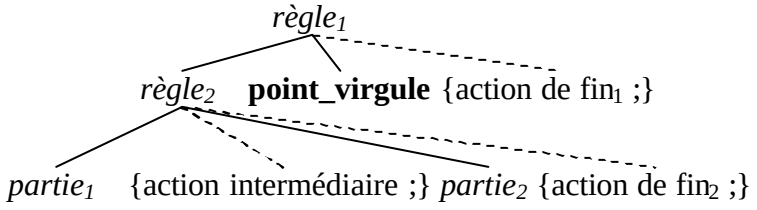


Figure 15 : Exemple d'action dans l'analyse syntaxique

Ici l'analyseur reconnaît « *partie*$_1$ », puis il effectue un décalage, et donc reçoit l'unité lexicale « *partie*$_2$ », à ce moment là il trouve le code de l'« action intermédiaire », détectant que ce n'est pas un token, il exécute l'action spécifiée. Puis une fois l'unité « *partie*$_2$ » reconnue, il demande le prochain token, et reçoit « **point_virgule** ». A cet instant précis, l'analyseur détecte l'« action de fin$_2$ » et l'exécute avant de réduire en « *règle*$_2$ », etc.

Une fois réduit jusqu'à l'axiome de la grammaire, toutes les actions ont été exécutées, et donc le compilateur est prêt à générer le code intermédiaire à partir du code source, épuré de ses erreurs, qui est désormais, bien conforme aux spécifications du langage.

2.5 La génération de code

La génération de code est certainement la phase la plus compliquée à implémenter, c'est pourquoi j'éviterai de m'appesantir dessus.

2.5.1 Définition et débats

La génération de code, au sens large, est la phase qui regroupe toute la seconde partie de la compilation, à savoir :

- la génération de code intermédiaire,
- l'optimisation de code,
- la génération de code final.

Cette phase difficile, est absente des interpréteurs et ne concernent donc, que les compilateurs. En effet, dans un interpréteur, l'exécution du code source, se fait au fur et à mesure de l'analyse syntaxique, tandis que le compilateur, une fois implémenté cette phase, génère un fichier exécutable (programme) que l'utilisateur lance ensuite indépendamment. Créer un compilateur est surtout avantageux en terme de rapidité, car un langage compilé est plusieurs milliers de fois plus rapide qu'un langage interprété, puisqu'il est traduit en langage machine. Le problème du compilateur réside dans sa faible portabilité, car il génère du langage machine qui est dépendant, de l'architecture matériel, du processeur, et du système d'exploitation. De surcroît, son utilisation reste réservé à des initiés, car il faut, une fois le code source rentré, compiler le code en exécutable, puis exécuter le programme ainsi crée

pour avoir le résultat.

L'interpréteur au contraire, peut sembler un bon compromis, car sans phase de génération de code, il évalue, et renvoie les résultats du code source en direct, en même temps que l'analyse s'exécute. Le problème des performances ne se remarque pas tant que l'on ne lui fait pas évaluer plusieurs milliers de lignes de codes, et même dans ce cas là le temps de traitement ne dépasse pas quelques secondes. En contre partie, l'interpréteur possède un coût de temps de développement énormément plus faible que celui d'un compilateur.

Bref, si on ne veut pas réinventer le C++, l'interpréteur peut sembler une bonne stratégie d'implémentation qui diminue de façon très significative, le temps de développement. Il est par contre essentiel, si le langage est destiné à un usage commercial auprès de professionnel, de ne pas éluder la phase de génération de code.

2.5.2 Fonctionnement

Grâce aux actions associées aux règles de l'analyse syntaxique, et à la table des symboles, construite durant les phases analytiques, est élaboré un graphe de programme. Ce graphe est une suite d'instructions linéaires traduites en langages intermédiaires, qui n'est alors composé que de conditions simple (« if » sans « sinon »), épuré des boucles, et ne fonctionnant qu'avec des sauts (« Goto »). Ces sauts constituent les arcs de notre graphe.

Une fois le graphe du programme effectué, on recherche les boucles dans le graphe, et les parties non accessibles. C'est ici qu'on optimise le code, en éliminant le code superflu (non atteint).

Enfin, la dernière partie est totalement dépendante du matériel et du système d'exploitation de la machine. En effet, le compilateur doit être recompilé sur tous les systèmes différents, sinon il ne marchera pas (de Windows à Linux par exemple). De plus, pour générer du langage machine dans cette partie, les instructions processeurs sont différentes d'un système à l'autre (processeur Motorola sous Mac, Intel ou AMD sous Windows et Linux, ou bien Sparc sous Unix, etc.).

Ainsi, la génération de code n'est absolument pas portable, d'un système vers un autre. C'est pourquoi il existe pour tous les langages des compilateurs différents pour chaque

système.

2.6 Les autres concepts

Pour finir avec cette présentation théorique de la compilation, voici les derniers concepts importants, que j'ai déjà abordé légèrement.

2.6.1 La gestion d'erreurs

Le code de gestion d'erreurs, regroupe toutes les instructions et procédures destinées à gérer et à récupérer les erreurs, et essayer de remplir les objectifs suivants :

- détecter le maximum d'erreurs que l'utilisateur pourrait faire en programmant,
- l'informer le plus possible sur la source de l'erreur, son type, et son emplacement,
- proposer une solution de résolution du problème,
- réussir à reprendre l'analyse malgré une erreur,
- détecter les erreurs qui ont eu lieu après la première erreur signalée.

La bonne gestion et récupération d'erreurs, forment un grand défi pour le concepteur d'un langage de programmation, car c'est une phase extrêmement délicate à implémenter. A cela plusieurs raisons : les erreurs peuvent apparaître à tout moment de la partie analytique, et même dans la génération de code (cf. Figure 3), deuxièmement la multitude de types d'erreurs, et leur polymorphisme, les rendent difficile à localiser. On peut, tout de même les réunir en plusieurs grandes familles :

- Les erreurs lexicales regroupent les mots ou les symboles non reconnus par le vocabulaire, ou simplement mal orthographié.
- Les erreurs syntaxiques regroupent les erreurs de grammaire, par exemple un mot reconnu mais mal placé, ou un mot manquant (expression arithmétique mal parenthésée).
- Les erreurs sémantiques sont des erreurs de typage, par exemple l'utilisation d'un opérateur arithmétique sur une chaîne de caractères.
- Les erreurs logiques sont des erreurs de programmation de l'utilisateur (une

boucle infinie par exemple).

- Les erreurs mathématiques regroupent les divisions par zéro, ou l'utilisation de fonctions sur des intervalles pour lequel elle n'est pas définie.
- Les erreurs de compilation sont provoquées par un fichier inclus manquant, ou une variable non définie.

Il est extrêmement important d'arriver à bien informer l'utilisateur sur la source de son erreur, et essayer si possible de lui proposer une solution. Un langage où les erreurs sont mal commentées est voué à l'échec, car les utilisateurs se découragent et se lassent assez vite, si ils restent bloqués sur un simple oubli d'un symbole, ou sur une simple faute de frappe. Par exemple, dans Le Lisp de l'INRIA (1984-1987), qui est un langage parenthésé qui devient rapidement illisible, l'oubli d'une parenthèse renvoie le message «read : erreur de syntaxe : EOF durant un READ » sans préciser la ligne ni la colonne (ni d'ailleurs qu'il manque une parenthèse). Il devient alors impossible de retrouver l'erreur lorsqu'on sait qu'en Lisp il n'est pas rare de comptabiliser une vingtaine de parenthèses sur une seule ligne.

L'autre grande difficulté est d'arriver à reprendre l'analyse syntaxique après une erreur. En effet si par exemple on a la grammaire suivante :

StructureSI ® **si** *condition* **alors** *expression*
| etc.

Et que l'utilisateur a saisi : « si (x < 2) x := 5 ; », il manque le mot clé «alors ». Le véritable problème réside dans le fait que l'analyseur syntaxique, qui attend le mot clé « alors », va renvoyer une erreur tant qu'il ne le trouvera pas. L'utilisateur va donc être assailli par un nombre d'erreurs égal au nombre de mots, qui suivent cette erreur. Le but est donc d'effectuer une récupération d'erreurs. Il existe également des méthodes différentes plus ou moins faciles à mettre en place, pour effectuer cette récupération :

- Mode panique : quand l'analyseur syntaxique rencontre une erreur, il élimine les symboles d'entrée un à un, jusqu'à retomber sur une unité lexicale dite de synchronisation, c'est-à-dire un caractère de fin de ligne (point virgule[11]) ou le

[11] : On évoque toujours le point virgule pour désigner le caractère fin de ligne, car c'est celui utilisé par la plupart des langages de programmations, mais cela n'a absolument rien d'absolu.

caractère de fin. Cette méthode est la plus sûre mais elle saute souvent beaucoup d'instructions qui ne sont alors pas reconnues.

- Récupération au niveau du syntagme : consiste à rajouter ou remplacer une entrée dans le code source, par exemple mettre un point virgule à la place d'une virgule, ou rajouter un point virgule oublié. Les inconvénients de cette méthode sont nombreux. En effet, il est possible de créer une boucle infinie en rajoutant le token au mauvais endroit, de plus il est impossible de gérer les cas dans lesquels l'erreur réelle s'est produite avant le point de détection.
- Production d'erreurs : si on a une idée précise des erreurs courantes qui peuvent être rencontrées, il est possible d'augmenter manuellement la grammaire du langage avec des productions qui engendrent les constructions erronées. C'est-à-dire de rajouter des règles de production qui reconnaissent les erreurs possibles. Cette méthode à l'énorme avantage de savoir à chaque fois dans quels contextes a été déclenché l'erreur, pour donner un message plus cibler à l'utilisateur.
- Enfin la correction globale : se réalise à l'aide d'un graphe qui présente des types de correction en fonction de l'erreur rencontrée, il faut ensuite choisir le plus court chemin pour aller d'une situation erronée vers une situation saine, avec le minimum de changement dans le code. Cette méthode est de loin la meilleure, mais elle est beaucoup trop longue à mettre en place, et n'a donc qu'un intérêt théorique.

Il est bien entendu judicieux de rechercher la solution la plus appropriée à ses besoins, mais on ne retiendra que celles qui sont vraiment intéressantes et possibles à mettre en place, à savoir le mode panique, et la production d'erreurs.

2.6.2 La table des symboles

La table des symboles est une liste contenant les mots réservés (pour éviter que l'utilisateur ne les redéfinisse), les constantes, les fonctions prédéfinies, et surtout les variables déclarées par l'utilisateur dans son code source. Elle conserve leur nom, leur adresse mémoire (et donc leur valeur), ainsi que leurs attributs comme par exemple la portée (dans quel bloc elles sont définies). Cette table est une liste dynamique pour permettre à l'utilisateur de saisir autant de variables qu'il désire.

La table des symboles interagit également, comme la gestion d'erreurs, dans toutes les phases de la compilation. C'est pourquoi il est difficile de la classer parmi une phase spécifique, mais il est vrai qu'elle est très sollicitée pendant la phase d'analyse syntaxique.

2.6.3 Les métacompilateurs

Evoqué depuis le début de ce mémoire, leur rôle, et leur définition peuvent encore vous paraître un peu flou. Les métacompilateurs sont des outils destinés à simplifier la mise en œuvre d'un compilateur ou d'un interpréteur. A savoir que pour créer un analyseur lexical, il faut implémenter entièrement les automates, et que pour réaliser un analyseur syntaxique il faut implémenter les grammaires, ce qui demande un temps extrêmement important. Les développeurs des premiers langages de programmation étaient obligés de passer par cette phase laborieuse de programmation, pour pouvoir juste commencer à développer leur langage. J. Ferber raconte : « le premier compilateur Fortran avait nécessité 18 hommes/années d'effort »[12]. C'est pourquoi on a cherché à automatiser certains traitements qui restent communs d'un compilateur à l'autre. C'est ainsi qu'on a mis au point les métacompilateurs, littéralement des «compilateurs de compilateurs », traduction de l'anglais compiler-compiler (ou comp-comp).

Depuis les premiers langages, entièrement implémentés manuellement, les développeurs ont donc axé leur recherche sur les métacompilateurs, mais le premier qui fut véritablement reconnu et qui aujourd'hui encore persiste, fut Lex et Yacc en 1975. Les métacompilateurs, permettent ainsi de rentrer directement des règles de définitions des analyseurs, ainsi que les actions que l'on veut appliquer pour chaque règle. Le métacompilateur génère ensuite des fichiers sources qu'il faut inclure au projet de compilation, et qui définissent l'implémentation de l'automate de reconnaissance lexicale, et de la grammaire définissant la syntaxe. Ces fichiers sont en C dans le cas de Lex et Yacc, mais il existe des métacompilateurs pour la plupart des langages (Pascal, Java, etc.). Il faut ensuite compiler ces fichiers générés avec le compilateur du langage approprié, pour obtenir son interpréteur ou son compilateur.

Le métacompilateur est vraiment l'outil qui a permis la démocratisation de la

[12] : J. FERBER [1997], « Cours de Compilation », Université de Montpellier : « Généralités » p 2.

compilation et son développement, en évitant de passer tant de temps sur l'implémentation des automates et des grammaires qui restent encore aujourd'hui des structures très lourdes à implémenter.

De plus, les métacompilateurs sont, en général, très optimisés, et prennent en compte la plupart des méthodes pour accélérer les traitements (analyse ascendante, analyseur prédictif,…). Ils permettent également, pour certains, de récupérer les tokens attendus lors du déclenchement d'une erreur, pour une meilleure information de l'utilisateur.

Par contre, ils présentent bien entendu des gros inconvénients, dont la cause provient de son mode d'implémentation. En effet, le métacompilateur *génère* le code des analyseurs, c'est-à-dire qu'il y a une phase non gérée par le développeur du compilateur, qui ne maîtrise, alors, pas toute la chaîne de production. Ce problème est inhérent à tout outil de développement qui utilise la génération de code : il manque de souplesse. Dès qu'un utilisateur va vouloir sortir un peu du cadre pour lequel est prévu le générateur, il va se butter à des impossibilités dues au limitation du langage. C'est le cas, par exemple pour des outils de développement assisté comme Windev ou même dans Visual C++, lorsqu'on veut toucher au fichier que l'outil a généré, le code devient vite instable, ou ne fonctionne plus du tout. Les métacompilateurs présentent également ces travers, lorsqu'il faut toucher à des paramètres avancés qui ne sont présents que dans les fichiers générés.

Mais dans l'ensemble, les métacompilateurs, essayent de gommer ces défauts, pour devenir plus souples ; et il est certain que le temps perdu à corriger ces légers inconvénients est totalement négligeable par rapport au gain de temps réalisé par l'utilisation de ces outils.

Voici donc les principaux concepts de la compilation, certes cela doit vous paraître très théorique et flou malgré que je m'efforce de vous en simplifier l'accès, mais tout deviendra certainement plus explicite lors des phases de réalisation, où vous verrez la théorie devenir pratique. Mais avant cela la dernière partie de ce premier chapitre dédiée à la compilation, vous éclairera, un peu plus, sur l'apport de l'utilisation de ce domaine.

III. DEBATS SUR LA COMPILATION

Dans cette dernière partie, nous allons parler de la compilation en générale, de ce qu'elle apporte, de ce qu'elle permet, ainsi que des divers domaines d'application qu'elle recouvre, mais aussi de ses inconvénients.

3.1 Apports et avantages

La compilation est un domaine « à part » de la programmation au même titre que l'intelligence artificielle car elle amorce carrément une méthode de pensées totalement différente. Par exemple, l'intelligence artificielle est très adaptée pour la programmation récursive, quasiment sans variable, et trouve son application pour l'implémentation de moteur d'inférence ou de réseaux neuronaux, domaines dans lesquels la programmation « classique » est incapable d'atteindre les mêmes résultats. Il en est de même pour la compilation, qui affiche des résultats surprenants pour certaines applications spécifiques par rapport à d'autres méthodes d'implémentation.

En effet, si on réalisait la réalisation d'un analyseur grammaticale, avec la logique de la programmation itérative, il faudrait analyser l'entrée et prévoir un cas pour chaque entrée différente avec des conditions. Et même si la compilation est un sous domaine de la programmation, au même titre que l'intelligence artificielle, elle utilise des notions propres, comme les automates et les grammaires qui amorcent véritablement une nouvelle façon de penser. Cette façon de penser lui permet d'être très performante dans certains domaines, mais reste, il est vrai, cantonnée à certaines applications, du fait de sa spécificité.

Mais même si les concepts qui la composent sont parfois d'une extrême complexité, la compilation conserve une grande facilité d'utilisation grâce aux métacompilateurs notamment.

3.2 Inconvénients de la méthode

La compilation souffre des inconvénients inhérents à sa condition, à savoir que c'est une méthode de conception, pour réaliser des langages de programmation. Elle appartient donc à la catégorie « programmation de bas niveau ». Heureusement, les métacompilateurs se sont adaptés et permettent de programmer avec des langages de haut niveau à la place de l'assembleur[13].

3.3 Domaine d'application

On l'a dit, la compilation reste restreinte à certains domaines et la création de langages de programmation est certainement l'activité qui l'utilise le plus, mais regroupe déjà beaucoup d'applications

- Les langages compilés sont plutôt réservés à la programmation pour les professionnels.
- Les langages interprétés ont également une application auprès des professionnels, mais leur implémentation peut s'orienter vers d'autres possibilités, comme la personnalisation de formules dans les progiciels (du fait de leur grand besoin de paramétrage), ou bien l'intégration d'un langage de macros dans les logiciels (la suite Office de Microsoft en est un bon exemple).

Mais il existe heureusement d'autres applications possibles dont voici une liste non exhaustive :

- Les interpréteurs peuvent également trouver des applications dans le support des formats de données, à savoir l'import de format de fichiers définis, plutôt que d'utiliser l'extraction partie par partie des différents champs, contenant les données, et qu'un simple retour chariot, ou espace mal placé peut perturber.
- La partie analytique de la compilation a prouvé de bon résultat dans les logiciels

[13] : J. Ferber présente une démonstration mathématiques qui montre la possibilité d'écrire le compilateur d'un langage dans le langage lui-même : J. FERBER [1997], ibid : « Généralités » p 2-3.

de reconnaissance vocale.

- On utilise aussi une partie des concepts de compilation dans les traducteurs et autres correcteurs orthographiques.

Bref, on se sert de la compilation dans tous les domaines utilisant la traduction d'un langage vers un autre, ou la reconnaissance d'un langage. Il est même envisageable (quelques projets existants) de traduire des programmes, écris dans un langage de haut niveau directement dans un autre[14].

3.4 Conclusion sur la compilation

Pour résumer cette première partie, la compilation est un domaine relativement compliqué, car elle présente un grand nombre de concepts très théoriques. Son utilité n'est certes plus à prouver, car sans cette méthode nos ordinateurs ne seraient pas ce que nous connaissons. Et même si elle date du début des ordinateurs modernes, elle n'est pas pour autant dépassée, l'intelligence artificielle en reprend même les principes pour implémenter la reconnaissance vocale. Nous allons maintenant aborder ensemble le passage de cette étude théorique vers la pratique, à savoir le travail que j'ai réalisé pendant mon stage dans la société SynOptis.

[14] : Sun finance actuellement un vaste programme de reconversion du système bancaire, et a lancé un appel d'offres pour toutes sociétés qui arriveras à faire un système capable de traduire les logiciels bancaires, fais en Cobol, vers Java

DEUXIEME PARTIE – ETUDE ET REALISATION

I. INTRODUCTION

Comme toute réalisation ambitieuse, ce projet nécessite, outre l'étude théorique du domaine que vous venez de lire, une bonne organisation de la réalisation, à savoir que j'ai découpé, le travail à effectuer en différentes phases :

- La phase d'étude préalable, qui regroupe l'étude du progiciel, et du module du formulateur déjà existant, ainsi que la recherche du métacompilateur pour réaliser le projet.
- La phase d'étude détaillée qui reprend le cahier des charges, et les différents objectifs du projet, ainsi que la phase d'étude avant réalisation concernant l'approche du projet.
- La phase de réalisation détaillant la partie technique du travail.
- Enfin la phase de résultats, qui, comme son nom l'indique affiche le bilan sur la fonctionnalité de l'outil réalisé.

C'est ces différentes phases que je vous invite à découvrir dans cette seconde partie.

II. ETUDE PREALABLE

L'étude préalable correspond au travail de préparation d'avant projet, pour ma part cette phase fut pour moi le moment de trouver le métacompilateur approprié pour la réalisation du projet, mais également d'étudier l'existant, puisque le progiciel SynOptis, embarquait déjà un formulateur.

2.1 Etude de l'existant

SynOptis est un progiciel, c'est-à-dire un logiciel qui tente d'être le plus possible paramétrable et souple pour répondre et correspondre à différentes professions (bureau d'études, SICTOM[15], éditeur et livreur de journaux, …), et différents domaines d'applications (collecte des déchets, tri sélectif, livraison de journaux,…). Cette souplesse obligatoire pour ce type d'application, doit s'appliquer dans tout le progiciel. De plus, SynOptis est également un outil d'aide à la décision, c'est-à-dire qu'il fournit des statistiques et des prévisions sur le travail des utilisateurs. Le problème qui s'est posé dès le début de la conception du logiciel, était de savoir comment faire pour rendre les formules de prévisions paramétrables. Bien entendu, il était tout à fait possible de proposer dans une fenêtre un certain nombre de formules différentes, mais l'outil aurait été ainsi totalement figé, et les utilisateurs auraient du attendre la version suivante, pour pouvoir utiliser la formule qu'ils avaient demandé de rajouter.

Ceci était totalement inacceptable au niveau de la conception, car d'une part le progiciel n'aurais pas rempli son office d'application entièrement paramétrable, en limitant le degré de liberté de l'utilisateur ; et d'autre part avec la multitude de champs qu'il existe dans la base de données, il aurait été impossible de saisir toutes les formules désirées par les clients, pour tous les champs.

[15] : Syndicat Intercommunal chargé de la Collecte et le Traitement des Ordures Ménagères, actuellement certain syndicat change d'appellation et préfère SIVOM (Syndicat Intercommunal Voué aux Ordures Ménagères)

C'est pourquoi l'applicatif SynOptis s'était doté dès sa deuxième version d'un formulateur. Cet outil a pour objectif de rentrer pour les champs que l'on désire calculer, par exemple estimer le poids d'ordures à ramasser sur une tournée de collecte, une ou plusieurs formules selon ce que l'on veut obtenir. Par exemple ici, on peut estimer le poids à ramasser en tonne/heure, en tonne/km,… .

Mais le formulateur existant, développé au début du projet SynOptis, souffre de problèmes importants qui ont poussés au besoin de refonte du module. Voici un aperçu du formulateur actuel, sur lequel vous pouvez déjà constater certains problèmes de clarté évidentes :

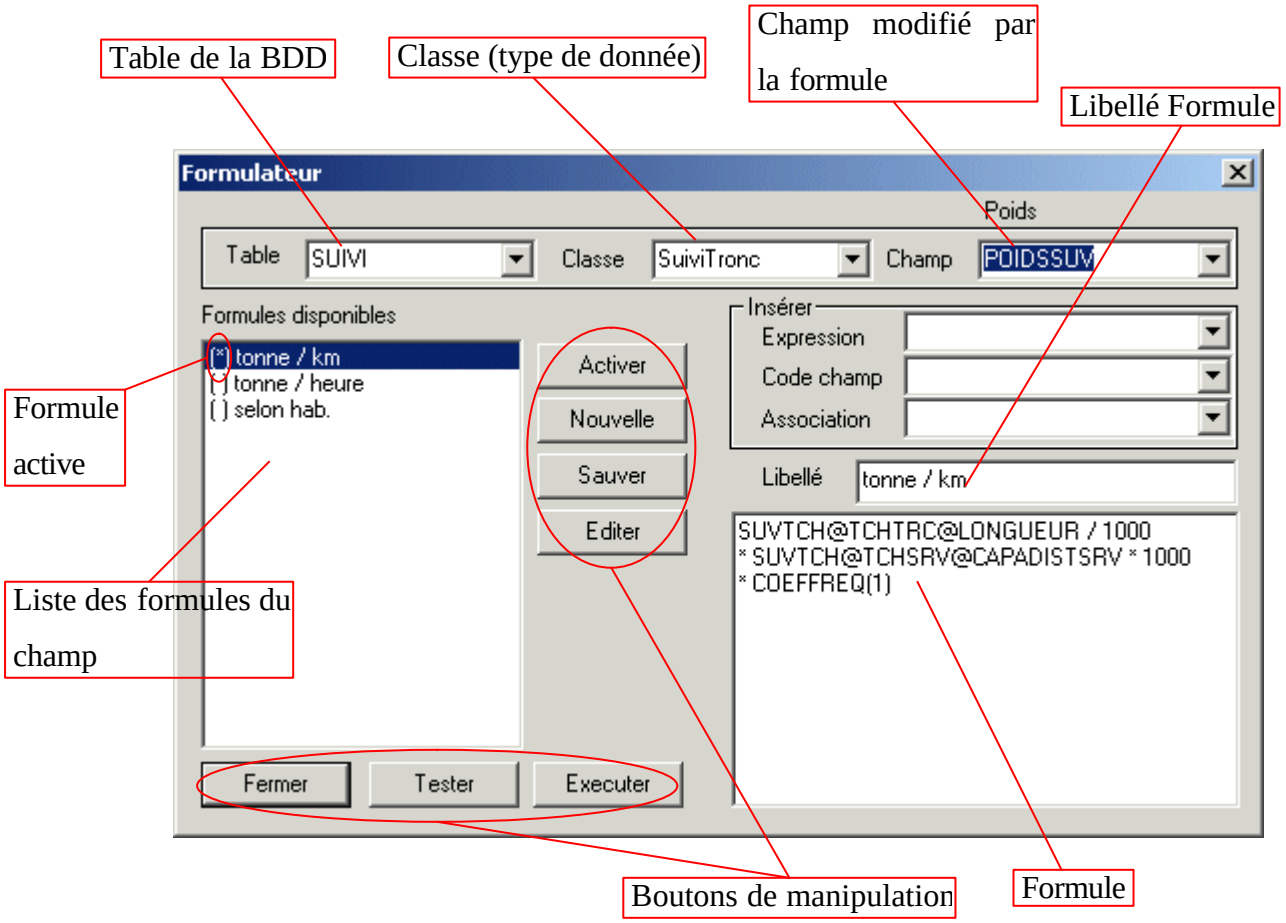


Figure 16 : Aperçu du formulateur existant

En faisant l'étude de ce module, voici les différents problèmes que j'ai pu y recenser :

- Tout d'abord, comme vous pouvez le voir, ce module manque totalement d'ergonomie, ce qui le rend difficile à aborder et à comprendre.

- Le code source de la grammaire originelle du langage du formulateur a été perdu, et il est donc d'autant moins facile à modifier, vu que les programmeurs sont obligés de modifier le code intermédiaire généré par l'outil de compilation.
- Certaines fonctionnalités ne sont pas implémentées, il est par exemple impossible de supprimer une formule, l'imbrication de plusieurs structures de contrôle (si, case, …), n'était pas gérée.
- Il est impossible de manipuler des valeurs de type chaîne, uniquement des numériques.
- Les formules sont limitées à 255 caractères.
- Il n'est pas possible de choisir si un champ ou une table sera visible dans le formulateur, d'où la possibilité pour l'utilisateur de modifier la valeur de champs réservés à la gestion du logiciel.
- Le code des formules est, comme vous pouvez le voir, illisible, et inaccessible à l'utilisateur, car il faut aller chercher les valeurs des champs dans la base de données, par des noms très peu explicites, et par des relations qui sont des notions totalement floues pour des utilisateurs parfois rebutés par l'informatique.
- L'interpréteur actuel ne lit les formules que par le biais d'un passage par un fichier sur le disque, ce qui ralentit considérablement les performances.
- Le progiciel SynOptis est entièrement programmé en C++, à savoir, en langage orienté objet, alors que le formulateur est lui en C et donc non orienté objet.

Il est évident que la correction de ces problèmes, doivent venir s'ajouter aux objectifs de réalisation du nouveau formulateur.

Après analyse des besoins de refonte de l'interpréteur, et de recherche des évolutions nécessaires pour arriver à remplir le cahier des charges (cf. III Etude détaillée), je me suis de suite aperçu qu'il m'est impossible de me resservir de ce qui a était fait. En effet, l'interface graphique ne me convenant pas, et le code source de la grammaire étant perdu, il ne me reste plus rien d'exploitable. C'est pourquoi j'ai décidé de recommencer le développement depuis le début, tout en gardant, bien sûr, la ligne directrice de l'ancien formulateur.

2.2 Test des métacompilateurs

En premier lieu, j'ai testé les différents produits existants qui permettent d'implémenter l'interpréteur, en les évaluant sur plusieurs critères : les performances, le prix (licence d'utilisation), mais surtout la compatibilité du code généré avec Visual C++ 6.0 pour l'intégrer au progiciel SynOptis développé avec ce dernier. J'ai ainsi passé au banc d'essai tous les métacompilateurs que j'ai pu trouver sur Internet, avec une grammaire de test simple, en l'occurrence la grammaire définissant les opérations arithmétiques (cf. Figure 4), pour implémenter une calculatrice ; et voici le tableau récapitulatif de ces tests :

Métacompilateur	Performances	Prix	Licence	Intégration
Gold Parser	Ne marche pas Ne fais que l'analyse syntaxique	Payant	Shareware	Ne marche pas
Parser Generator	Correcte	50 $	Libre une fois payé	Ok avec Visual C++
CUP	Rapide	Gratuit	Libre	Uniquement pour JAVA
ANTLR	Correcte	Gratuit	GNU	Uniquement pour Java
Visual Parse ++ 5	Ne marche pas	495 $	Libre une fois payé	Ne marche pas
Berkeley Yacc	Rapide	Gratuit	GNU	OK mais laborieuse (pas de code C++)
Anagram	Ne marche pas	Payant	Libre une fois payé	Ne marche pas
Cygwin	Rapide	Gratuit	GNU	OK mais pas très intégré dans Visual
Bison Flex Wizard	**Rapide**	**Gratuit**	**GNU**	**OK très bonne intégration dans Visual C++**

Finalement mon choix s'est orienté vers Bison Flex Wizard, qui est tout simplement l'équivalent des outils Flex et Bison de Linux, compilé sous Windows. Les développeurs de Bison Flex Wizard ayant rajoutés une option dans les menus de Visual C++ pour créer un projet de compilation, l'intégration n'en est que plus aisée et est la meilleure parmi tous les logiciels testés. En effet, on utilise l'éditeur pour taper ses fichiers Flex et Bison, que on

compile directement dans l'interface de Visual C++.

Flex et Bison sont les outils libres de Linux, clones de Lex et Yacc d'Unix. Comme ils ont été développés sous licence GNU (GPL), ils sont gratuits et redistribuables, et le code source est fourni. De plus, ils sont toujours considérés comme étant parmi les métacompilateurs les plus puissants, et offrent une grande souplesse de développement.

Flex

Flex est un générateur d'analyseur lexical dont la syntaxe est exactement la même que celle de Lex présentée dans l'Annexe 2. Un fichier flex (ou lex) se décompose en 3 parties (cf. Figure 17 : sur le schéma les mots clés sont présentés en vert, et les parties en bleu sont de légers exemples). Flex ne présente rien de très spécifique à connaître, il faut juste bien avoir assimilé les opérations ensemblistes présentées dans l'Annexe 2, pour définir les tokens désirés. Néanmoins précisons que c'est le fichier Flex qui doit contenir la fonction de lecture du flux d'entrée YYINPUT.

Bison

Bison est un générateur d'analyseur syntaxique, structuré en trois grandes parties également, présentées sur le schéma suivant avec le même code couleur que précédemment. Bison est, quant à lui beaucoup plus spécifique dans sa syntaxe. Premièrement par rapport à la syntaxe BNF, que l'on a déjà vu plus haut, Bison remplace la flèche qui suit la première règle d'un non terminal par « : », et à la fin des règles d'un non terminal, on rajoute un « ; » qui n'est pas obligatoire mais qui permet une meilleure lisibilité entre les règles.

Ensuite dans la première sous partie de la définition de la syntaxe, il faut définir avec le mot réservé « %union », les types de données qui seront manipulées dans l'analyseur, pour chaque non terminal ou autres éléments des règles de production. Il est ainsi possible de manipuler plusieurs types de données différents, puisqu'il s'agit de l'équivalent d'une union en C. Le mot clé « %token » définit les unités lexicales que renvoie l'analyseur fait avec Flex, qui peut être éventuellement suivi du type du token, déclaré dans « %union », que l'on note, « <type_voulu> ».

« %left », « %right », et « %nonassoc » servent à définir la priorité des opérateurs unaires et binaires (« %left » indique que l'on commence par évaluer la partie de gauche de

l'expression, par exemple pour un opérateur arithmétique, «%right » que l'on commence par la partie droite, pour l'opérateur d'affectation par exemple). Finalement le mot clé «%type » définit le type des non terminaux, et «%start » précise l'axiome de la grammaire.

Flex	*Bison*
1. Partie définition et inclusion de librairies %{ #include <stdio.h> … %} **2. Partie formation des tokens** Définitions des types de tokens réutilisables blancs [\t\n]+ chiffre [0-9] … %% Définitions des tokens et valeurs de renvoie {blancs} ; {chiffre} return (NUM) ; "(" return (PARO) ; ")" return (PARF) ; "+" return (PLUS) ; … %% **3. Partie Fonction utilisateur** **int foncExemple() {** **…**	**1. Partie définition et inclusion de librairies** %{ extern int yylex() ; … %} **2. Partie définition de la syntaxe** Définitions des terminaux et non terminaux utilisés %union {int type_entier ;} %token <type_entier> NUM %token PARO PARF %left PLUS %type <type_entier> axiome T %start axiome … %% Définitions des règles de production axiome : axiome PLUS T {$$ = $1 + $3 ;} \| T {$$ = $1 ;} ; T : PARO axiome PARF {$$ = $2 ;} \| NUM {$$ = $1 ;} ; … %% **3. Partie Fonction utilisateur** **int yyerror() {**

Figure 17 : Structure de fichiers Flex et Bison

Il existe beaucoup d'autres mots clés pour affiner la grammaire, mais le schéma ci-dessus vous présente les plus utilisés.

Les seuls éléments qui doivent encore vous intriguer, sont les « $$ », et autres « $n ». « $$ » représente la valeur du non terminal à gauche de la flèche. « *Les valeurs associées aux symboles apparaissant dans la partie droite de la règle […] sont dénotées par « $n » où « n » est la position du symbole dans la partie droite. La numérotation commence à 1 et tous les éléments sont pris en compte (c'est-à-dire même les actions et les littéraux).* »[16]. Voici un schéma pour expliciter cette définition :

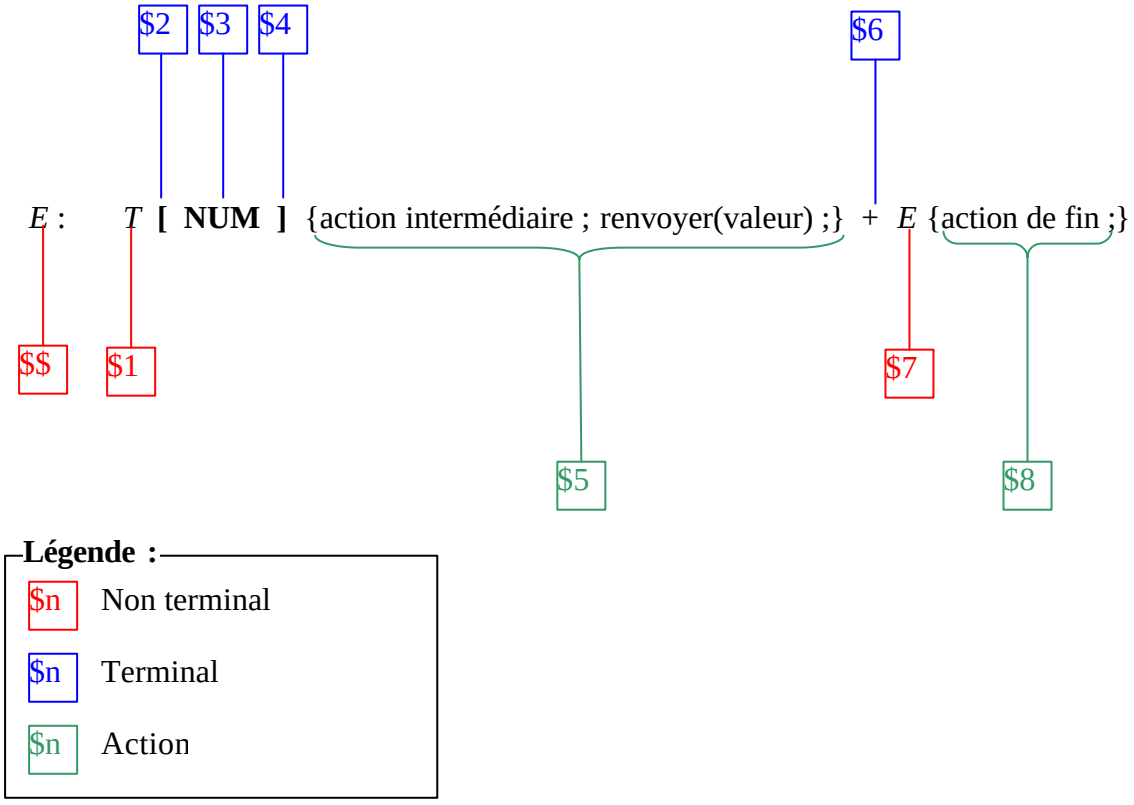


Figure 18 : Manipulation des valeurs des règles

Après avoir cerné les problèmes de l'existant, et trouvé un métacompilateur pour réaliser le projet, passons maintenant à la phase d'étude avant réalisation.

[16] : J. BONNEVILLE [1999], « Cours de techniques de compilation – Documentation Yacc – Bison », Université de Lyon : 2.2.2 Les règles – Les Variables Yacc/Bison

III. ETUDE DETAILLEE

Cette phase présente l'étude préalable à la réalisation, à savoir qu'elle regroupe l'établissement et l'étude du cahier des charges, ainsi que la réalisation sur papier, de la future application.

3.1 Objectif

Avant d'entamer la programmation du module, il a fallu définir avec Fabien Rodes, le cahier des charges, c'est-à-dire récapituler les fonctionnalités techniques du futur formulateur, dont voici les principales composantes demandées par le dirigeant, ou issues de l'étude préalable que vous venez de lire :

Refonte du formulateur :

- Réaliser un langage de programmation performant et facile d'accès pour les néophytes.
- Le formulateur ne doit pas être perturbé par l'utilisation d'espace, ou de retours chariot.
- Le langage doit savoir manipuler des valeurs numériques, ou des chaînes de caractères.
- Le langage doit gérer les opérations arithmétiques, ainsi que les principales fonctions mathématiques (cos, sin, …).
- Il doit être également possible d'utiliser les opérateurs relationnels (opérateur de comparaison : « < », « > », …), ainsi que les opérateurs booléens (« et », « ou » et « non »).
- Le langage du formulateur doit permettre la saisie de variables, soit directement par une affectation, soit lorsqu'une variable est utilisée sans être initialisée, le formulateur doit demander sa valeur.
- Les structures de contrôles « If » et «Case » doivent être implémentées. Il doit être également possible d'imbriquer plusieurs structures de contrôles.

- Le formulateur doit être capable de lire un code source directement depuis la mémoire sans passer par un fichier sur le disque dur.
- Le module doit, au maximum être implémenté en programmation orienté objet.
- Le langage de programmation doit être prévu pour manipuler les champs de la Base De Données, pour lire une valeur ou pour la modifier.
- Enfin le formulateur doit posséder un meilleur code de gestion des erreurs.

Refonte de l'interface graphique :

- La nouvelle interface doit être plus claire, et plus fonctionnelle que le précédent système, qui était très hermétique et inabordable.
- Le nouvel éditeur doit permettre de saisir des formules de plus de 255 caractères.
- Il faut également trouver un système pour limiter le nombre de tables et de champs accessibles dans le formulateur, pour éviter de modifier des champs réservés.

Dans l'ensemble, le dirigeant, m'a laissé les mains libres, pour la conception, tant que l'interface ou le formulateur répondent aux exigences de simplifications.

3.2 Conception sur « papier »

Je nomme ici la conception sur papier, les différentes études de conception que j'ai fait mentalement ou sur papier, avant de commencer la programmation.

3.2.1 Le formulateur

Le langage

C'est ici que je présenterais, non pas les algorithmes, puisque mon travail n'est pas de concevoir des traitements compliqués, mais la grammaire BNF. Voici donc la grammaire que j'ai élaboré, qui sera rassurez vous largement expliqué e plus bas.

axiome	:	/*VIDE*/ {}
	|	*axiome bloc* {}
bloc	:	**DEBUT** *liste_instr* **FIN** {}
	|	*instruction* {}
	|	**error** {}
liste_instr	:	*instruction liste_instr* {}
	|	/*VIDE*/ {}
instruction	:	*condition* **FINL** {}
	|	*declaration* **FINL** {}
	|	*structSI* {}
	|	*structCAS* {}
structSI	:	**SI** *condition* {} **ALORS** *blocSI* {} *blocSINON* {} **FINSI** {}
blocSI	:	liste_instr {}
blocSINON	:	**SINON** *blocSI* {}
	|	/*VIDE*/ {}
structCAS	:	**SELON** *condition* {} *blocCAS blocDEFAUT* **FINCAS** {}
blocCAS	:	**CAS** *intervalle* **DEUXPOINT** *blocSI* {} *blocCAS* {}
	|	/*VIDE*/ {}
blocDEFAUT	:	**DEFAUT DEUXPOINT** *blocSI* {}
	|	/*VIDE*/ {}
declaration	:	**VAR RECOIT** *condition* {}
condition	:	*condition op_comp conditionOU* {}
	|	*conditionOU* {}
op_comp	:	**EQ** {}
	|	**NE** {}
	|	**GT** {}
	|	**LT** {}
	|	**GE** {}
	|	**LE** {}
conditionOU	:	*conditionOU* **OU** *conditionET* {}
	|	*conditionET* {}
conditionET	:	*conditionET* **ET** *conditionNON* {}
	|	*conditionNON* {}

```
conditionNON :   NON expressionADD {}
             |   expressionADD {}
expressionADD :  expressionADD PLUS expressionMUL {}
             |   expressionADD MOINS expressionMUL {}
             |   expressionMUL {}
expressionMUL :  expressionMUL FOIS expressionFCT {}
             |   expressionMUL DIVISE expressionFCT {}
             |   expressionFCT {}
expressionFCT :  FCNT valeur {}
             |   expressionFCT PUISS valeur {}
             |   valeur {}
intervalle   :   crochet valeur VIRGULE valeur crochet {}
             |   op_comp valeur {}
             |   valeur {}
crochet      :   CROO {}
             |   CROF {}
valeur       :   PARO condition PARF {}
             |   NUM {}
             |   CHAINE {}
             |   MOINS valeur %prec NEG {}
             |   PLUS valeur %prec POS {}
             |   VAR {}
             |   BDD relationBDD {}
relationBDD  :   FLECHE BDD relationBDD {}
             |   /*VIDE*/ {}
```

Figure 19 : Grammaire du langage du Formulateur

Pour mieux comprendre le rôle de toutes ces règles, voici une légère explication de ce qu'elles reconnaissent, et des actions qui leur sont associées :

axiome : l'axiome est le non terminal de départ de la grammaire, il a donc un rôle très important puisque c'est lui qui définit si l'interpréteur sera réentrant ou non, et c'est lui qui

s'occupe de l'écriture des résultats à l'écran ou dans la BDD,.

bloc : définit la notion de bloc d'instructions, c'est-à-dire soit plusieurs instructions, délimitées par un token de début et un de fin (ici DEBUT et FIN), soit une instruction seule, soit encore une erreur, c'est-à-dire que toutes erreurs de syntaxe seront récupérées dans ce cas, c'est ici qu'est géré une partie de la récupération sur erreur.

liste-instr : ce non terminal n'apporte rien de particulier, si ce n'est de définir qu'une liste d'instructions se compose d'un nombre illimité d'instructions juxtaposées.

instruction : une instruction se compose, soit d'une structure de contrôle (Si, Selon) soit d'une ligne d'expression se terminant par un caractère de fin de ligne FINL qui est dans le formulateur un point virgule.

structSI, *blocSI*, *blocSINON* : gèrent la structure de contrôle conditionnelle « Si », avec bien entendu un « Sinon » optionnel. Comme on l'a vu tout à l'heure, le « Si » est délicat à gérer dans la grammaire, car c'est une source d'ambiguïté. Pour ma part, je me suis mis à la place d'un utilisateur néophyte, et je me suis rendu compte que l'approche des langages de haut niveau, tel le C, le Pascal, ou autre, du « Si » ne semblerait pas logique, pour des gens parfois rebutés par l'informatique. En effet dans ces langages, les balises de «début » et de « fin », ne sont pas obligatoires s'il n'y a qu'une seule instruction dans les blocs ; par contre elles deviennent indispensables, dans le cas où il y en aurait plusieurs. Plutôt que d'imposer cette façon de penser compliquée et source d'erreurs chez les néophytes, j'ai préféré m'inspirer des langages de scripts (comme sous Linux par exemple), qui utilisent une balise de fin de la structure conditionnelle. C'est pourquoi j'utilise le «FINSI » dans ma structure qui offre l'énorme avantage, de supprimer l'ambiguïté du « sinon en suspens » vu plus haut.

structCAS, *blocCAS*, *blocDEFAUT*, *intervalle*, et *crochet* : sont les règles utilisées pour définir le « Selon » (Case ou switch dans les autres langages). Là encore je me suis mis à la place d'un utilisateur non informaticien, pour trouver la syntaxe qui serait la plus facile à assimiler pour lui. Car si on traduit les structures «case » des autres langages en français, on s'aperçoit vite que cela ne voudrait pas dire grand-chose pour quelqu'un qui ne sait pas programmer. En effet le « case » du pascal, signifierait en fançais : « **cas** <variable> **de** <valeur1> :… ». Vous conviendrez que ce n'est pas très explicite. C'est pourquoi j'ai mis en place un « case » qui ressemble vaguement à celui du C, mais qui se veut beaucoup plus

performant au niveau du tri, comme le montre l'exemple ci dessous :

```
Debut
    variable := 5 ;                 On initialise une variable
    Selon variable                  Selon la valeur de cette variable
        Cas 1 : …                   Cas variable = 1 (Faux)
        Cas [2, 5[ :…               Cas variable appartient à l'intervalle [2, 5[ (Faux)
        Cas ]4, 5] :…               Cas variable appartient à l'intervalle ]4, 5] (Vrai)
        Cas > 4 :…                  Cas variable > 4 (Vrai)
        Cas <> 6 :…                 Cas Variable <> 6 (Vrai)
        …
        Defaut :…                   Cas par défaut (Faux)
    Fincas
Fin
```

Figure 20 : Exemple des fonctionnalités du selon

Cet exemple vous montre une partie des cas qu'il est possible de faire avec le « selon » du formulateur. On peut donc rentrer soit une seule valeur, pour spécifier l'égalité, soit un opérateur de comparaison et une valeur, soit faire un intervalle mathématiques avec des inclusions et des exclusions (« [» , «] »).

declaration : définit la déclaration de variable, et initialise sa valeur. Le symbole d'affectation retenu pour plus de clarté est le « := », le simple « = » est utilisé comme opérateur de comparaison. En effet, l'utilisation du «== » comme en C ne m'a pas semblé très judicieux.

condition, *op_comp*, *conditionOU*, *conditionET*, *conditionNON* : gèrent les conditions, les opérateurs de comparaisons et les opérateurs booléens. L'enchaînement des règles en cascade, comme elles sont définies, permet de gérer la priorité des opérateurs. A savoir que plus une règle est basse (dans l'enchaînement) plus elle est réduite tôt, et plus elle est prioritaire.

expressionADD, *expressionMUL*, *expressionFCT* : sont les non terminaux qui définissent les opérateurs mathématiques, ainsi que les fonctions mathématiques (cos, sin,…)

ou autre. Ici également la priorité est gérée par l'enchaînement des non terminaux. On retrouve la grammaire « ETF » de la figure 4, car c'est à partir de cette simple grammaire que j'ai réalisé le langage du formulateur, en l'étoffant de plus en plus de règles.

valeur : c'est le non terminal le plus prioritaire, car tout en bas de l'arbre d'analyse. C'est lui qui définit les terminaux, et renvoie leur valeur pour la manipulation dans l'interpréteur. La première règle rend les expressions parenthèsables ; les deux règles suivantes définissent l'entrée d'une simple valeur, numérique ou alphanumérique. La quatrième et la cinquième règle définissent la possibilité de précéder une valeur d'un signe « + » ou « − », pour gérer, par exemple, la formule suivante : « 2 + −3 », sans obliger l'utilisateur à rajouter des parenthèses. Enfin la règle « *valeur* → **VAR** » reconnaît les variables, et si elles ne sont pas initialisées, l'action associée demande sa valeur et la lit sur le flux d'entrée choisi.

Enfin *relationBDD* : est le non terminal qui permet au formulateur, couplé à une procédure externe, de lire et modifier la base de données. Ainsi pour accéder aux champs d'une table, on utilise la syntaxe :

« TABLE_SOURCE→CLE_MIGRE→TABLE_LIE→CHAMP_VOULU »

L'implémentation

Une fois la syntaxe du langage définie, je me suis fixé moi-même des contraintes quant à l'implémentation du formulateur :

- Premièrement le formulateur doit être, d'après le cahier des charges, le plus possible implémenté en objet.
- Ensuite je tiens à ce que l'interpréteur reste un module indépendant de l'interface graphique, et donc tous les traitements devront être indépendant. A savoir que le formulateur, devra permettre d'interpréter un code source depuis un fichier, le clavier, ou depuis la mémoire, et devra rester fonctionnel dans toutes ces utilisations. Cela me permettra de pouvoir plus facilement débuguer l'application en évitant de passer par le progiciel et l'interface graphique.
- Le formulateur devra être également capable d'écrire sur n'importe quelle sortie : écran, fichier, mémoire.

Avec ces différentes implémentations, l'interpréteur restera indépendant et réutilisable, pour plusieurs applications dans le logiciel.

3.2.2 L'interface graphique

Pour remplacer l'interface graphique existante, il semblait primordial de changer le système de sélection des formules et de navigation dans la base de données, qui était très peu explicite dans le formulateur actuel. Il fallait également plus d'espace d'écriture, et une interface plus claire et conviviale, et surtout plus proche des repères des utilisateurs. En effet, puisque l'interface contient un éditeur de texte, il faut que cet éditeur ressemble aux références des néophytes, à savoir Word ou Works. C'est pourquoi j'ai pensé à une interface composée de deux fenêtres au lieu d'une seule.

La première fenêtre servirait aux manipulations sur les formules, elle contiendrait un tree-view, qui présenterait les tables modifiables, avec pour fils, les classes de types de données, reliées à cette table, puis les champs de la table. Ses champs seraient également les pères des formules qu'ils contiennent.

Pour manipuler ces champs et ces formules, des boutons seraient disponibles, « ajout/suppression » de formule, « activer » (en effet dans la plupart des cas, il existe plusieurs formules par champs, mais une seul doit être activée pour les calculs), « exécuter » qui lance le calcul de la formule, et « éditer » qui lance la seconde fenêtre contenant l'éditeur.

La seconde fenêtre serait composée du même tree-view que précédemment, légèrement modifié. En effet, il ne présenterait que la table du champ dont on modifie la formule, et ses champs. Par contre, il afficherait les clés de façon différenciée (avec des petits icônes par exemple), pour permettre à l'utilisateur de double-cliquer dessus et d'accéder ainsi à la table reliée et à ses champs. Libre à l'utilisateur d'accéder à une autre table, etc. Pour simplifier cette navigation dans la base de données, l'utilisateur aurait à sa disposition des bulles d'aides, dans le tree-view (également dans celui de la fenêtre de sélection), lui donnant le détail des champs ou tables, pour mieux expliciter les noms des éléments de la base de données.

En plus de l'arbre de navigation, l'éditeur serait composé d'une barre de menu reprenant les fonctions courantes des traitement de texte : copier/coller, annuler, rétablir,

sauvegarder, plus un bouton « test » pour tester la formule, et d'une barre d'état (en bas de la fenêtre) pour afficher les informations sur le boutons, ainsi que des renseignements tel que l'état du pavé numérique, du bouton de majuscule, et du bouton d'insertion.

La majeure partie de la fenêtre serait occupée par la zone d'édition, et la partie inférieure par la zone de retour, qui afficherait les erreurs et les résultats. Enfin un champ permettrait la saisie du libellé de la formule.

Une fois le langage du formulateur, la méthode d'implémentation, et les lignes directrices de l'interface graphique déterminés, nous allons maintenant aborder la phase d'implémentation, à proprement parlé, c'est-à-dire les différents développements, et tâches que j'ai effectué durant mon stage.

IV. REALISATION

La phase de réalisation est certainement la plus longue à mettre en place. C'est pourquoi je l'ai découpé en trois sous parties : la phase de développement de l'interpréteur, le développement de l'interface graphique et l'intégration dans le progiciel, puis la phase de déboguage. Une quatrième sous partie sera consacrée aux travaux connexes réalisés en dehors de mon projet.

4.1 L'interpréteur

L'interpréteur que j'ai réalisé, a été implémenté avec Flex et Bison donc, sous Visual C++. La grammaire utilisée est celle présentée Figure 19. Une fois la grammaire saisie, j'ai du implémenter plusieurs classes et fonctions, pour rendre l'interpréteur fonctionnel :

- Tout d'abord, j'ai du créer plusieurs classes représentant les différents types de données manipulables par le formulateur (valeur, symbole, intervalle, opérateur,…)
- J'ai également implémenté la classe «FonctionForm» pour pouvoir permettre de rajouter des fonctions de bases au formulateur, sans retoucher au code.
- La classe « VariableES » est utilisée par l'interpréteur pour récupérer le code source de n'importe quel type de flux d'entrée, et renvoyer les résultats et les erreurs sur un flux de sortie quelconque.
- La classe « Contexte » est indispensable pour la gestion des structures de contrôles conditionnelles.
- Pour le bon fonctionnement de l'interpréteur, il est bien entendu indispensable d'implémenter les actions rattachées à chaque règle de l'analyseur lexical, et de l'analyseur syntaxique, ainsi que les fonctions annexes (yyerror par exemple).
- Enfin la classe « FormulateurParser » définit le formulateur lui-même.

Nous allons maintenant détailler un peu plus l'implémentation de ces classes, et fonctions.

La classe Valeur

Le formulateur doit, on l'a dit, manipuler indifféremment des valeurs numériques, et des chaînes de caractères. Or, il n'est pas possible en C, ou même dans un autre langage, de manipuler ces deux types de la même façon. C'est pourquoi, il fallait trouver une solution pour éviter d'avoir tous les traitements de la grammaire en double, d'où l'implémentation de la classe « Valeur ». En effet, cette dernière est composée d'une union contenant une chaîne de caractères (CString), d'un réel (double), et d'un pointeur sur fonction, pour pouvoir recevoir tout type de données ; à cela s'ajoute un entier qui définit son type.

Cette classe est implémentée avec un constructeur, une fonction pour récupérer la valeur (get…), et une pour la modifier (set...), pour chaque type de données. Une fonction nommée « convert », convertit n'importe quel valeur et renvoie un pointeur sur un objet « Valeur » du type demandé (CHAINE ou NUM).

Enfin, tous les opérateurs de comparaisons et l'opérateur d'affectation de la classe « Valeur », sont surchargés, pour faciliter l'utilisation dans le formulateur, et ainsi éviter de nombreux traitements, participant ainsi à la simplification de l'outil pour l'utilisateur qui n'a pas à se soucier des conversions de types.

La classe Symbole

La classe «Symbole » répond aux besoins de l'interpréteur de pouvoir manipuler des variables, et des fonctions. En effet, le formulateur se trouverait terriblement amputé de sa puissance si on ne pouvait stocker des valeurs dans des variables. C'est pourquoi, j'ai implémenté la classe «Symbole », qui est dérivée de la classe « Valeur », c'est-à-dire que c'est une valeur nommée, ayant un type supplémentaire qui peut être, VAR pour une variable, FCNT pour une fonction, ou BDD pour un champ de la base de données. Les attributs de « Symbole » sont, bien entendu, accompagnés des fonctions pour les lire et les modifier, ainsi que des constructeurs pour les initialiser. La classe « Symbole » possède également les mêmes opérateurs surchargés que sa classe mère, mais dispose, en plus d'une fonction pour exécuter, une fonction stockée comme symbole (sin, cos,…).

La structure Intervalle

Elle est composée de deux valeurs une minimum et une maximum, avec deux entiers

qui précisent leur état (inclus ou exclu).

Le type Opérateur

Est utilisé pour éviter de répéter les traitements des règles de comparaisons, « < », « > », «>= », «<= », «= », «<> », il s'agit en fait d'un simple entier qui renvoie le numéro de l'unité lexicale.

La classe FonctionForm

Cette classe, n'a que peut d'intérêt car elle permet, en fait, de rajouter des fonctions de bases aux langages (c'est ainsi que sont entrées cos, sin, tan, etc.), évitant ainsi de devoir modifier le code, et accroissant, un peu plus la souplesse et la réutilisabilité de l'interpréteur. La classe «FonctionForm » est implémentée avec le minimum de fonctions, et d'opérateurs nécessaires.

La classe VariableES

Cette classe peut paraître futile, dans l'optique de la future utilisation du formulateur dans le progiciel SynOptis, car elle permet d'utiliser différents flux d'entrée/sortie (clavier, écran, fichiers, variables mémoire,…) et que le logiciel n'en utilisera qu'une seule, à savoir le passage par variables. Pour ma part, je la considère d'une bien plus grande utilité. En effet, c'est elle qui assure la liaison entre l'interface graphique et l'interpréteur, en implémentant des fonctions pour avancer dans le source (quel qu'il soit), pour écrire sur le flux de sortie, etc. Cette classe est sans nul doute, une des classes maîtresse de l'intégration de l'interpréteur dans l'interface graphique, et une autre grande preuve de souplesse du formulateur. Pour moi, elle avait encore une autre utilité, car elle m'a permis de développer et de débuguer le module plus facilement en l'utilisant en mode console, c'est-à-dire en saisissant les formules au clavier, ou par fichiers, et en ayant les résultats à l'écran, sans avoir à passer par l'interface graphique, qui nécessitait beaucoup plus d'opérations.

La classe Contexte

La classe «Contexte », fut une des dernières développées, car elle répond aux besoins des structures conditionnelles qui furent implémentées vers la fin du module. Elle contient deux listes de «Symboles » qui seront utilisées lors de la sauvegarde dite de contexte, qui

stocke les valeurs des variables de la table des symboles, et les valeurs des non terminaux en cours d'évaluation (voir « Les actions associées aux règles »).

La classe FormulateurParser

La classe «FormulateurParser » est la classe de base de l'interpréteur, même si son nom ne l'indique pas[17]. Elle possède comme attributs :

- Un pointeur sur une fonction de lecture : cette fonction est utilisée lors de la reconnaissance d'une variable non initialisée, pour lui affecter une valeur. Si ce pointeur est nul, alors on utilise l'entrée standard (clavier). Ce pointeur de fonction permet de garder l'indépendance entre l'interpréteur et l'interface graphique.
- Un pointeur sur une fonction de lecture dans la base de données : idem que précédemment, sauf qu'ici la fonction retourne un entier pour dire si le token lu fait partie de la base de données (champ ou table), ou s'il s'agit d'une relation valide (table→champ ou champ→table) dans la base de données, et renvoie la valeur du champ, dans un pointeur passé en paramètres.
- Un entier qui contient le numéro de l'erreur détectée dans la syntaxe.
- Deux entiers qui stockent la valeur de la ligne et la colonne où est apparue une erreur.
- Deux variables de type « VariableES », une d'entrée, et une de sortie.
- Une liste de « Symbole » qui contiendra la table des symboles.
- Une pile de contexte, pour gérer les imbrications de structure de contrôles et par conséquent de contexte mémoire.

Outre ces attributs, elles disposent de plusieurs méthodes qu'il n'est pas nécessaire de toutes détailler :

- Un constructeur très paramétrable et un destructeur.
- Plusieurs fonctions pour changer les fonctions de lecture (simple ou dans la base de données).

[17] : La classe « Formulateur » et la classe « Parser », existent déjà dans le progiciel SynOptis. En effet, elles étaient utilisées par l'ancien formulateur, et j'ai du par conséquent choisir un autre nom afin d'éviter les conflits.

- Deux fonctions, de gestion des contextes : pour la sauvegarde et le chargement.
- Une fonction qui recherche un symbole dans la table des symboles selon son nom.
- Enfin, moteur de l'interpréteur, la fonction Parse() qui appelle tout le processus d'analyse.

Après cette description on pourrait croire que la totalité de l'interpréteur est orientée objet, remplissant en cela parfaitement un des objectifs du cahier des charges ; ce qui n'est pas le cas malheureusement. En effet, Bison Flex Wizard génère un projet et les fichiers correspondants, et crée une classe (ici « FormulateurParser »), qui a comme fonction de base Parse (). Mais en fait, je me suis aperçu en recherchant dans le code source généré, pour essayer de comprendre son fonctionnement, que le code objet est bien présent, mais il appelle des fonctions statiques que sont yyparse et yylex. Ces fonctions sont définies dans les fichiers générés. Le problème réside dans le fait, que voulant utiliser la programmation orientée objet, il m'a été impossible d'accéder à mon objet « FormulateurParser », dans le code de yylex ou yyparse. Car si la méthode Parse () de « FormulateurParser », appelle bien yyparse, cette dernière n'appartient pas à l'instance de l'objet, et donc n'a pas accès à ses attributs ou méthodes. C'est pourquoi j'ai du utiliser, des astuces de développement pour pouvoir accéder à mon objet à l'intérieur de ces fichiers, ce qui empêche, l'interpréteur d'être totalement orienté objet.

Les actions associées aux règles

Dans la grammaire présentée sur la Figure 19, les actions ne sont pas rédigées mais sont symbolisées par les accolades « {} ». Comme vous pouvez le constater, il y en a un grand nombre, qui représentent autant de blocs de code plus ou moins longs. Bien entendu, les blocs ne sont pas toujours très conséquents, certains même sont vides, d'autres ne sont constitués que d'une ligne. Mais si la plupart gèrent des opérations relativement simples (les opérateurs arithmétiques par exemple), d'autres peu nombreux, sont de véritables casse-têtes à implémenter, et c'est le cas, bien entendu, des structures de contrôle.

Lorsque j'ai voulu écrire le code de gestion des structures de contrôle, je me suis retrouvé face à un gros inconvénient de l'interpréteur. Je m'explique : dans un compilateur on exécute d'abord les phases d'analyse lexicales et syntaxiques, puis une fois que le code source

a été reconnu conforme au langage, le compilateur génère du code intermédiaire, puis son code final, et l'exécutable, grâce à des informations recueillies pendant l'analyse syntaxique ; et c'est au moment où l'utilisateur va lancer l'exécutable généré par le compilateur qu'il verra le résultat de son code source. Or, dans un interpréteur le code source est analysé et le résultat calculé, dans le même temps, à savoir durant la phase d'analyse syntaxique, ce qui présente d'énormes avantages sur le compilateur au niveau de l'implémentation.

Mais dans le cas d'une structure conditionnelle, que se passe t'il ? Et bien imaginons le code suivant :

```
Debut
        Var1 := 4 ;
        Si (Var1 < 5) Alors
                Var1 := Var1 * 5 ;
        Sinon
                Var1 := Var1 – 10 ;
        FinSi
Fin
```

Figure 21 : Exemple de code source avec une structure de contrôle conditionnelle

Dans ce cas précis l'interpréteur va reconnaître l'expression, il va d'abord évaluer la condition et s'apercevoir qu'elle est vraie, il faudrait donc en toute logique exécuter le code du « Alors ». Ensuite, l'interpréteur va évaluer le code contenu dans le « Alors », pour reconnaître la syntaxe. La valeur de « Var_1 » va passer de 4 à 20. Puis l'interpréteur va continuer son analyse syntaxique, à la recherche d'une erreur. Arrivé dans le «Sinon », il va reconnaître que la syntaxe est correcte et donc évaluer l'action, à savoir que « Var_1 » va passer de 20 à 10, et garder cette valeur jusqu'à la fin de la reconnaissance. La valeur finale de « Var_1 » est erronée, car l'interpréteur aurait du garder la valeur de 20.Mais le problème de l'interpréteur réside dans le fait qu'il est obligé de descendre dans toutes les branches de l'arbre de reconnaissance, pour détecter une erreur de syntaxe, et que dans le même temps il évalue les valeurs de branches qu'il n'aurait pas du exécuter.

C'est pourquoi pour parer à ce problème qui se reproduit dans toutes les structures de contrôles, j'ai mis en place un système de contexte (classe « Contexte »). Ce système de

contexte sauvegarde ou charge les valeurs des variables de la table des symboles, ainsi que certaines valeur des dollars, qui risquent d'être modifiées dans la mauvaise branche. Voici le résumé de ce fonctionnement pour le « Si » et le « Selon » :

Position de l'action	La condition est vraie	La condition est fausse
Après le « Si »		On sauvegarde le contexte On empile le contexte
Après le « Alors »	On sauvegarde le contexte On empile le contexte	On charge le contexte On dépile le contexte
Après le sinon	On charge le contexte On dépile le contexte	

Figure 22 : Gestion des contextes mémoire dans le « Si »

Position de l'action	L'état est vérifié	L'état est faux
Après le « Selon »	On sauvegarde le contexte On empile le contexte	
Après chaque « Cas »	On modifie le contexte de sommet de pile On le sauvegarde	On recharge le contexte de sommet de pile
Après « Défaut »	Si au moins un cas a été vérifié on recharge le contexte de sommet de pile	Si aucun cas ne s'est exécuté, on modifie le contexte de sommet de pile

Figure 23 : Gestion des contextes mémoire pour le « Selon »

Brièvement, le fonctionnement des contextes se résume à sauvegarder le contexte quand l'action devait être exécutée (par exemple l'action du « Alors », si la condition du « Si » est vraie), et on restaure le contexte de sommet de pile si l'action, qui vient peut être de modifier les variables, ne devait pas s'exécuter (par exemple l'action di « Sinon », si la condition du « Si » est vraie).

La sauvegarde des contextes, sauve toutes les variables potentiellement modifiables par l'utilisateur, ainsi que certains dollars (« $n ») qui sont susceptibles d'être modifiés. En effet, lors de la sauvegarde des variables de la table des symboles, Bison modifie (je ne saurais expliquer pourquoi) les valeurs des dollars.

La restauration d'un contexte supprime toutes les variables de la table des symboles, et fusionne la table avec la liste des variables du contexte. Puis restaure les valeurs des dollars sauvegardées.

Les actions des autres règles sont en général moins compliquées, et ne présentent pas un grand intérêt à être détaillées. Notons tout de même, l'action associée à la règle lexicale qui reconnaît les identificateurs des variables, qui appelle la recherche dans la base de données du terme rencontré, puis parcourt la table des symboles pour vérifier son existence. Enfin, si elle n'existe pas elle crée l'entrée dans la table des symboles.

Les fonctions annexes des analyseurs

Pour réduire les traitements on est souvent amené à implémenter des routines qui permettent de supprimer un nombre significatif de lignes de codes. C'est le cas de l'analyseur lexicale qui comporte plusieurs fonctions annexes, tel que « Formulateurtest », qui teste si les valeurs manipulées sont bien des numériques (par exemple dans les traitements des opérateurs arithmétiques), et appelle le code de gestion d'erreur dans le cas contraire. J'ai également implémenté, « FormulateurAppartient » qui teste l'appartenance d'une valeur à un intervalle.

Le code de gestion d'erreurs

Dernier gros développement de l'interpréteur, la gestion des erreurs, s'annonçait comme l'un des plus ardu. En effet, pour avoir un meilleur retour des erreurs à l'utilisateur, j'ai du utiliser plusieurs des techniques de gestion d'erreurs, que nous avons vu précédemment (cf. Première partie – La compilation, 2.6.1 La gestion d'erreurs).

Ainsi, j'ai utilisé la production d'erreurs (voir Figure 19 «*bloc* : **error** »), qui permet d'éviter de sortir brutalement de l'analyseur syntaxique, sur une erreur de syntaxe. Par cette méthode, l'erreur est reconnue comme faisant partie de la syntaxe, de plus on sait exactement quel type d'erreurs a été détecté, ce qui permet de mieux cibler le message.

J'ai également utilisé le mode panique, lorsqu'une erreur entravant significativement l'analyse syntaxique est détectée. Dans ce cas, j'ignore toutes les erreurs qui la suivent, et qui sont dues uniquement à cette erreur précédente. Par exemple dans le cas suivant :

```
Si (Var1 < 5)
        Var1 := Var1 * 5 ;
Sinon
…
```

Ici l'omission du « Alors », obligatoire après la condition, va déclencher une erreur sur le token suivant, à savoir « Var_1 ». Mais n'ayant pas reconnu « Var_1 », il ne reconnaîtra pas « := », etc. Pour éviter de renvoyer une erreur à chaque fois, le mode panique va se resynchroniser sur le premier « ; » qu'il va rencontrer, et ignorer toutes les erreurs, jusqu'à ce point de synchronisation.

Dans tous les cas, la procédure de gestion d'erreurs yyerror (Formulateurerror) est appelée, pour afficher les erreurs. Cette procédure est très sollicitée également par le code des actions associées aux règles. Dans l'analyseur lexical par exemple, un token non reconnu par le langage va renvoyer une erreur lexicale, ou bien l'utilisation d'une chaîne dans une opération arithmétique (autre que l'addition qui effectue également la concaténation de chaînes), va renvoyer une erreur sémantique signifiant l'incompatibilité des types.

La procédure «Formulateurerror » renvoie au total, six types de messages d'erreurs différents :

- Deux pour les erreurs sémantiques : dans le cas où des opérateurs arithmétiques sont utilisés sur une chaîne ou lorsqu'une chaîne est utilisée dans une condition.
- Un message pour les erreurs logiques ou mathématiques : lors d'une division par zéro, par exemple.
- Deux messages différents pour les d'erreurs syntaxiques : dans le cas où une erreur de syntaxe est détectée, l'interpréteur est capable de proposer une solution pour résoudre l'erreur. Mais dans certain cas où il existe trop de possibilités, l'interpréteur renvoie un autre message, avec les propositions de résolution des erreurs les plus courantes (exemple : « vérifiez que vous n'avez pas oublié un ";" »).
- Enfin, dernier type de messages d'erreurs renvoyé par le formulateur : les messages d'erreurs lexicales, dans le cas de l'utilisation d'un symbole ou mot non reconnu par le langage.

Tous ces types de messages sont constitués des mêmes éléments que ceux figurant sur le schéma ci-dessous. Enfin pour résumer l'implémentation de l'interpréteur, la figure 25 vous détaillera la relation entre les différents éléments qui le constitue.

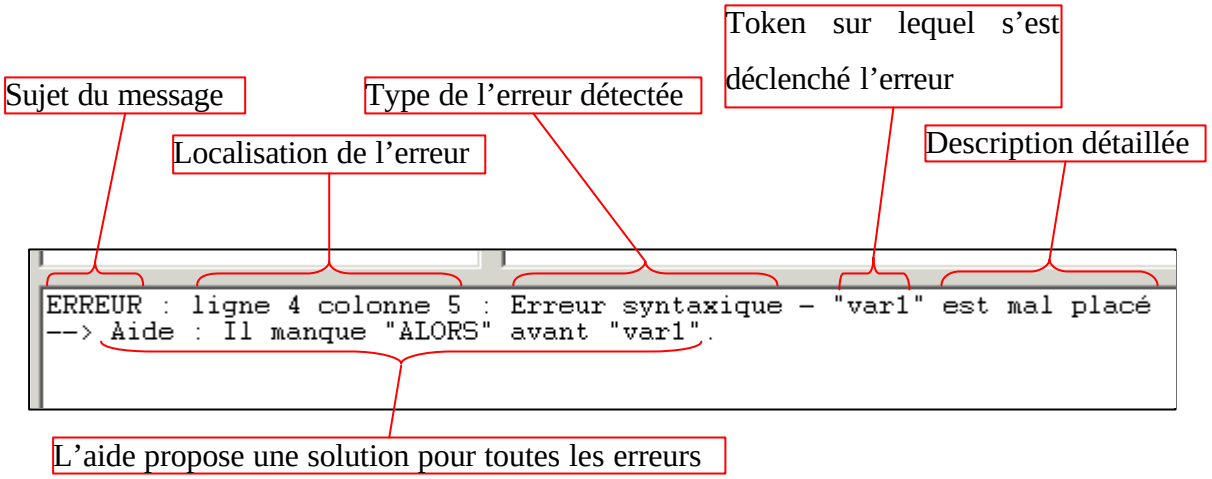


Figure 24 : Structure des messages d'erreurs renvoyés par l'interpréteur

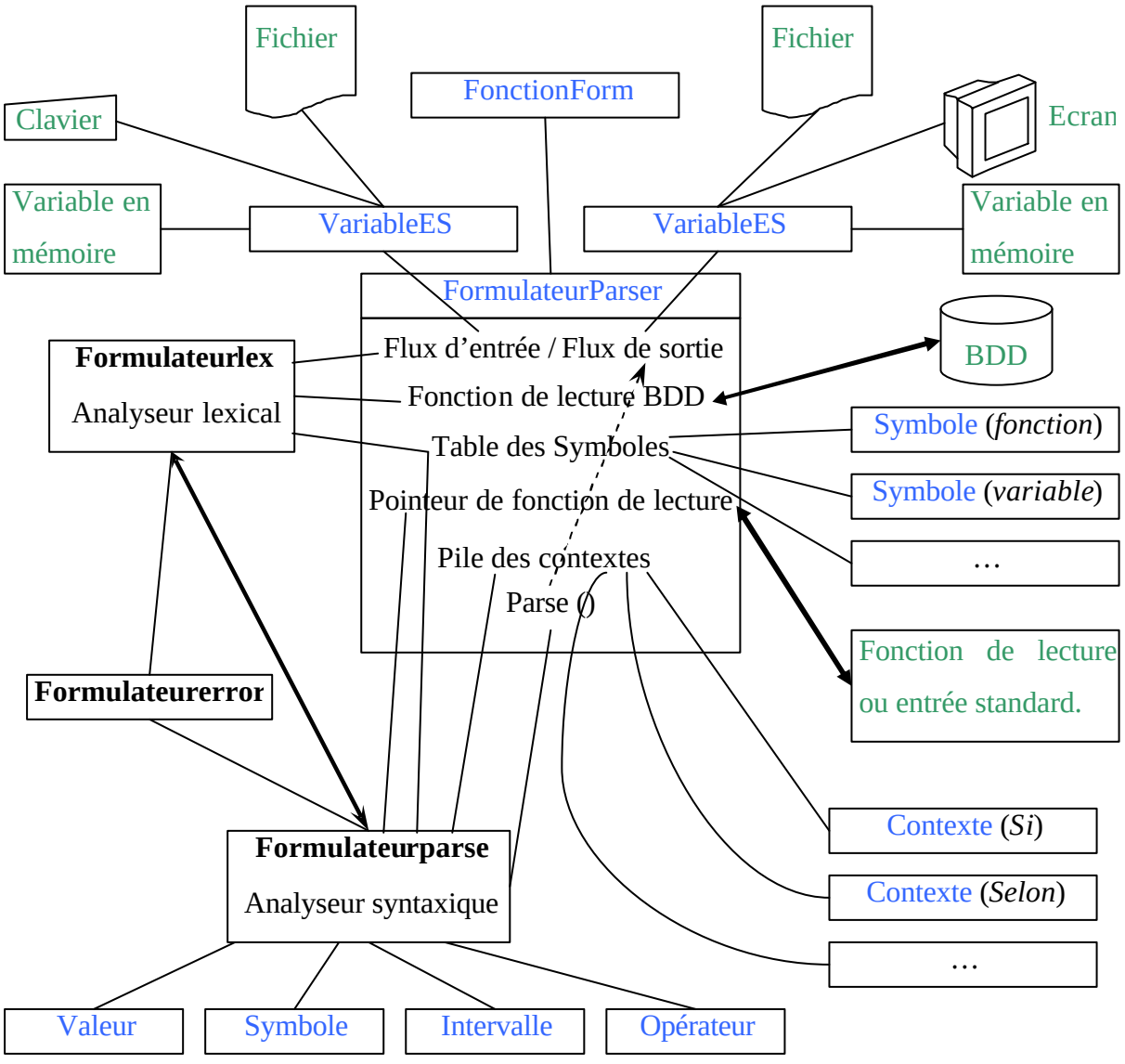


Figure 25 : Résumé du fonctionnement de l'interpréteur

Ce schéma essaye de résumer la relation du Formulateur avec les classes implémentées pour son fonctionnement. Ici les classes sont représentées en bleu, les éléments externes au formulateur sont en vert, et les noms des fonctions sont en gras. Les relations entre les éléments sont représentées par les traits noirs, et le sens de la relation est indiqué par les flèches.

Une fois le formulateur implémenté et fonctionnel, j'ai débuté le développement de l'interface graphique.

4.2 Interface graphique et intégration dans SynOptis

Pour développer l'interface graphique je me suis conformé aux idées exposées plus haut, dans la partie « 3.2.2 L'interface graphique » de cette même section dédiée à la réalisation du projet.

L'implémentation en elle-même ne présente pas un grand intérêt, car il s'agit en effet, de programmation graphique Windows, fait avec Visual C++. La seule classe assez étoffée, est la classe qui définit l'arbre de visualisation de la base de données (tree-view).

Outre l'implémentation de la partie graphique, j'ai du me plonger dans les « entrailles » du progiciel SynOptis, pour modifier certaines classes gérant la base de données, pour pouvoir utiliser des formules de plus de 255 caractères, et pour pouvoir rajouter la visibilité des champs, et des tables. Le progiciel disposant d'un éditeur interne de base de données, j'ai du, également modifier ce dernier pour permettre à l'utilisateur de choisir les champs et les tables visibles. Ces modifications furent difficiles à réaliser, pour deux principales raisons :

- D'une part j'ai du toucher aux classes de bases du progiciel. Or, le progiciel SynOptis est un projet énorme, contenant plus de 400 000 lignes de codes, et en C lorsque que on modifie un fichier librairie (« .h ») il faut recompiler tous les fichiers qui ont inclus ce « .h » ; ce qui demande près de deux heures !
- D'autre part, j'ai du reprendre un code qui n'est pas le mien, très peu documenté, et qui est d'une incroyable complexité, d'où ma difficulté à le modifier.

Par conséquent, pour ne pas vous ennuyer avec l'implémentation graphique du formulateur, je vous présenterai simplement les fenêtres modifiées, ou crées dans la partie résultats.

En ce qui concerne l'intégration de l'interpréteur, dans le progiciel Synoptis, elle s'est révélée grandement simplifiée par l'implémentation de la classe VariableES (voir plus haut). Il m'a suffit alors d'appeler la fonction «ChangeStream » de la classe «FormulateurParser » avec pour paramètres les variables liées au contenu des zones textes de l'interface graphique, et le sens du flux (entrée ou sortie). Ainsi, toute l'opération d'intégration doit intégralement son succès à l'implémentation orientée objet du formulateur.

Pour finir, croyez bien que la place réservée dans ce mémoire au développement de l'interface graphique, n'est absolument pas proportionnelle à son temps d'implémentation. En effet, cette partie m'a demandée quasiment autant de temps, sinon plus, que le développement de l'interpréteur en soi.

4.3 Phase de déboguage

On appelle communément « phase de déboguage », la partie du développement comprenant l'ensemble des traitements et corrections destinés à éliminer les erreurs de fonctionnement (bogues) d'un logiciel.

Or, il est très difficile d'effectuer cette phase dans le développement d'un langage de programmation, car il existe une infinité de cas à tester. Il est bien entendu possible de tester si la syntaxe est bien reconnue, mais il quasiment impossible, de savoir si les traitements s'exécuteront toujours comme dans l'exemple que l'on a validé. Par exemple, lors de l'implémentation des structures de contrôles, le «si » fonctionnait, tout comme le «selon », indépendamment l'un de l'autre. Mais, en essayant une formule contenant un «si » imbriqué dans un « selon », une erreur incompréhensible est apparue, qui était due à un conflit dans les contextes.

C'est pourquoi la phase de déboguage de l'interpréteur nécessitera, un certain temps d'utilisation par les employés de la société SynOptis, avant de pouvoir être stabilisé. J'ai pour ma part consacré, beaucoup de temps pour tester des formules, et enlevé tous les bogues

relevés. Mais le temps demandé par une phase de test complète, sort complètement du cadre de mon stage effectué au sein de la société SynOptis.

Au niveau de l'interface graphique, la phase de déboguage fut beaucoup plus simple. En effet, j'ai testé tous les boutons et fonctions de cette partie, et traqué les bogues principaux.

Dans tous les cas, je me suis assuré que les fonctionnalités demandées dans le cahier des charges sont bien toutes implémentées, et prête à être utilisées. Je ne prétends certes pas être le premier développeur à écrire un programme sans bogues, mais le formulateur a atteint un niveau de développement suffisamment stable, pour commencer à être utilisé.

4.4 Travaux connexes

La société SynOptis dans laquelle j'ai effectué mon stage de fin d'études, est on l'a dit une petite entreprise (six employés). Or, la place d'un stagiaire ou d'un employé, dans ce genre de structure, se révèle être beaucoup plus importante que dans une multinationale, où chaque employé à une fonction propre. En effet, il arrive souvent de devoir effectuer des tâches qui sortent du cadre de son poste. L'une des plus répandue, et des moins appréciée des stagiaires, et d'assurer un minimum d'accueil aux clients, à savoir répondre au téléphone, lorsque les employés sont déjà occupés.

Mais durant ces six mois, j'ai souvent du parer au manque de temps des titulaires, et du les décharger de tâches annexes, comme l'achat et l'installation de matériel pour l'arrivée des autres stagiaires au sein de la société, qui n'avait pas une capacité d'accueil suffisante, en terme de matériel (ordinateurs, écrans, chaises,…). Avec l'arrivée des nouveaux stagiaires, nous avons du également effectuer, ensemble, la restructuration des bureaux, et la réorganisation des espaces de travail.

De plus, avec un autre stagiaire, nous avons entamé la restructuration et la sécurisation du réseau qui menaçait de s'effondrer sous la surcharge des nouveaux ordinateurs. En effet, les disques durs du serveur contenant les sources, montraient des signes de faiblesses, et ceux de la passerelle Internet étaient complètement hors d'usage (arrêts intempestifs). C'est pourquoi nous avons convaincu le dirigeant, M. Rodes, d'acheter un nouveau serveur, avec

des disques durs sécurisés pour éviter la perte de données importante (système RAID). L'ancien serveur est devenu une passerelle sous Linux qui sert également de Firewall et gère toute la sécurité du réseau. Enfin, l'ancienne passerelle est devenue un serveur d'impression, où ses disques fatigués ne menacent pas la perte de données. Le schéma de ce réseau est disponible dans les annexes.

Tous ces travaux annexes sortaient certes totalement du cadre de mon projet, mais sont courantes dans des entreprises de cette taille.

V. RESULTATS

Dans cette dernière partie je présenterai, les résultats obtenus à la fin de mon projet. Je vous proposerais donc, les captures d'écran des fenêtres réalisées, ainsi que l'étude de l'adéquation du module, avec le cahier des charges originel.

5.1 Aspect du formulateur réalisé

5.1.1 La fenêtre de sélection des formules

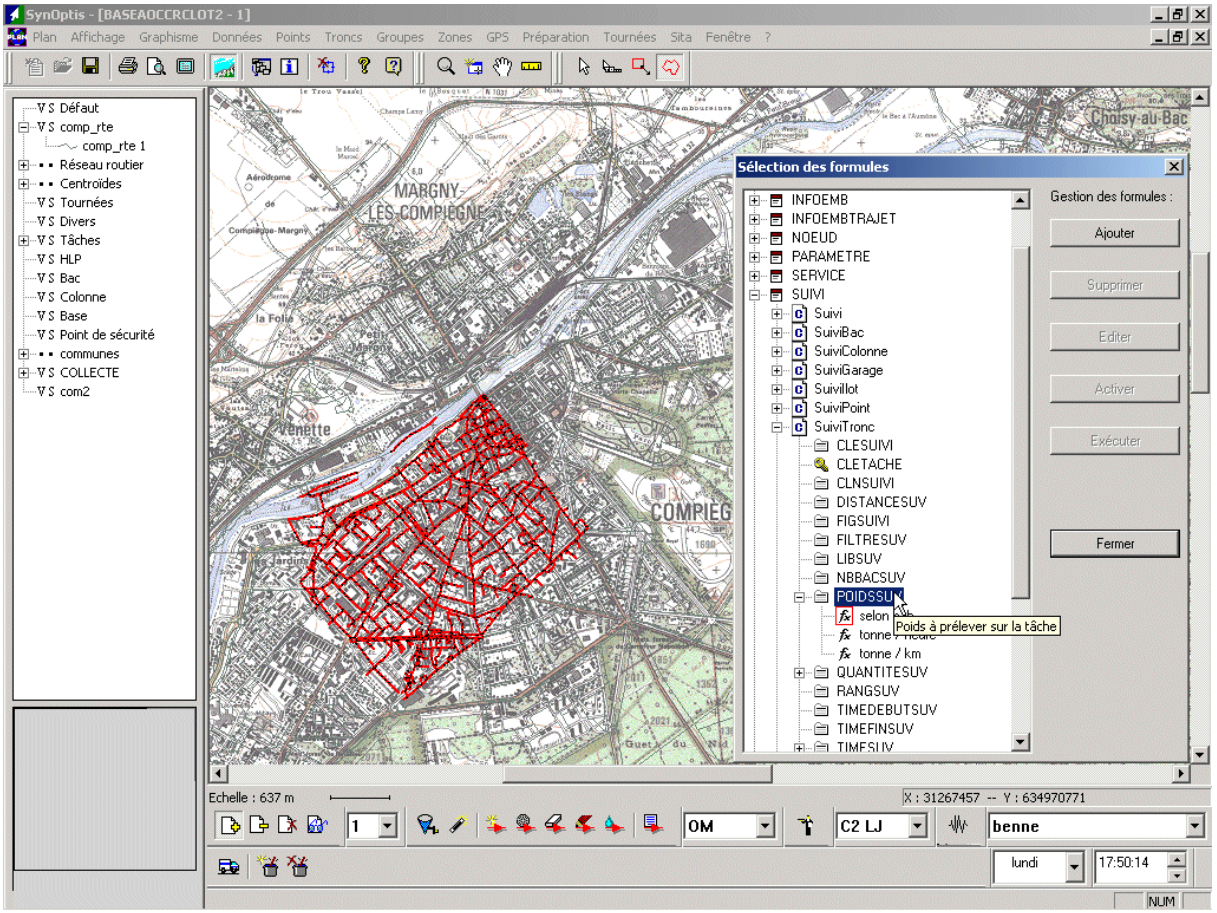


Figure 26 : Fenêtre de sélection des formules

On accède au formulateur, par le menu « Données » du progiciel SynOptis. Lorsqu'on

a cliqué sur la ligne formulateur, apparaît la fenêtre de sélection des formules (ci-dessus avec en fond l'application SynOptis). Observons cette fenêtre de plus près :

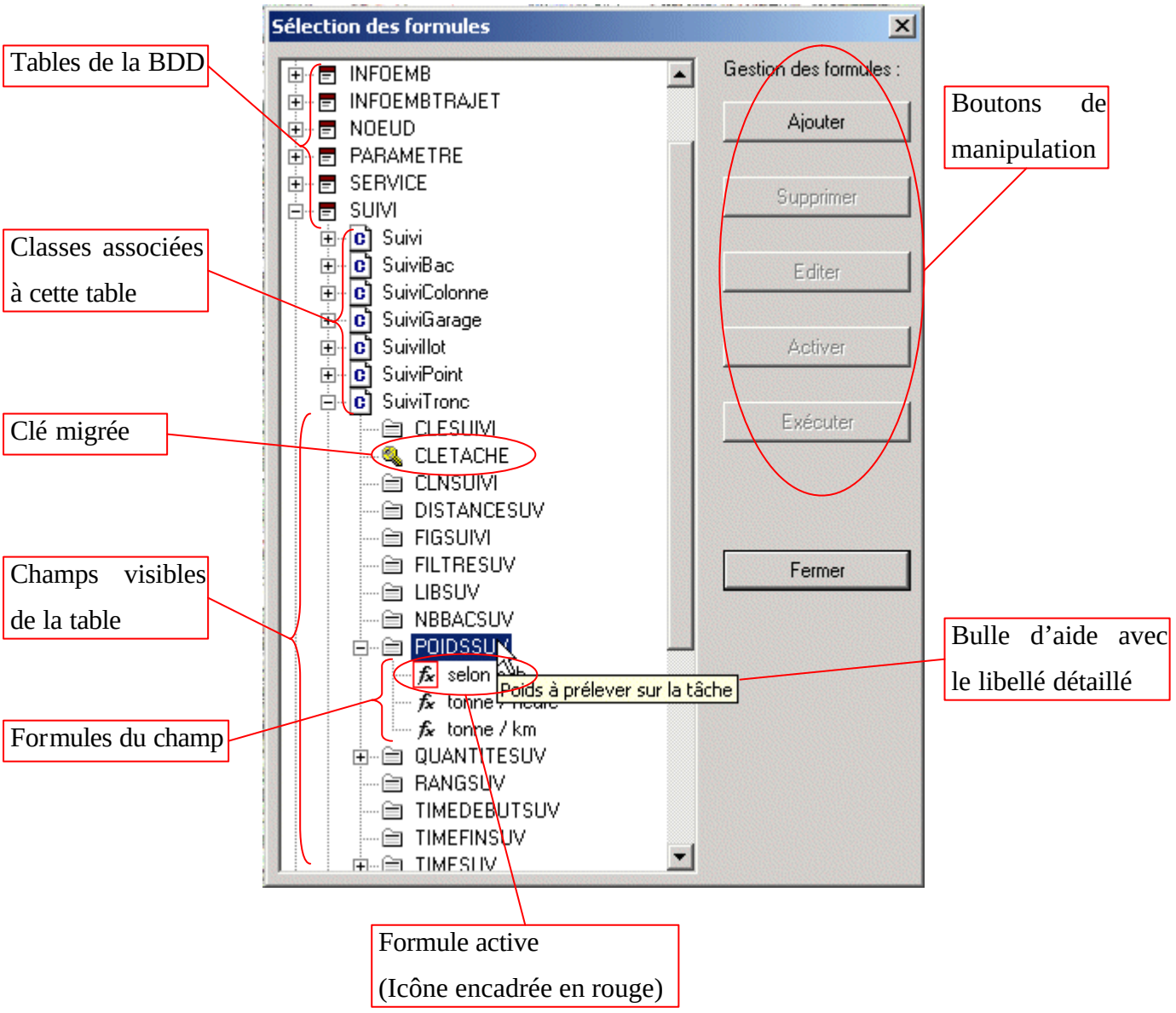


Figure 27 : Détail de la fenêtre de sélection

Quelques précisions :

- Les boutons ne s'activent que lorsqu'on est positionné sur un champ (bouton « ajouter »), ou sur une formule (tous les autres).
- Les classes correspondent aux types de données que l'on peut manipuler. Par exemple, ici on choisit la table « suivi » pour effectuer les calculs des champs du suivi d'une trournée, la classe « SuiviTronc » pour signifier que l'on veut

s'occuper uniquement des troncs du réseau routier (rue, avenue,…), et le champ « PoidsSuv » à savoir le « Poids à prélever sur la tâche ». Pour calculer ce champ on a choisi la formule active «selon habitant », qui donneras l'estimation voulu sur la tournée.

Une fois la formule de calcul choisie, lorsque l'on crée une tournée sur un quartier, le progiciel estimera le champ voulu en fonction de la formule validée. Par exemple, dans le schéma ci-dessous, on a crée des tournées sur un quartier de Bayonne, en imposant une contrainte de poids de 800kg, et en choisissant la formule de calcul « poids/km » pour estimer le poids sur la tournée.

Figure 28 : Estimation des formules sur une tournée

5.1.2 L'éditeur de formules

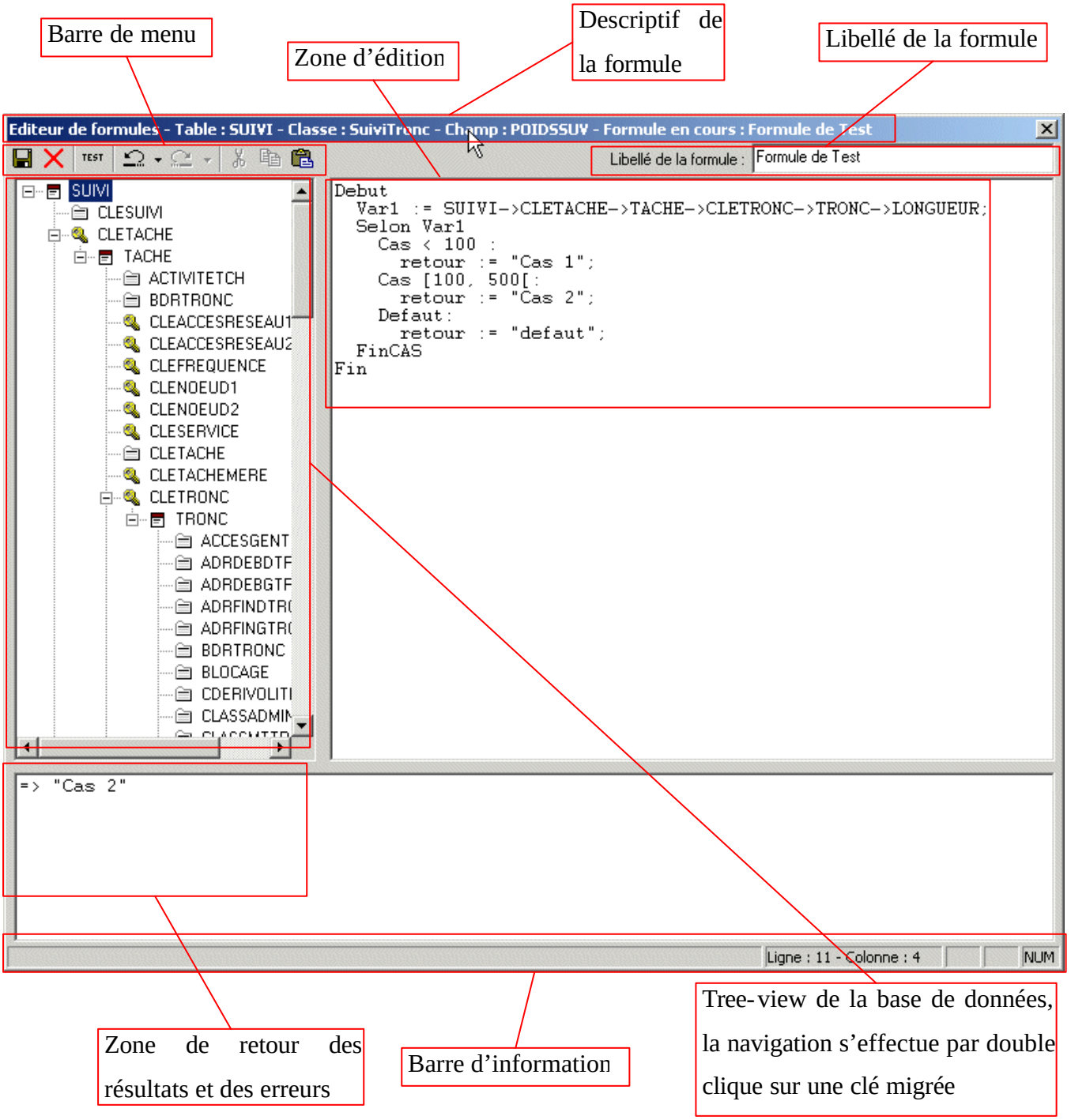


Figure 29 : Fenêtre d'édition des formules

L'éditeur présente également des options d'éditions avancées, comme le menu d'annulation et de rétablissement multiple :

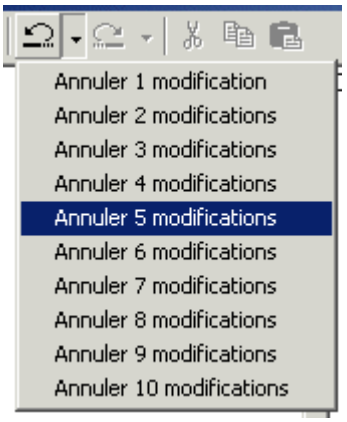


Figure 30 : Menu d'annulation multiple

Il n'y a rien de bien particulier à relever sur cette fenêtre, mis à part le bouton « Test » qui lance l'interpréteur.

5.1.3 L'éditeur de base de données

L'éditeur de base de données était un module déjà existant dans le progiciel SynOptis. Je l'ai modifié pour permettre aux utilisateurs de décider eux-mêmes s'ils voulaient masquer les champs ou les tables, dans le formulateur. Ce développement n'étant pas primordial, bien qu'il m'ait pris beaucoup de temps, vous ne trouverez les captures d'écrans qu'en annexe (cf. annexe 4).

5.2 Adéquation avec la demande

Pour pouvoir répondre à cette question, je vais reprendre un par un les objectifs de ce projet.

Refonte du formulateur :

- « *Réaliser un langage de programmation performant et facile d'accès pour les néophytes* »
  - Le nouveau langage du formulateur affiche des performances très correctes dans le sens ou la manipulation des données ne dépasse jamais quelques secondes. Mis à part ces considérations, il est difficile de calculer avec

précision la performance du nouvel interpréteur.

- En terme de puissance, le formulateur a atteint ses objectifs, puisque toutes les fonctionnalités demandées sont implémentées.
- Au niveau de la facilité d'utilisation, le langage a également rempli son but, puisqu'il offre la syntaxe la plus simple, qu'un outil de la sorte puisse avoir. En effet, même si une formation pour les clients sera toujours nécessaire, il n'est guère possible de simplifier plus le langage actuel (absence de déclaration de variable, variable non typée, conversion automatique, absence d'utilisation des blocs dans les structure de contrôles,…) sans lui faire perdre sa fonctionnalité.

- « *Le formulateur ne doit pas être perturbé par l'utilisation d'espace, ou de retours chariot.* »
  - Les espaces, les tabulations et les retours chariot sont gérés.
- « *Le langage doit savoir manipuler des valeurs numériques, ou des chaînes de caractères.* »
  - Grâce à la classe « Valeur » le formulateur manipule indifféremment les valeurs numériques et les chaînes de caractères.
- « *Le langage doit gérer les opérations arithmétiques, ainsi que les principales fonctions mathématiques (cos, sin, …).* »
  - Les opérateurs arithmétiques, les puissances, et les fonctions mathématiques courantes sont implémentés, avec la possibilité de rajouter n'importe quelles fonctions, par le biais de la classe « FonctionForm ».
- « *Il doit être également possible d'utiliser les opérateurs relationnels (opérateur de comparaison : « < », « > », …), ainsi que les opérateurs booléens (« et », « ou » et « non »).* »
  - Tous ces opérateurs sont implémentés et fonctionnels.
- « *Le langage du formulateur doit permettre la saisie de variables, soit directement par une affectation, soit lorsqu'une variable est utilisée sans être initialisée, le formulateur doit demander sa valeur.* »
  - Le formulateur gère les variables en nombre illimités et il est bien entendu possible de faire des affectations.
  - La lecture d'une valeur pour l'initialisation est gérée grâce à un pointeur sur une fonction externe.

- « *Les structures de contrôles « If » et « Case » doivent être implémentées. Il doit être également possible d'imbriquer plusieurs structures de contrôles.* »
    - Les structure des contrôles « Si » et « Selon » sont en état de marche, avec pour le « Selon » une praticité que nous aimerions avoir dans les langages de programmation de haut niveau.
    - L'imbrication des structures est gérée par la classe « Contexte »
- « *Le formulateur doit être capable de lire un code source directement depuis la mémoire sans passer par un fichier sur le disque dur.* »
    - La classe « VariableES » implémente cette fonctionnalité.
- « *Le module doit, au maximum être implémenté en programmation orienté objet.* »
    - L'interpréteur a été développé le plus possible en langage orienté objet, mais les limitations de Flex et Bison, ont définitivement avortées cet effort d'implémentation.
- « *Le langage de programmation doit être prévu pour manipuler les champs de la Base De Données, pour lire une valeur ou pour la modifier.* »
    - L'interpréteur est capable d'aller chercher une valeur dans la base de données, de vérifier qu'il existe bien une relation entre des champs et des tables, mais à l'heure ou j'écris ces lignes, la modification de la base de données et la dernière étape à boucler pour finir le projet.
- « *Enfin le formulateur doit posséder un meilleur code de gestion des erreurs.* »
    - Le code de gestion d'erreurs, inexistant dans la première version du formulateur, est maintenant implémenté avec les différentes méthodes les plus performantes, pour donner la meilleure information à l'utilisateur.

Refonte de l'interface graphique :

- « *La nouvelle interface doit être plus claire, et plus fonctionnelle que le précédent système, qui était très hermétique et inabordable*. »
    - C'est une appréciation très subjective et je vous laisse seul juger, entre le système de l'ancien formulateur (Figure 16, p48) et le nouveau système de des figures 27 et 29, lequel vous trouvez le plus clair.
- « *Le nouvel éditeur doit permettre de saisir des formules de plus de 255 caractères.* »

    - La base de données a été modifiée, et l'éditeur crée pour gérer ce problème.
- « *Il faut également trouver un système pour limiter le nombre de tables et de champs accessibles dans le formulateur, pour éviter de modifier des champs réservés.* »
    - J'ai modifié la base de données, et l'éditeur de dictionnaire pour tenir compte de la visibilité des champs (cf. Annexe 4).

Le comparatif des objectifs et des réalisations, montre bien que le formulateur n'a plus qu'une fonctionnalité à acquérir pour être conforme à la demande de la société SynOptis. Le développement n'est certes pas tout à fait complet, mais les tâches annexes réalisées au sein de l'entreprise (restructuration du réseau,…), l'ont amputé d'un temps précieux. Néanmoins, le développement de cette dernière fonctionnalité sera court, car il ne nécessite qu'une fonction de modification de la base de données dans la progiciel SynOptis, et n'appelle pas à une modification de l'interpréteur.

CONCLUSION

Résumé

Ce mémoire est la synthèse des recherches et des développements effectués pour la réalisation d'un interpréteur, permettant la saisie de formules de calcul pour les champs de la base de données d'un progiciel d'optimisation, dans le but d'augmenter sa puissance. Ce besoin se justifie par la nature même d'un progiciel, qui est un logiciel très paramétrable pour convenir à plusieurs professions d'utilisateurs et à plusieurs domaines d'applications. Pour effectuer ce projet, un domaine très particulier de l'informatique fut choisi : la Compilation, qui est, en effet, spécialisée pour la conception d'un langage de programmation.

Comme tout travail de recherches ambitieux, ce document expose d'abord les études et les approfondissements personnels que j'ai du réaliser préalablement, au développement, afin de mieux comprendre et aussi de vous expliquer les concepts théoriques de la compilation. En effet, la compilation présente nombre de concepts compliqués telles que les grammaires, les automates, l'analyse lexicale, l'analyse syntaxique, etc. Pour résumer, ces concepts, voici une courte définition de chacun :

- La grammaire est un ensemble de règles définissant la syntaxe d'un langage. C'est l'élément central de la création d'un langage, car c'est grâce à elle qu'il prend forme. Cette grammaire ne doit en aucun cas être ambiguë. On dit qu'une grammaire est ambiguë s' il existe plusieurs phrases reconnues par le même arbre de dérivation
- Selon une définition simplifiée, un automate est un graphe orienté, dont les arcs sont valués avec les lettres et symboles reconnus par le langage. L'automate possède également une dérive qui le rend inexploitable : le non déterminisme, qui comme l'ambiguïté des grammaires, rend l'analyseur qui s'en sert, non fonctionnel.
- L'analyse lexicale, quant à elle, est la première opération opérée par le compilateur ou l'interpréteur, elle permet de reconnaître le lexique (vocabulaire) du langage, en découpant le flux d'entrée en tokens ou unité lexicale.
- L'analyse syntaxique est la phase qui vérifie si les unités lexicales renvoyées par l'analyseur lexical sont placées dans un ordre correct, c'est-à-dire conforme à la syntaxe du langage.

La compilation qui reste un domaine compliqué de l'informatique fut extraordinairement simplifiée, par l'utilisation des métacompilateurs qui automatisent l'implémentation de l'analyseur lexical, et de l'analyseur syntaxique. En effet, les métacompilateurs créent l'automate nécessaire à l'analyse lexicale, à partir des règles de lexique rentrées ; et la grammaire de l'analyse syntaxique grâce aux règles syntaxique, qu'on leur donne.

Les principes de compilation sont, certes très ciblés à certains domaines d'applications, mais conviennent parfaitement à la réalisation de mon projet. Néanmoins, je n'implémenterai qu'un interpréteur qui possède toutes les phases nécessaires pour mener à bien mon développement, en offrant une puissance suffisante, et un temps de développement très léger par rapport à celui d'un compilateur complet.

Une fois présenté cette étude théorique, passons à la phase de réalisation du formulateur. J'ai tout d'abord étudié l'existant pour en ressortir les défauts et lacunes, que je dois absolument éviter, et corriger dans le nouveau formulateur. L'étude de l'existant m'a amené à me rendre compte que je ne pourrai pas me baser sur l'ancien formulateur, mais que je devrai entreprendre entièrement la refonte de l'interpréteur et son interface graphique. Après avoir choisi Bison Flex Wizard pour implémenter le langage, je débutais donc le développement du projet par le concevoir sur papier. Une fois la grammaire capable de reconnaître la totalité des expressions du langage de programmation, et les idées principales pour simplifier l'interface graphique validée, j'ai pu débuter la phase d'implémentation du formulateur.

L'interpréteur comporte de nombreuses classes, que je ne redétaillerai pas ici. Sachez toutefois que l'interpréteur réalisé, est capable grâce à ses classes de lire et d'écrire sur n'importe quel flux d'entrée/sortie ; et qu'il est « presque » entièrement orienté objet. En effet les limitations de Flex et Bison empêchent d'implémenter un compilateur totalement orienté objet. Le formulateur en l'état actuel est totalement indépendant de l'interface graphique.

Allié à la refonte de l'interface graphique opéré, le formulateur est aujourd'hui entièrement fonctionnel.

Apports du projet

Ce projet m'a apporté énormément de connaissances techniques sur la compilation, en général, et sur la partie analytique surtout. J'ai, en outre appris à maîtriser les outils Flex et Bison (Lex et Yacc), en réussissant à comprendre, et parfois même à interférer dans le fonctionnement interne des fichiers générés. Enfin, j'ai pu me familiariser avec l'outil de développement de Microsoft, Visual C++ qui m'était alors totalement inconnu, et par la même approfondir mes connaissances en C.

Le travail au sein de la société SynOptis m'a également apporté d'autres connaissances techniques, en effectuant notamment la restructuration du réseau ; mais m'a surtout permis de travailler en équipe sur certaines parties du développement.

Ouverture et évolutions futures

Le formulateur dans l'état actuel du développement est fonctionnel et actuellement testé par les employés de la société SynOptis. Mais ma collaboration au projet, n'est pas terminée. En effet, j'ai signé avec la société SynOptis, un Contrat à Durée Determiné de 6 mois, pour finir les dernières modifications de développement, et pour rajouter de nouvelles fonctionnalités au formulateur :

- La reconnaissance de syntaxe, c'est-à-dire différencier par des couleurs et des effets de police, les types d'unité lexicale (mots réservés, variables, fonctions, etc.)
- L'auto-indentation du code.
- La saisie semi-automatique assistée, à savoir proposer à l'utilisateur les choix possibles au fur et à mesure de la saisie.
- Enfin, le développment le plus important sera sans aucun doute, de doter le progiciel SynOptis, de « macros », similaires à Word ou Excel, pour permettre à l'utilisateur, via le formulateur de piloter le logiciel avec des scripts.

Voici donc les évolutions futures du formulateur, au sein du progiciel SynOptis. Ces évolutions vont permettre à long terme de faciliter l'utilisation et la maintenance de

l’application (grâce aux macros notamment), et ainsi rendre l’applicatif SynOptis un peu plus professionnel, et fonctionnel.

GLOSSAIRE

AMBIGUÏTE

Caractère d'une grammaire à avoir plusieurs arbres d'analyse pour une phrase donné, à savoir qu'une grammaire pour être exploitable ne doit pas, si possible, être ambiguë.

ANALYSE LEXICALE

Partie de l'analyse syntaxique qui divise les chaînes de caractères en mots. L'analyse lexicale utilise, entre autres, les signes de ponctuation pour séparer les chaînes de symboles en éléments utilisables pour l'analyse syntaxique.

ANALYSE SYNTAXIQUE OU SEMANTIQUE

Décomposition d'une phrase ou d'une chaîne de caractères en segments, au moyen de règles prédéfinies, dans le but de comparer la structure de ces segments avec celles définies dans une grammaire de référence. En informatique, on peut utiliser l'analyse syntaxique dans plusieurs cas. On l'utilise, entre autres, pour vérifier la conformité syntaxique d'un programme avec son langage de programmation, avant que le programme ne soit lancé, on l'utilise dans un compilateur pour traduire en langage machine un programme écrit en langage source, on l'utilise dans les programmes de SGBD et les systèmes experts pour traduire le langage humain en une forme exploitable par le programme, et enfin, on peut l'utiliser en intelligence artificielle pour la reconnaissance de la parole ou la lecture de textes numérisés.

AXIOME

Non terminal duquel on débute la dérivation.

BASE DE DONNEES (B.D.D.)

Ensemble structuré de fichiers interreliés dans lesquels les données sont organisées selon certains critères en vue de permettre leur exploitation.

B.N.F. (BACKUS-NAUR FORM)

Métalangage utilisé pour décrire et exprimer la syntaxe des langages de programmation. La forme de Backus-Naur est constituée de caractères et de symboles

spéciaux qui permettent de décrire la façon dont s'organise une grammaire sans devoir se référer à un langage de programmation. Développée par John Backus et Peter Naur vers 1960, la forme de Backus-Naur a été le premier métalangage à apparaître. À l'époque, il avait d'abord été créé pour définir le langage ALGOL 60.

COMPILATION

Action de traduire automatiquement, à l'aide d'un compilateur, ou d'un interpréteur, un programme écrit en langage de haut niveau en un programme équivalent en langage machine. Le code d'un programme ayant subi une compilation peut ensuite être exécuté sur une plate-forme informatique précise.

EMULATEUR

Logiciel ou matériel permettant de rendre compatible un système avec un autre non compatible à l'origine. On peut utiliser un type d'émulateur pour obtenir que deux modèles d'imprimantes se comportent de la même façon et un autre type pour permettre d'utiliser le même programme sur deux ordinateurs non compatibles.

GENIE LOGICIEL

Ensemble des connaissances, des procédés et des acquis scientifiques et techniques mis en application pour la conception, le développement, la vérification et la documentation de logiciels, dans le but d'en optimaliser la production, le support et la qualité.

GRAMMAIRE

Ensemble de règles qui définisse un langage. La grammaire en compilation est souvent normalisée sous forme B.N.F.

ITERATIF

Qualifie un traitement ou une procédure qui exécute un groupe d'opérations de façon répétitive jusqu'à ce qu'une condition bien définie soit remplie.

IMPLEMENTER

Réaliser la phase finale d'élaboration d'un système, afin que le matériel et les logiciels soient concrètement opérationnels.

INCUBATEUR

Organisme qui aide de nouvelles entreprises à démarrer en leur fournissant des locaux, des services multiples, des conseils et de la formation jusqu'à ce qu'elles deviennent autonomes, et en favorisant les échanges avec des entreprises déjà installées. Parmi les services proposés par ces organismes, on peut trouver la gestion des ressources humaines, la recherche de financement et de partenariats industriels, la commercialisation, le recrutement, l'assistance juridique, la rédaction du plan d'affaires, la gestion de la propriété intellectuelle, la logistique, l'expertise comptable. Quant à l'environnement matériel mis à la disposition des jeunes entreprises, il peut comprendre la messagerie téléphonique, la photocopie, la télécopie, l'équipement informatique et la mise en réseau, le secrétariat, l'assistance technique informatique, le développement d'un site Web, la conférence vidéo. *Pépinière d'entreprises* peut désigner aussi l'édifice où est établi cet organisme.

INFORMATIQUE EMBARQUEE

Branche de l'informatique qui traite de l'intégration de systèmes informatiques à des dispositifs, machines ou systèmes (ex. : appareils ménagers, machines industrielles, avions, automobiles, missiles, montres, appareils médicaux, etc.), et ce, dans le but d'en assurer le pilotage.

INTELLIGENCE ARTIFICIELLE (I.A.)

Discipline scientifique relative au traitement des connaissances et du raisonnement humain dans le but de les reproduire artificiellement et ainsi de permettre à un appareil d'exécuter des fonctions normalement associées à l'intelligence humaine : raisonnement, compréhension, adaptation, etc.

INTERPRETEUR

Programme qui traduit les instructions d'un langage évolué en langage machine et les exécute au fur et à mesure qu'elles se présentent

LANGAGE DE PROGRAMMATION

Langage artificiel comprenant un ensemble de caractères, de symboles et de mots régis par des règles qui permettent de les assembler, utilisé pour donner des instructions à un ordinateur. L'une des principales qualités d'un langage de programmation est la portabilité qui permet à un logiciel développé dans un langage de programmation à grande portabilité, de

fonctionner sur différents types de systèmes informatiques. Le terme « langage de programmation » recouvre une vaste panoplie, allant du langage machine formé d'instructions binaires, aux langages de haut niveau dont la forme se rapproche des langages naturels.

METACOMPILATEUR

Programme utilitaire servant à produire le compilateur d'un langage. Si on fournit les spécifications d'un langage à un métacompilateur, il produit un compilateur pour ce langage

MODELISATION STOCHASTIQUE

Méthode mathématique qui utilise le calcul des probabilités pour l'exploitation des données statistiques.

NON TERMINAL

Symbole rencontré dans les règles de production de la grammaire, et qui est dérivable (qui se trouve au moins une fois à gauche d'un flèche).

PROGICIEL

Logiciel d'application paramétrable, destiné à la réalisation de diverses tâches. Le logiciel de traitement de texte Word, par exemple, est un logiciel d'application dans la mesure où il informatise une tâche. C'est aussi un logiciel bureautique puisque les tâches informatisées sont des tâches de bureau. Mais ce n'est pas un progiciel puisqu'il n'est pas paramétrable. Cependant, les versions à venir le seront de plus en plus. AutoCAD est un exemple de progiciel de conception et de dessin assistés par ordinateur parce qu'il est paramétrable selon les besoins spécifiques de ses utilisateurs (architectes, dessinateurs, ingénieurs, techniciens) et de ses domaines d'application (électricité, mode, usinage, conception de navires et d'avions).

PROGRAMMATION ORIENTE OBJET (P.O.O.)

Dont les constituants sont des objets composés de leurs données définitoires et de leurs procédures de manipulation.

PROGRAMMATION STRUCTUREE

Caractère d'un langage, a utilisé des blocs structurés par un symbole de début et un de fin. Un langage structuré ne doit pas exécuter de Goto.

RECHERCHE OPERATIONNELLE

Ensemble des méthodes, le plus souvent mathématiques, statistiques et informatiques, conduisant à l'optimisation des décisions à partir d'une analyse systématique des données d'un problème posé par une activité humaine ainsi que d'une réflexion logique sur les facteurs en cause et les options possibles.

TERMINAL

Symbole rencontré dans les règles de production de la grammaire, qui n'est pas dérivable (qui ne se trouve jamais à gauche d'une flèche).

TOKEN

Plus petite identité lexicale reconnue. Les tokens sont renvoyés par l'analyseur lexical, à l'analyseur syntaxique lorsqu'il les a reconnus. Souvent token est employé comme synonyme de mot.

VARIABLES

Caractère ou groupe de caractères qui représente une valeur et qui, lors de l'exécution d'un programme de calculateur, correspond à une adresse.

LISTE DES TABLEAUX ET FIGURES

TABLE DES MATIERES

BIBLIOGRAPHIE

ALFRED AHO, RAVI SETHI et JEFFREY ULLMAN [1989], « COMPILATEURS Principes, techniques et outils », Paris, InterEditions

ETIENNE BERNARD [1997], « Mini manuel d'utilisation Lex et Yacc », Institut Supérieur d'Informatique et d'Automatique, Sophia Antipolis : http://www.linux-france.org/article/devl/lexyacc/minimanlexyacc.html

PHILIPPE BILLARD [2001], «Les 35 technos de l'après PC – Les langages de développement (Techno 20) » dans *01 Informatique*, Paris, du 15 octobre 2001

J. BONNEVILLE [1999], « Cours de techniques de compilation – Documentation Lex – Flex », Université de Lyon : http://www710.univ-lyon1.fr/~ascil/docs/flex_bonnev/Lex.html

J. BONNEVILLE [1999], « Cours de techniques de compilation – Documentation Yacc – Bison », Université de Lyon : http://www710.univ-lyon1.fr/~ascil/docs/bison_bonnev/Yacc.html

DOMINIQUE BOUCHER, PH. D. [1995], « IFT6820 : Langages de programmation et compilation », Université Montréal : http://www.iro.umontreal.ca/~boucherd/Cours/ift6820/semaine1.html

JUTTA DEGENER [1995], « The ANSI C Grammar – Lex Specification », Lysator, Lynköping University : http://www.lysator.liu.se/c/ANSI-C-grammar-l.html

JUTTA DEGENER [1995], « The ANSI C Yacc Grammar », Lysator, Lynköping University : http://www.lysator.liu.se/c/ANSI-C-grammar-y.html

CHARLES DONNELLY et RICHARD STALLMAN [1995], « Bison – The YACC-compatible Parser Generator » : http://dinosaur.compilertools.net/bison/index.html

J. FERBER [1997], « Cours de Compilation », Université de Montpellier : http://www.lirmm.fr/~ferber/Compilation/compil.htm

ANDREW FERGUSON [2000], « The History of Computer Programming Languages. », Princeton New Jersey : http://www.princeton.edu/~ferguson/adw/programming_languages

HENRI GARRETA [1999], « Le langage et la bibliothèque C++ - Norme ISO », Université de Luminy : http://www.dil.univ-mrs.fr/~garreta/Polys/PolyCpp.pdf . La version complète n'est plus disponible sur le site internet, il s'agit d'un fichier pdf allégé, la version complète est éditée maintenant sous le même titre aux éditions Ellipses.

HENRI GARRETA [2001], « Techniques et outils pour la compilation », Université de Luminy : http://www.dil.univ-mrs.fr/~garreta/Polys/PolyCompil.pdf

FRANÇOIS GUILLIER [2002], « Histoire de l'informatique » : http://www.histoire-informatique.org

STEPHEN T. JOHNSON [1974], « YACC : Yet Another Compiler-Compiler », AT&T Bell Laboratories, New Jersey : http://dinosaur.compilertools.net/yacc/index.html

JERZY KARCZMARCZUK [2002], « Compilateurs et interprètes », Université de Caen : http://users.info.unicaen.fr/~karczma/matrs/Maitcomp/compilation_h.pdf

J.P. KRIMM [2001], « Structure générale d'un compilateur », Université de Grenoble : http://www-imss.upmf-grenoble.fr/IMSS/dciss/Enseignements/TAL/Compil/01intro.ps

CEDRIC LECLERC et JULIEN MOREAU [2000], « Rapport de projet de Compilation », Ecole ISTY Informatique, Versailles : http://romuald.isty-info.uvsq.fr/~moreau_j/fr/compil/

ALAIN LECOMTE [2000], « Langages et communications », Université de grenoble : http://brassens.upmf-grenoble.fr/~alecomte/alaincours.html

M. E. LESK et E. SCHIMDT [1979], « Lex – A Lexical Analyzer Generator » : http://dinosaur.compilertools.net/lex/index.html

PHILIPPE MARQUET [1993], « Travaux Dirigés et Travaux Pratiques pour la Construction d'un Petit Compilateur de Pascal », Laboratoire de Recherche en Informatique de l'Université des Sciences et Technologies de Lille, révision de 1995 :

http://www.lifl.fr/~marquet/ens/pcp/

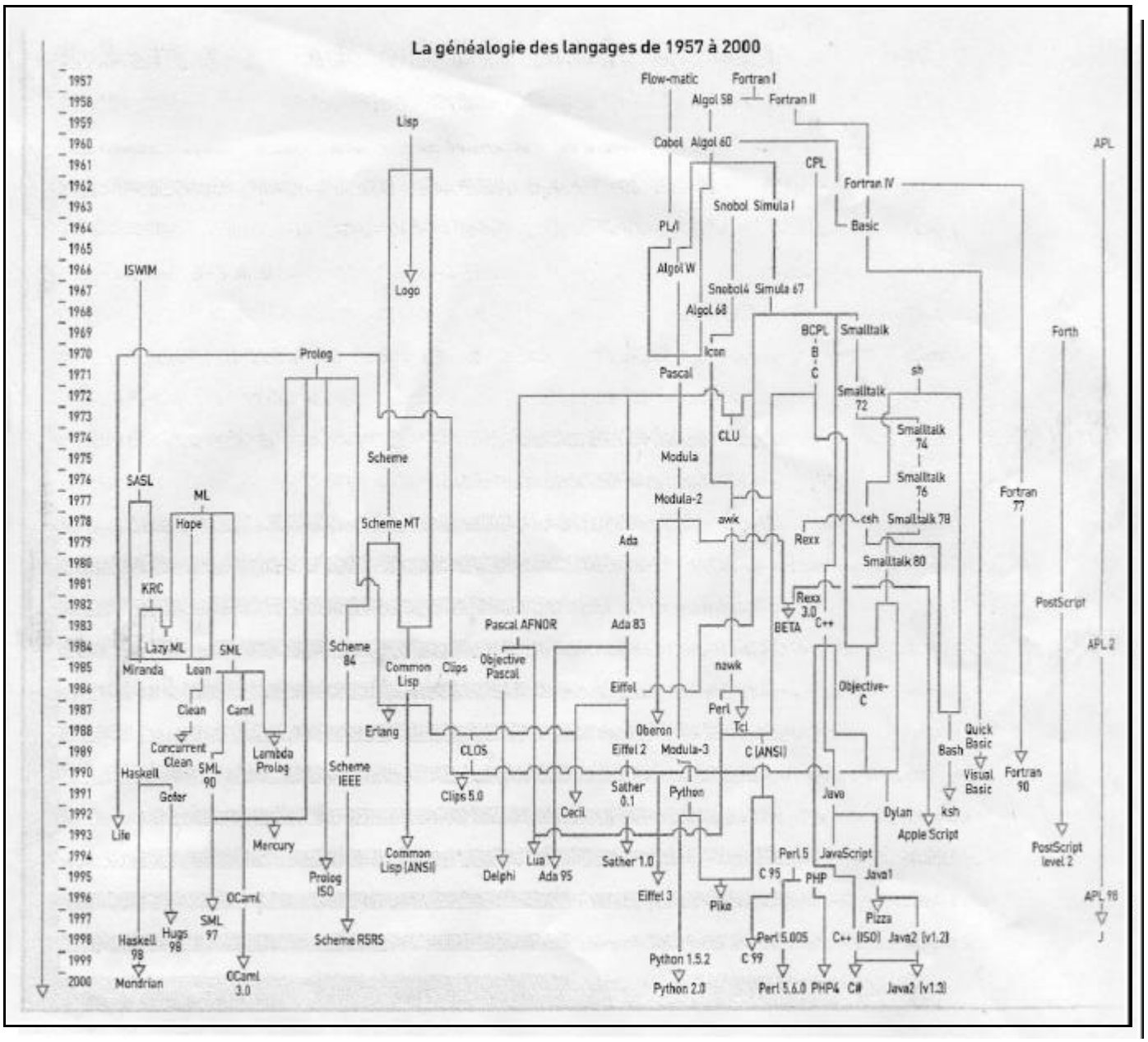
JEAN-PIERRE MICELI [2002], « Flex et Bison », Ecole d'ingénieur du canton de Vaud, Lausanne : http://www.eivd.ch/len/bison/flex_bis.htm

VERN PAXSON [1995], « Flex, Version 2.5 – A Fast Scanner Generator », Lawrence Berkeley Laboratory, Berkeley : http://dinosaur.compilertools.net/flex/index.html

P.D. TERRY [1996], « Compilers and Compiler Generators – An introduction with C++ », Université de Rhodes, révision de janvier 2000 : http://www.scifac.ru.ac.za/compilers/

ANNEXES

Annexe 1 : La généalogie des langages



Shéma extrait du 01 informatique de la semaine du 15 octobre 2001

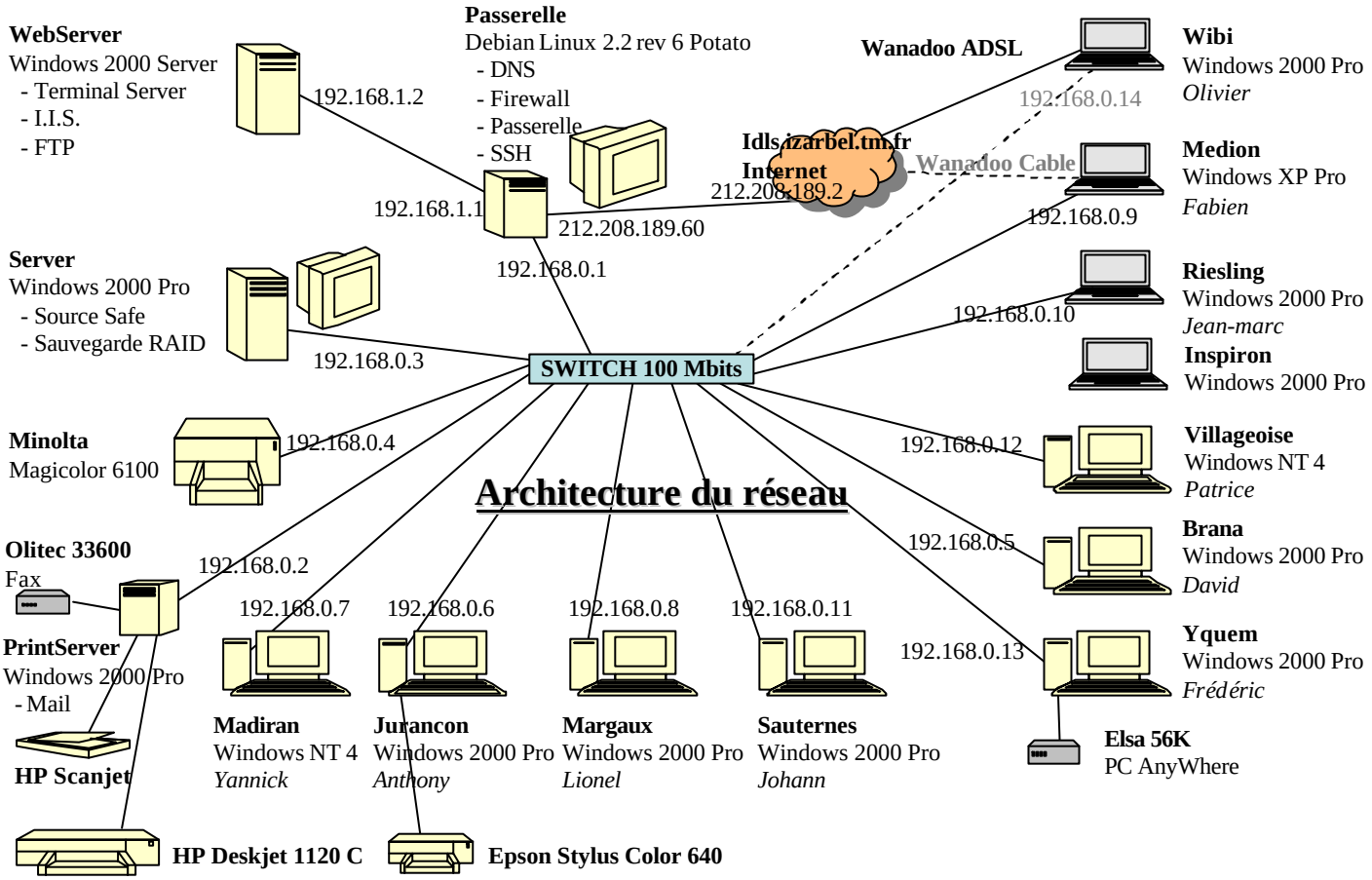
Annexe 2 : Les expressions régulières de Lex

2.2 Les expressions régulières

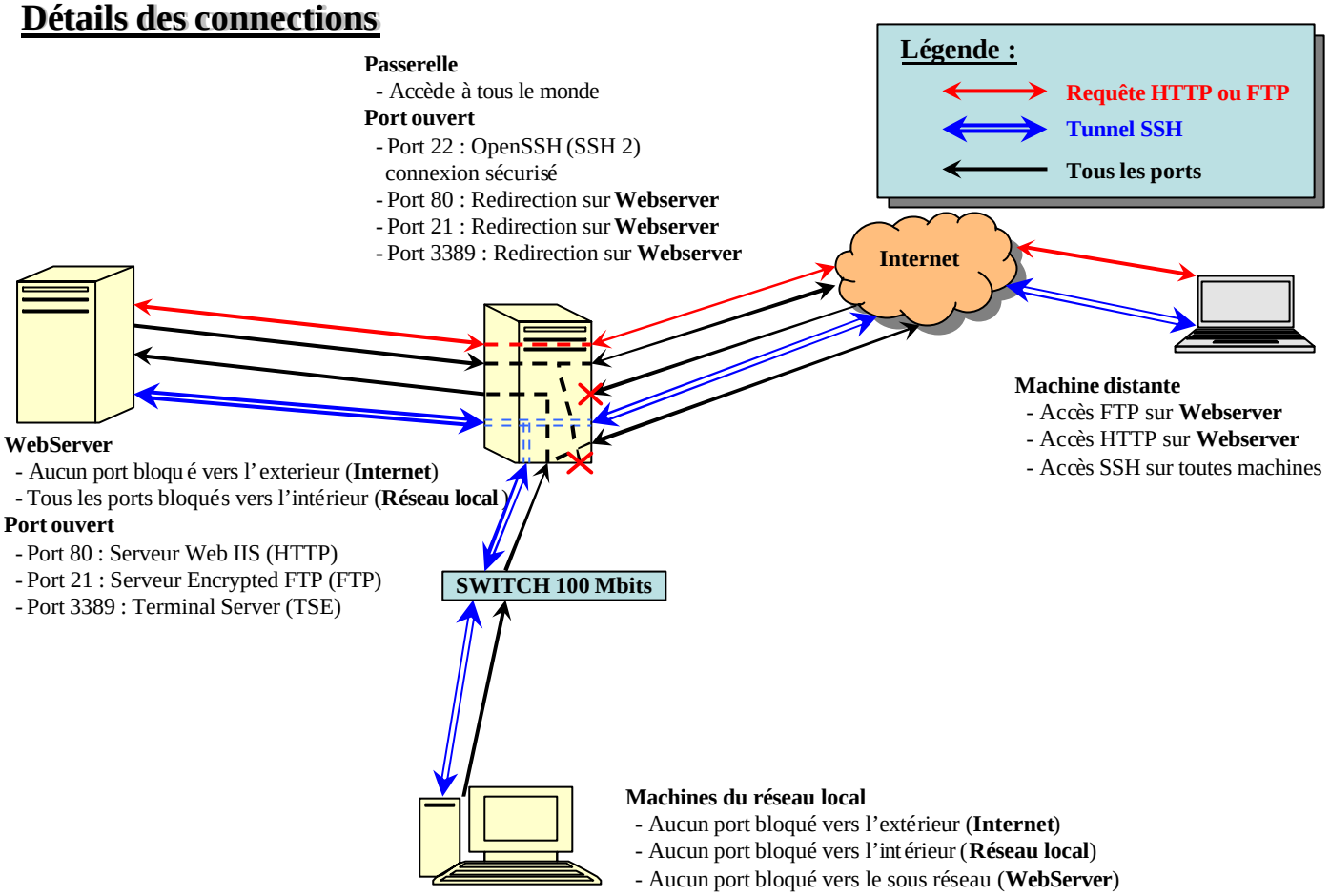
Symbole	Signification
x	Le caractere 'x'
.	N'importe quel caractere sauf \n
[xyz]	Soit x, soit y, soit z
[^bz]	Tous les caracteres, SAUF b et z
[a-z]	N'importe quel caractere entre a et z
[^a-z]	Tous les caracteres, SAUF ceux compris entre a et z
R*	Zero R ou plus, ou R est n'importe quelle expression reguliere
R+	Un R ou plus
R?	Zero ou un R (c'est-a-dire un R optionnel)
R{2,5}	Entre deux et cinq R
R{2,}	Deux R ou plus
R{2}	Exactement deux R
"[xyz\"foo"	La chaine '[xyz"foo'
{NOTION}	L'expansion de la notion NOTION definie plus haut
\X	Si X est un 'a', 'b', 'f', 'n', 'r', 't', ou 'v', represente l'interpretation ANSI-C de \X.
\0	Caractere ASCII 0
\123	Caractere ASCII dont le numero est 123 EN OCTAL
\x2A	Caractere ASCII en hexadecimal
RS	R suivi de S
R|S	R ou S
R/S	R, seulement s'il est suivi par S
^R	R, mais seulement en debut de ligne
R$	R, mais seulement en fin de ligne
<<EOF>>	Fin de fichier

Extrait du mini manuel Lex et Yacc d'Etienne Bernard

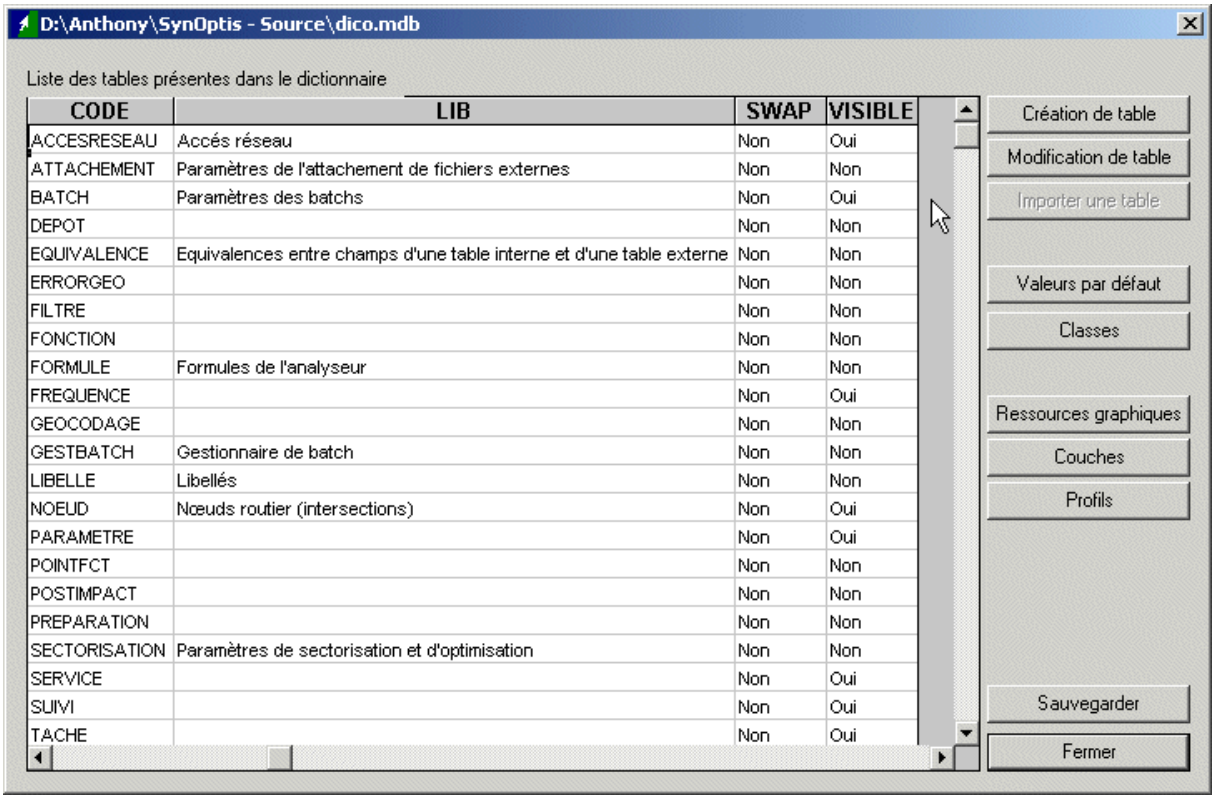
Annexe 3 : Le schéma du réseau de SynOptis



Détails des connections



Annexe 4 : L'éditeur de dictionnaire



CODE	LIB	SWAP	VISIBLE
ACCESRESEAU	Accés réseau	Non	Oui
ATTACHEMENT	Paramètres de l'attachement de fichiers externes	Non	Non
BATCH	Paramètres des batchs	Non	Oui
DEPOT		Non	Non
EQUIVALENCE	Equivalences entre champs d'une table interne et d'une table externe	Non	Non
ERRORGEO		Non	Non
FILTRE		Non	Non
FONCTION		Non	Non
FORMULE	Formules de l'analyseur	Non	Non
FREQUENCE		Non	Oui
GEOCODAGE		Non	Non
GESTBATCH	Gestionnaire de batch	Non	Non
LIBELLE	Libellés	Non	Non
NOEUD	Nœuds routier (intersections)	Non	Oui
PARAMETRE		Non	Oui
POINTFCT		Non	Non
POSTIMPACT		Non	Non
PREPARATION		Non	Non
SECTORISATION	Paramètres de sectorisation et d'optimisation	Non	Non
SERVICE		Non	Oui
SUIVI		Non	Oui
TACHE		Non	Oui