Institut National Polytechnique
de Lorraine

Centre de Recherche en
Informatique de Nancy

THESE

présentée pour l'obtention du grade de

Docteur de l'Institut National Polytechnique de Lorraine
Spécialité Informatique

par

Samuel CRUZ LARA SILVA

Ingénieur de l'Institut Technologique et des Etudes Supérieures de Monterrey
Unidad Estado de México. MEXIQUE

Sujet

GEODE : Un système pour la génération d'environnements de programmation intégrés

Thèse soutenue publiquement le 25 novembre 1988
devant la Commission d'Examen,
composée de MM.

Président et Rapporteur	.....	Claude PAIR
Rapporteur	.....	Bernard LANG
Examinateurs	.....	Jean Claude DERNIAME
		Norbert EBEL
		Jean Pierre FINANCE
		Michael GRIFFITHS

Institut National Polytechnique de Lorraine

Centre de Recherche en Informatique de Nancy

THESE

présentée pour l'obtention du grade de

Docteur de l'Institut National Polytechnique de Lorraine
Spécialité Informatique

par

Samuel CRUZ LARA SILVA

Ingénieur de l'Institut Technologique et des Etudes Supérieures de Monterrey
Unidad Estado de México. MEXIQUE

Sujet

GEODE : Un système pour la génération d'environnements de programmation intégrés

Thèse soutenue publiquement le 25 novembre 1988
devant la Commission d'Examen,
composée de MM.

Président et Rapporteur	.....	**Claude PAIR**
Rapporteur	.....	**Bernard LANG**
Examinateurs	.....	**Jean Claude DERNIAME**
		Norbert EBEL
		Jean Pierre FINANCE
		Michael GRIFFITHS

A la mémoire de mon grand père Roberto Silva Mendoza.

A ma famille.

A Yvette.

Remerciements.

Je voudrais tout d'abord remercier M. Claude Pair qui a bien voulu être rapporteur et qui me fait l'honneur de présider ce jury de thèse; sa contribution à promouvoir la recherche en informatique, à la mise en place du CRIN et son action dans les organismes gouvernementaux, le place sans doute comme l'un des plus importants éléments des milieux universitaires français. Malgré ses nombreuses responsabilités, il a toujours été dans la meilleure disposition pour m'aider et pour répondre à mes questions. Qu'il trouve dans ces lignes l'expression de ma profonde reconnaissance.

Ma dette est aussi considérable à l'égard de M. Bernard Lang de l'INRIA. Je le remercie très vivement d'avoir accepté d'être rapporteur de ce travail, et de l'effort qu'il a accepté d'y consacrer. Quiconque ayant travaillé de près ou de loin sur les environnements de programmation connaît l'importante contribution de M. Lang. Sa présence dans ce jury me fait un grand honneur.

Je dois aussi beaucoup à mon directeur de thèse M. Jean-Claude Derniame. Non seulement il m'a accepté dans son équipe "Génie Logiciel", mais il m'a appris ce qu'est la recherche dans ce domaine. Quelque fût le problème posé, en informatique ou dans d'autres domaines, il m'a toujours aidé avec beaucoup de sollicitude. Je le remercie profondément pour tout ce qu'il m'a apporté durant ces cinq années de thèse.

Je suis très reconnaissant à l'égard de M. Norbert Ebel de l'université de Lausanne, qu'en dépit de ses nombreuses occupations a bien voulu se déplacer jusqu'à Nancy pour faire partie de ce jury. Il travaille depuis plusieurs années dans le domaine des éditeurs structurés. Sa longue expérience dans ce domaine le désignait tout naturellement pour l'évaluation de cette thèse. Je le remercie vivement de l'intérêt qu'il a bien voulu porte à mon travail.

Je voudrais aussi exprimer ma profonde gratitude envers M. Jean-Pierre Finance, et à travers lui, en tant que Directeur du CRIN, remercier ce laboratoire et tous ses membres qui m'ont si bien accueilli. Que tous ceux qui m'ont aidé, de quelque façon que ce soit,

trouvent dans ces lignes l'expression de mes sentiments les meilleurs.

Je remercie aussi vivement M. Michael Griffiths de l'université d'Aix-Marseille qui a bien voulu faire partie de ce jury de thèse. Sa longue expérience dans le domaine de la compilation et des langages de programmation apportent un élément important pour l'évaluation de mon travail.

Je tiens à remercier tout particulièrement l'équipe "génie logiciel" du CRIN pour le cadre symphatique et fraternel dans lequel cette équipe a l'habitude de travailler. Aussi, je voudrais exprimer ma profonde gratitude envers mes collègues de bureau Christine Fay-Varnier et Dominique Colnet. Je les remercie de leur amitié, de leur esprit de camaraderie et du soutient inconditionnel qu'ils me témoignent depuis mon arrivée en France.

Je voudrais également remercier d'autres personnes qui ont bien voulu m'aider pour la rédaction, la correction et la réalisation de ce document, en particulier Gérôme Canals, Claude Godart, Khalid Benali, Nacer Boudjlida, Danielle Marchand, Isabelle Gnaedig et René Schott. Que ceux que j'oublie, inconsciemment, puissent m'en excuser.

Je ne saurais terminer sans exprimer toute ma reconnaissance à ma famille, qui malgré la distance et le temps, n'a jamais cessé de me soutenir et de m'encourager, même dans les moments les plus difficiles. Aussi, je voudrais remercier Yvette de son soutient et de son amour. Personne ne sait mieux qu'elle tout ce que je lui dois et tout ce qu'elle signifie pour moi.

Finalement, je remercie vivement le Conseil National de Science et Technologie du Mexique (CONACYT), ainsi que le Comité d'Etudes pour la Formation d'Ingénieurs (CEFI) de m'avoir offert la possibilité de réaliser ce travail.

SOMMAIRE.

INTRODUCTION GENERALE

Introduction.

Pendant très longtemps les personnes les plus réticentes à utiliser les ordinateurs comme support pour leur propre travail étaient, paradoxalement, les informaticiens! Mais les temps changent.

Les outils logiciels peuvent être définis comme des logiciels qui aident ou qui automatisent le processus de développement de logiciels, afin de réduire l'effort humain nécessaire. Les outils logiciels incluent bien sûr des outils associés au langage de programmation, comme un compilateur, mais aussi des outils que l'on peut appeler généraux, comme les programmes de tri ou ceux de traitement de textes.

Actuellement, il existe une énorme prolifération d'outils logiciels. Ainsi, on dispose sur la plupart des machines, de plusieurs éditeurs de texte, de plusieurs débogueurs, etc..., pour le traitement de différents langages de programmation ou le développement de divers projets.

Bien entendu, plus un outil logiciel est général, plus il est souple, mais moins il tient compte des contraintes ou des particularités propres au langage de programmation et/ou au projet à développer. En plus, ces types d'outils ne sont pas intégrés. Chacun représente les programmes d'une manière qui lui est propre et qui dans la plupart des cas est différente de celle des autres. En conséquence plusieurs outils ne peuvent être appliqués simultanément sur le programme lors de sa construction. Ne serait-il pas souhaitable de pouvoir exécuter un programme, corriger les erreurs éventuelles, puis continuer l'exécution à l'endroit où elle avait été interrompue ?

Un environnement de programmation intégré est constitué d'un ensemble d'outils qui travaillent simultanément sur le programme tout au long de sa construction, et qui collaborent et coopèrent en établissant des relations entre eux. De plus, les outils de l'environnement sont basés sur un langage donné, ce qui permet de tenir compte des caractéristiques propres de ce dernier. Par exemple, un éditeur syntaxique vérifie et corrige

la syntaxe des programmes au moment même où ils sont construits. Cette manière de concevoir un environnement de programmation permet, par exemple, d'exécuter et/ou déboguer un programme même s'il est incomplet. Sa correction peut donc être vérifiée au fur et à mesure de sa construction.

Le fait que les outils soient basés sur un langage déterminé implique qu'ils ne pourront être utilisés que pour le traitement de ce langage. Pour résoudre ce problème, deux voies sont possibles : la première consiste à construire des outils généraux, puis à paramétrer ces outils par une description du langage de programmation concerné. Malheureusement, cette voie est peu viable car l'utilisation de certains outils n'a de sens que dans le cadre de langages spécifiques (p.e. en ADA, on pourrait penser à la création d'outils traitant le parallélisme, tandis que dans PASCAL ce type d'outils serait inutile). Une deuxième voie consiste à générer des environnements de programmation adaptés aux besoins spécifiques du langage. Cette solution implique l'existence d'un générateur d'environnements de programmation, parametré d'un côté par une description du langage à traiter, et d'un autre côté par une description des outils constituant l'environnement de ce langage. Le mécanisme de génération d'environnements de programmation a été largement adopté à présent. Pour citer quelques exemples : CENTAUR [Clément 1986], ALOE [Medina-Mora 1981], CPS [Reps 1984].

Les environnements de programmation générés par ces systèmes possèdent les caractéristiques suivantes :

- Ils sont constitués d'un ensemble d'outils intégrés, c'est-à-dire, qu'ils travaillent simultanément sur le programme durant sa construction;
- Les outils sont basés sur une description du langage de programmation à traiter;
- Les environnements générés sont hautement interactifs.

Lorsqu'un environnement de programmation possède ces caractéristiques, on dira qu'il s'agit d'un environnement de programmation intégré (EPI).

Cependant, tous les générateurs d'EPI ne fonctionnent pas de la même manière et ils souffrent en général d'un ou plusieurs inconvénients :

- Tous les outils de l'environnement généré ne sont pas décrits en utilisant un même formalisme;
- La description du langage de programmation concerné est constituée de plusieurs parties qui ne sont pas toujours facilement maniables;
- Lorsqu'il s'agit d'utiliser l'environnement généré, l'utilisateur n'est pas en mesure de choisir les outils de l'environnement qui lui conviennent;
- Dans certains cas les outils générés sont beaucoup plus restrictifs que les outils "classiques".

Nous avons donc cherché à concevoir un générateur pouvant pallier ces inconvénients, tout en conservant les avantages et les caractéristiques des EPI. Ce générateur est le système GEODE* (Générateur d'EnvirOnnements DE programmation intégrés) [Cruzlara 1986], [Cruzlara 1987], [Canals 1988]. Lors de la conception de GEODE nos objectifs furent les suivants :

- Avoir un seul et unique formalisme pour la description du langage de programmation à traiter;
- Avoir un seul et unique formalisme pour la description des outils de l'EPI;
- Avoir la possibilité de définir séparément les outils de l'EPI tout en offrant un moyen de les intégrer ultérieurement;
- Permettre la construction des sous-environnements de programmation;
- Fournir au concepteur de l'environnement des outils de haut niveau.

Le système GEODE que nous présentons génère des EPI à l'aide d'une grammaire attribuée [Knuth 1968].

Des rappels et des généralités sur la notion d'environnement de programmation sont exposés dans le premier chapitre. Nous y présentons brièvement les concepts de base ainsi que les motivations et les objectifs de la construction du système GEODE.

* Le premier acronyme utilisé fût GEPOL (Générateur d'Environnements de Programmation Orientés vers le traitement d'un Langage de programmation particulier.

Le deuxième chapitre fait une présentation générale, mais précise du système GEODE. Nous exposons l'architecture du système en faisant la distinction entre les niveaux de conception et d'utilisation d'un environnement de programmation.

Une introduction aux grammaires attribuées est effectuée dans le troisième chapitre. Plusieurs exemples y sont étudiés afin de pouvoir mieux appréhender, par la suite, l'utilisation de ces grammaires dans le système GEODE.

Plusieurs outils sont construits à l'aide de grammaires attribuées dans GEODE. Le quatrième chapitre en fait une présentation exhaustive. L'aspect évaluation d'attributs et largement étudié et l'algorithme incrémental que nous avons implanté est analysé.

Le cinquième chapitre concerne l'implantation de GEODE. Nous présentons une introduction à la programmation par objets, et nous montrons comment utiliser ce style de programmation pour représenter, d'un côté, les grammaires attribuées et les programmes, et d'un autre côté, pour la réalisation de l'évaluation d'attributs.

Le sixième chapitre est une étude comparative entre GEODE et d'autres systèmes similaires. L'accent est mis sur les différences d'utilisation, leurs avantages et/ou leurs inconvénients.

Enfin, nous présentons les conclusions de nos travaux ainsi que les voies de recherche qui pourraient encore être explorées dans le domaine de la génération d'environnements de programmation intégrés.

Introducción.

Durante mucho tiempo las personas que más dudaron en integrar la computadora a su propio trabajo fueron, paradójicamente, aquellas que trabajaban en el medio ambiente de la informática. Pero, ¡los tiempos cambian!

Las herramientas de software pueden definirse como el conjunto de programas que ayudan y/o que automatizan el proceso de construcción de programas, con el objeto de reducir el esfuerzo humano requerido. Estas herramientas de software incluyen, obviamente, programas dependientes de los lenguajes de programación, como los compiladores, pero también herramientas más generales como lo serían los programas de clasificación o de tratamiento de textos.

Actualmente hay una enorme proliferación de herramientas de software. La mayor parte de las computadoras de hoy en día disponen de varios tipos de editores de textos, de verificadores, etc ... para el tratamiento de varios lenguajes de programación o de varios proyectos.

Obviamente, una herramienta de software más general es más flexible, pero ésta no toma en cuenta las restricciones o las características propias del lenguaje de programación y/o del proyecto en cuestión. Además, este tipo de herramientas no están integradas, lo que significa que cada herramienta representa internamente los programas a su manera y que en la mayor parte de los casos estas representaciones internas difieren según la herramienta utilizada. El resultado es que las diversas herramientas no pueden ser utilizadas simultaneamente durante la construcción de un programa. ¿No sería deseable que se pudiera ejecutar un programa, corregir los errores eventuales y despúes, continuar la ejecución en el lugar en la que ésta fué interrumpida?

Un entorno de programación integrado está constituído por un conjunto de herramientas que no trabajan independientemente : éstas trabajan simultaneamente durante la construcción de un programa y colaboran y cooperan estableciendo relaciones entre ellas.

Además, las herramientas del entorno están basadas en un lenguaje determinado, lo cual nos permite disponer de herramientas que toman en cuenta las características propias de ese lenguaje. Por ejemplo, un editor syntáctico verifica y corrige la syntáxis de los programas al mismo tiempo que el programador los construye. Esta manera de diseñar un entorno de programación nos permite, por ejemplo, de ejecutar y/o verificar un programa que no ha sido totalmente terminado. El usuario puede entonces verificar la corrección de su programa mientras lo esta construyendo.

Ahora bien, el hecho de que las herramientas estén basadas en un lenguaje de programación implica, necesariamente, que éstas no podrán ser usadas más que para el tratamiento de ese lenguaje. Para resolver este problema había dos posibilidades : la primera consistía en construir herramientas generales y parametrarlas despúes con la descripción del lenguaje de programación en cuestión. Desgraciadamente, esta solution es poco viable, pues el uso de ciertas herramientas no tiene sentido más que en el contexto del tratamiento de un lenguaje determinado (p.e. en ADA podría pensarse en la construcción de herramientas para tratar el paralelismo, mientras que en PASCAL este tipo de herramientas sería completamente inútil). Una segunda solución consistía en generar entornos de programación adaptados a las necesidades específicas de un lenguaje. Esta solución implica la existencia de un generador de entornos de programación que sería parametrado con une descripción del lenguaje en cuestión y con una descripción de las herramientas del entorno de programación que será generado. El mecanismo de generación de entornos de programación ha sido ampliamente aceptado hoy en día. Por citar algunos ejemplos : CENTAUR [Clément 1986], ALOE [Medina-Mora 1981], CPS [Reps 1984].

Los entornos de programación generados por estos sistemas poseen las siguientes características :

- Los entornos están constituídos por un conjunto de herramientas integradas que pueden trabajar simultaneamente durante la construcción de un programa;
- Las herramientas están basadas en una descripción del lenguaje de programación en cuestión;
- Los entornos generados son altamente interactivos.

Cuando un entorno de programación posee estas características, diremos que se trata de un entorno de programación integrado (EPI).

No obstante, los generadores de EPI no funcionan de la misma manera y adolecen en general de una o varias desventajas :

- No todas las herramientas del entorno son definidas por el mismo formalismo;
- La descripción del lenguaje de programación correspondiente está formada por diversas sub-descripciones que no son facilmente manipulables;
- El usuario del entorno de programación no puede decidir cuales son las herramientas que desea utilizar;
- En algunos casos las herramientas del entorno de programación son mucho más restrictivas, en cuanto a su uso se refiere, que las herramientas "clásicas".

Nuestro objetivo es de disponer de un generador que pueda conservar todas las ventajas de los otros generadores, pero sin tener sus inconvenientes. Este sistema es GEODE* (**G**enerador de **Ent**O**r**nos **DE** programación integrados) [Cruzlara 1986], [Cruzlara 1987], [Canals 1988]. Los objetivos de GEODE son los siguientes :

- Disponer de un solo y único formalismo para describir el lenguaje de programación utilizado;
- Disponer de un solo y único formalismo para describir las herramientas que deberá contener el entorno generado;
- Ofrecer la posibilidad de definir independientemente las herramientas del EPI pero también de un medio que nos permita reunirlas posteriormente;
- Ofrecer al diseñador del entorno herramientas de alto nivel.

Este documento presenta el sistema GEODE que es un generador de entornos de programación integrados, a través de una gramática atribuída [Knuth 1968].

* El primer acrónimo utilizado fué GEPOL (Generador de Entornos de Programación Orientados hacia el tratamiento de un Lenguaje de programación determinado.

Las generalidades y conceptos de base de los entornos de programación, así como las motivaciones y objetivos de la construcción del sistema GEODE, son presentados en el primer capítulo.

El segundo capítulo es una presentación general, pero precisa del sistema GEODE. La arquitectura del sistema es presentada asi como la diferencia que existe entre los niveles de diseño y utilización del entorno de programación.

Una introducción a las gramáticas atribuídas es realizada en el tercer capítulo. Diversos ejemplos son presentados, con el objeto de facilitar la comprensión en lo que se refiere al uso de este tipo de gramáticas en el sistema GEODE.

Diversas herramientas son construídas por medio de las gramáticas atribuídas. El cuarto capítulo constituye une presentación exhaustiva. El concepto de evaluación de attributos es ampliamente estudiado y el algoritmo de evaluación incremental que desarrollamos es analyzado.

El quinto capítulo se refiere a la implantación de GEODE. Una introducción a la programación orientada objeto es presentada, así como una descripción de la manera en la que usamos este tipo de programación para implantar, por un lado, las gramáticas atribuídas y los programas, y por otro lado, la ejecución de la evaluación de atributos.

En el sexto capítulo realizamos un estudio comparativo entre GEODE y otros sistemas similares. Nuestra atención es concentrada en analyzar las ventajas y desventajas de la utilización de GEODE, con respecto a la utilización de esos sistemas.

Finalmente, presentamos la conclusión de nuestro trabajo y los sectores en los que consideramos que se pueden realizar investigaciones futuras dentro del área de la generación de entornos de programación integrados.

Introduction.

During many years there were some kind of people who didn't want to incorporate the computer into their work. These people were, paradoxically, the computer world people. Fortunately, times change!

Software tools may be defined as programming tools which automatize software development in order to reduce the necessary human effort. Software tools may be language-dependent tools as compilers, but also general-purpose tools as sorting or word processing programs.

At the present time, there is an enormous proliferation of software tools. So we actually have on a single computer, text editors, debuggers, ... etc. These kinds of tools are generally used for processing several programming languages or for developing several projects.

Obviously, a general tool is quite flexible but it has no idea about the programming language it is working with or about the project that is being developed. In addition, these tools are not integrated, that means that each software tool represents programs in a particular way, and in most cases this way of representing programs is not compatible with other tools. The result is that these software tools cannot be applied simultaneously during program's construction. One would want to execute a program, to correct errors and then to continue execution at the place where it has been interrupted.

An integrated programming environment is formed by a set of software tools that do not work independently : they work simultaneously during program's construction and they collaborate by establishing several relations between them. In addition, these software tools are language-based tools, so they have knowledge about the programming language they are working with. For example, syntactic editors verify and correct the syntax of programs while we are constructing them. Design programming environments in this way allow us, for example, to execute and/or to debug a program even if it has not been finished

yet. Thus, we can verify program's correcness while we are constructing it.

Nevertheless, working with language-based software tools implies that we can use these tools only for processing one language. To solve this problem there are two possible solutions : in the first one we built general-purpose tools and then we parametrize them by a language description. Unfortunately, this solution is not a practical one because using a tool is a language-dependent matter (i.e. in ADA we may build software tools for dealing with parallelism, while for PASCAL this kind of software tools will be meaningless). The second solution is to generate programming environments according to the programming language requirements. This solution implies the existence of an environment generator that will be parametrized, on the one hand, by the language's description and, on the other hand, by the environment tools' descriptions. At the present time, the environment generation approach has been fully adopted. As an example : CENTAUR [Clément 1986], ALOE [Medina-Mora 1981], CPS [Reps 1984].

Programming environments generated by these systems have the following particularities :

- programming environments are a set of integrated software tools, so they work simultaneously during program's construction;
- the environment's tools are language-based tools;
- generated environments are highly interactive.

A programming environment having these particularities will be called an integrated programming environment (IPE).

Even if all IPE generators do not work in the same way, the generated environments frequently have several drawbacks :

- there are several formalisms to describe the environment's tools;
- the programming language description is constitued by several sub-descriptions that are not always easy to handle;
- when using a generated environment, users have no means to decide which tools

they want to work with;
- in some cases, integrated environment's tools are quite more restrictives than "classical" environment's tools.

We have then decided to build an integrated environment generator that may resolve these drawbacks but keeping all advantages of IPEs. This generator is the GEODE* system (Générateur d'**Envir**Onnements **DE** programmation intégrés) [Cruzlara 1986], [Cruzlara 1987], [Canals 1988]. GEODE's main purposes are the following :

- to have a single formalism allowing a description of the programming language;
- to have a single formalism allowing to describe all the environment's tools;
- to have the possibility to describe all these tools individually, but providing a way to join them together at a later time;
- to create programming subenvironments;
- to provide high level tools to the programming environment's designer.

This dissertation introduces the GEODE system. GEODE is a generator of integrated programming environments from an attribute grammar [Knuth 1968].

The first chapter is an introduction to programming environments. We briefly present programming environments' main concepts and we describe ours motivations and objectifs in building the GEODE system.

The second chapter is a general but explicit presentation of the GEODE system. We introduce the GEODE's architecture and we establish the difference between user's and designer's levels.

The third chapter introduce attribute grammars. Several examples are analyzed in order to understand the attribute grammar mechanism which will be extensively used in the GEODE system.

* The first acronyme used was GEPOL (Générateur d'Environnements de Programmation Orientés vers un Langage de programmation particulier).

The fourth chapter is an exhaustive presentation describing how to use the attribute grammar mechanism in GEODE. Several software tools, as syntactic editors, are constructed using this mechanism. We also discuss about attribute evaluation and we introduce our incremental algorithm whose complexity is analyzed.

The fifth chapter presents some aspects concerning GEODE's implementation. We introduce object-oriented programming and we show how to use it, in the one hand, to represent attribute grammars and programs, and in the other hand to perform attribute evaluation.

The sixth chapter is a comparative study between the GEODE system and other similar systems. We mainly present the advantages and drawbacks of using GEODE in comparison with using the other systems.

Finally, a conclusion is presented. We make some final remarks about our work and we discuss about several future research fields in generating and using programming environments.

CHAPITRE I

Chapitre I.

LES ENVIRONNEMENTS DE PROGRAMMATION : GENERALITES ET RAPPELS.

1. LA NOTION D'ENVIRONNEMENT DE PROGRAMMATION.

Pendant très longtemps les personnes les plus réticentes à utiliser les ordinateurs en tant qu'outil de support pour leur propre travail était les informaticiens! Mais, heureusement les temps changent et actuellement on peut dire que l'une des avances les plus significatives en informatique est sans doute la création d'outils qui assistent les programmeurs lors de la construction des programmes [Barstow 1984].

Il est évident que les informaticiens ont toujours utilisé, malgré tout, des outils logiciels qui les aident dans leur travail de construction de logiciels, tels que les éditeurs et les compilateurs. Ensuite, il y a eu l'apparition de nouveaux outils tels que les interprètes et les débogueurs, mais ce n'est que maintenant que l'on construit des véritables environnements pour aider les informaticiens dans leur tâche difficile de construire un programme vraiment adapté aux besoins des utilisateurs.

Cette section est consacrée à l'étude des environnements de programmation au sens le plus général du terme. Nous définissons un environnement de programmation comme un ensemble d'outils aidant le programmeur dans sa tâche de construire des programmes. Afin d'étudier ces environnements nous allons établir la classification suivante :

- Boîtes à outils;
- Environnements de programmation intégrés;
- Ateliers de génie logiciel.

Cette classification est quelque peu arbitraire d'autant plus que la division entre les

différentes catégories n'est pas très nette. Cependant, nous considérons qu'elle convient parfaitement au cadre dans lequel nous avons placé nos recherches. Les sections suivantes sont donc dédiées à l'étude des boîtes à outils, des environnements de programmation intégrés et des ateliers de génie logiciel. Nous incluons dans la discussion des commentaires au sujet de l'avenir de ce type d'outils.

1.1 LES BOITES A OUTILS.

Chronologiquement parlant, les boîtes à outils constituent les premiers ensembles d'outils ayant pour but d'assister les constructeurs de programmes. Une boîte à outils est constituée de plusieurs outils tels que les éditeurs de textes, les compilateurs, les débogueurs, les éditeurs de liens, mais aussi, dans les boîtes à outils les plus évoluées, des outils d'aide à la spécification, à la conception et même à la maintenance.

La caractéristique fondamentale des boîtes à outils est leur *généralité*, c'est-à-dire que les outils les constituant peuvent traiter aussi bien des programmes que des spécifications ou des structures modulaires. Par exemple, sous UNIX [Kernighan 1981] un éditeur de textes peut être utilisé pour construire un programme mais aussi pour la création de "makefiles" qui pourraient éventuellement aider à la maintenance.

Une autre caractéristique très importante des boîtes à outils est leur *évolutivité*. En effet, une boîte à outils est constitué de plusieurs modules que nous pourrions appeler des modules de base. D'autres outils peuvent donc être construits à partir de ces modules de base. Les premiers peuvent à leur tour être utilisés comme des modules de base et ainsi de suite. Ceci peut encore être illustré par le "shell" du système UNIX.

Ces caractéristiques doivent certainement être vues comme des avantages mais elles peuvent aussi se transformer en des inconvénients. Par exemple, reprenons le cas d'un éditeur de textes : il est clair que dans une boîte à outils un éditeur de textes peut aussi bien construire un programme COBOL, qu'un programme C ou qu'un programme LISP.Mais il est aussi clair que ces trois langages de programmation ne possèdent pas du tout les mêmes caractéristiques, ni syntaxiquement ni sémantiquement parlant. Combien de temps nous font

perdre des erreurs qui concernent uniquement des aspects syntaxiques ! Ne serait-il souhaitable que les éditeurs de textes aient une connaissance plus approfondie du langage qu'ils manipulent ?

Un autre inconvénient, lié certes au premier, est la *"séquentialité"*. Par exemple, dans une boîte à outils lorsque l'on veut produire un programme exécutable à partir d'un texte source, il faut appliquer plusieurs outils dans un ordre strictement séquentiel : il faut d'abord le construire avec l'éditeur de textes, ensuite il faut le compiler, le déboguer, faire l'édition des liens et c'est seulement à ce moment-là que l'on sera en mesure d'obtenir un programme exécutable. Evidement, si il y a une erreur qui n'est détectée qu'au moment de l'exécution (p.e. des erreurs liées à la sémantique dynamique du langage) il faudra réappliquer toutes les outils en séquence, c'est-à-dire qu'il faudra modifier le programme avec l'éditeur de textes, le recompiler et refaire l'édition de liens. Ces inconvénients viennent principalement du fait que les outils n'ont aucune connaissance sur les objets qu'ils manipulent et que très souvent chaque outil représente internement ces objets d'une manière différente aux autres.

Un dernier inconvénient des boîtes à outils est leur *"manque de cohérence"*. Cet inconvénient signifie que chaque outil de la boîte représente les programmes différemment des autres. Par exemple, un éditeur de textes représente le programme comme un simple texte, tandis qu'un compilateur le représentera sûrement par un arbre abstrait, ou bien, par un arbre de dérivation. Ce manque de cohérence implique que si une erreur est détectée par le compilateur il faudra la corriger avec l'éditeur syntaxique, mais aussi il faudra réappliquer le compilateur à toutes les parties du programme y compris celles qui n'ont pas été touchées par la modification. Ne serait-il pas souhaitable que le compilateur n'ait à recompiler que la partie du programme où la modification a été effectuée ?

Il est clair que les boîtes à outils possèdent des avantages importants mais aussi des inconvénients qui peuvent s'avérer très contraignants. Intuitivement, il serait souhaitable que l'on puisse disposer des avantages sans avoir pour au tant les inconvénients. Dans cette optique est né le concept d'environnement de programmation intégré que nous analyserons dans la section suivante.

1.2 LES ENVIRONNEMENTS DE PROGRAMMATION INTEGRES.

Dans la section précédente nous avons analysé le concept de boîte à outils et nous en avons cité les avantages et inconvénients. Dans cette section nous présentons ,brièvement, une nouvelle manière de concevoir des environnements de programmation, qui prétend avoir les avantages de boîtes à outils, sans en avoir les inconvénients. Ces environnements sont appelés environnements de programmation intégrés, ou environnements de programmation interactifs.

Un environnement de programmation intégré (EPI) est constitué d'un ensemble d'outils qui ne travaillent pas de manière indépendante : ils travaillent simultanément sur le programme tout au long de sa construction et ils collaborent et coopèrent établissant des relations entre eux. De plus, les outils de l'environnement sont basés sur un langage donné, ce qui nous permet d'avoir des outils tenant compte des caractéristiques propres d'un langage. Par exemple, un éditeur syntaxique vérifie et corrige la syntaxe des programmes au moment même où on les construit. Cette manière de concevoir un environnement de programmation nous permet, par exemple, d'exécuter et/ou déboguer un programme même s'il est incomplet. Nous pouvons ainsi vérifier la correction d'un programme au fur et à mesure de sa construction.

Or, le fait que les outils soient basés sur un langage donné implique nécessairement que ces outils ne pourront être utilisés que dans le cadre du traitement de ce langage, ce qui est un inconvénient par rapport aux boîtes à outils : dans un environnement de programmation intégré un éditeur nous permettant de construire un programme PASCAL ne pourra pas être utilisé pour construire un programme ADA. Pour éliminer cet inconvénient, deux voies étaient possibles : la première consistait a construire des outils généraux et puis de paramétrer ces outils par une description du langage de programmation concerné. Malheureusement, cette voie est peu viable car l'utilisation de certains outils n'a de sens que dans le cadre de certains langages (p.e. en ADA, on pourrait penser à la création d'outils traitant le parallélisme, tandis que dans PASCAL ce type d'outils serait inutile). Une deuxième voie consistait alors à générer des environnements de programmation adaptés aux besoins spécifiques d'un langage. Cette solution implique donc l'existence d'un générateur

d'environnements de programmation parametré d'un côté, par une description du langage à traiter, et d'un autre côté par une description des outils constituant l'environnement de ce langage. Le mécanisme de génération d'environnements de programmation a été largement adopté à présent. Pour citer quelques exemples : CENTAUR [Clément 1986], ALOE [Medina-Mora 1981], CPS [Reps 1983] et GEODE.

Les EPI doivent posséder les caractéristiques suivantes [Barstow 1984] :

- Ils doivent offrir un important ensemble d'outils orientés vers le traitement d'un langage de programmation particulier;
- Ils doivent tenir compte de la structure intrinsèque des programmes : un programme n'est plus considéré comme une simple suite de caractères;
- Ils sont hautement interactifs;
- Ils doivent être intégrés;
- Etant donné qu'ils peuvent demander aux utilisateurs un effort supplémentaire (p.e. décrire les motivations qui ont impliqué une décision prise), ces systèmes doivent faire sentir aux utilisateurs que leurs efforts seront récompensés à la fin.

Tous les systèmes indiqués ci-dessus ne génèrent pas des EPI ayant toutes ces caractéristiques, c'est pourquoi nous allons nous baser, dans un premier temps, sur un système idéal capable de générer ce type d'environnements. Ce système, qui n'a jamais été construit, est une généralisation du système A décrit par Terry Winograd dans l'article "Breaking the Complexity Barrier (Again)" [Winograd 1973].

Pour s'occuper de la complexité des systèmes générateurs d'EPI, Winograd propose deux solutions :

- La première propose d'utiliser des mécanismes permettant de représenter des connaissances de façon uniforme et simple. Les grammaires à contexte libre et le calcul de prédicats en sont les meilleurs représentants;

- La deuxième proposition est encore quelque peu utopique. Il s'agit de de faire en

sorte que les programmes puissent apprendre par eux mêmes. Le rêve est que l'on puisse commencer avec une structure minimale à laquelle on fournira un ensemble de données et que le programme se débrouille ensuite pour construire ses propres complexités.

Quelque soit le solution adoptée, les EPI générés par A doivent offrir aux utilisateurs les facilités suivantes :

- Détection d'erreurs;
- Réponses aux questions concernant le programme et/ou l'activité de programmation;
- Libération des tâches banales;
- Débogage.

Le système A doit donc combiner la puissance d'un compilateur, d'un interprète, d'un éditeur syntaxique, d'un système de débogage, d'un système de documentation et d'un "problem-solver". En outre, pour pouvoir interagir convenablement ces outils doivent constituer un système intégré et non pas une collection d'outils indépendants. L'utilisateur doit pouvoir mélanger les activités d'écrire un programme, de l'éditer, de l'exécuter (même par morceaux), de répondre aux questions le concernant, de stipuler des informations nouvelles, d'accroître le stockage des concepts abstraits le décrivant et de le déboguer. Toutes ces activités devant être réalisées sans avoir à faire des aller-retour entre les différents outils, où à lire et écrire des fichiers, ou le reste de la mécanique généralement associée au traitement des programmes. Enfin, tous ces outils doivent pouvoir communiquer entre eux : l'information du modèle est vitale au compilateur et au débogueur; l'éditeur doit pouvoir reconnaître le moment où les changements entraînent des modifications sur le modèle; le débogueur doit connaître très précisément les représentations utilisées par le compilateur et l'interprète; et ainsi de suite à travers tout le système.

Concentrons nous maintenant sur la façon dont les EPI générés par A doivent être manipulés. Considérons tout d'abord la forme des programmes :

Un programme doit contenir, mises à part les instructions qui le composent, des

"indications". Il doit en plus être représenté par des structures des données contextuelles.

Un aspect fondamental de la philosophie de A est que le système doit non seulement manipuler un programme, mais il doit savoir ce que le programme est censé faire. Ceci n'est pas l'approche de la vérification de programmes : le programmeur n'a pas à donner des ensembles d'assertions devant être vérifiées à l'entrée ou à la sortie d'une pièce de code. Au lieu de cela, le programmeur doit être capable de stipuler des informations à des niveaux d'abstraction différents. Il doit pouvoir attacher une "indication" à n'importe quelle pièce de code : un appel de fonction, une procédure, un bloc ou même l'accès à une simple variable.

Les "indications" proposées par Winograd sont :

- Conditions (Pré-post conditions);
- Assertions;
- Descriptions abstraites indiquant ce qu'une pièce de code est censée faire;
- Indications en langage naturel.

En plus, les programmes construits en utilisant un EPI généré par A doivent être structurés de manière contextuelle, c'est-à-dire que la structure du programme ne sera pas perçue de la même manière par tous les outils. Dans l'hypothèse d'un avenir prometteur, le programmeur écrira seulement les "indications" et l'environnement construira le programme.

En ce qui concerne l'exécution des programmes les EPI générés doivent contourner la loi de la complexité de Parkinson : Plus la puissance de machines augmente, plus le désir de construire de programmes plus complexes et plus grands augmente, et à chaque niveau il y a des nouvelles limitations imposées par la capacité de traitement.

Pour contourner donc cette loi de la complexité, Winograd propose ce qu'il appelle la "redondance multi-niveaux". L'idée de base est que l'on doit pas être obligé de trouver un compromis entre efficience et souplesse. On peut disposer des plusieurs ressources pour le même "job". Chaque ressource ayant une priorité qui lui est propre. Le système s'occuperait alors d'orchestrer les différentes ressources utilisées par les différents "jobs". Un exemple de

la "redondance multi-niveaux" est la coexistence de programmes compilés et interprétés dans plusieurs environnements de programmation construits au tour de LISP, Le_Lisp15.2 [Chailloux 1986], INTERLISP [Teitelman 1981] et MACLISP [Moon 1974].

Les environnements générés par A doivent avoir au moins trois modes différents d'exécuter les programmes : le mode compilé, le mode interprété et le mode intelligent. Les deux premiers modes correspondent au modes compilé et interprété que l'on peut trouver déjà sur plusieurs environnements de programmation (c.f. environnements LISP ci-dessus). Néanmoins, pour que le compilateur puisse générer du code vraiment efficient, il faudra utiliser un "problem-solver" capable d'interpréter les "indications" décrites ci-dessus de manière à disposer d'une compilation intelligente.

Le troisième mode, le mode intelligent, est une sorte d'interprète intelligent.Pendant qu'il interprète une pièce de code, il tient compte des "indications" pour vérifier que toutes les conditions stipulées sont respectées, il possède des structures de contrôle lui permettant de connaître l'état du déroulement du programme et il interagit avec tous les mécanismes de débogage de l'environnement. L'exécution d'un programme peut se dérouler en utilisant ces trois modes. Tout doit être synchronisé et contrôlé automatiquement par l'environnement.

Finalement, Winograd propose d'avoir une sorte de système de gestion de fichiers s'occupant de tous les programmes qui font partie d'une application ou d'un projet quelconque. Les programmes constitueraient donc une "structure de bloc", qui n'est pas une simple structure arborescente : un programme ou sous-bloc peut se trouver dans l'intersection de plusieurs blocs plus grands et les division des blocs peuvent être faites de nombreuses manières (p.e. on peut diviser les blocs d'après les types des structures de données qu'ils manipulent, d'après le moment où ils sont appelés lors de l'exécution d'une tâche donnée ou même selon le nom du programmeur qui les a construits). Nous reparlerons de cette structuration avec quelque détail dans la section consacrée aux ateliers de génie logiciel.

Si nous résumons les idées de Winograd, les environnements de programmation intégrés doivent posséder les caractéristiques suivantes :

* Caractéristiques générales des EPI :
  - Outils basés sur langage de programmation à traiter;
  - Outils intégrés;
  - Programmes ≢ Suite de caractères; Programmes ≡ Structures de données;
  - L'environnement doit savoir ce qu'il fait;
  - Haute Interactivité;
  - Convivialité.

* Pour s'occuper de la complexité inhérente aux EPI :
  - Grammaires à contexte libre et calcul de prédicats;
  - Intelligence Artificielle.

* Buts des EPI :
  - Détection d'erreurs;
  - Réponse aux questions concernant le programme et/ou l'activité de programmation;
  - Libérer l'utilisateur des tâches banales;
  - Débogage.

* Pour parvenir à ces buts en EPI doit avoir :
  - Compilateur Incrémental;
  - Interprète Incrémental;
  - Editeur Syntaxique;
  - Système de documentation;
  - "Problem-Solver".

* Les programmes traités par les EPI doivent :
  - Contenir des "indications" (i.e. conditions, assertions, descriptions abstraites sur ce qui est fait, indications en langage naturel);
  - Etre structurés de manière contextuelle;
  - Etre exécutés dans un environnement de redondance multi-niveaux;
  - Etre réunis dans une structure de bloc.

Enfin, en ce qui concerne les systèmes générateurs d'environnements de programmation intégrés :

* Implantation du système :
  - Le système haute doit être hautement interactif;
  - Le système hôte doit offrir des primitives de haut niveau :
    + Manipulation des structures de données;
    + Manipulations système (i.e. interruptions, multi-programmation, multi-traitement, ...).

* Construction du système :
  - "Bootstrapping";
  - Génie Logiciel;
  - Intelligence Artificielle.

En une seule phrase Winograd dit que dans l'avenir les environnements de programmation intégrés devront être des systèmes basés sur la connaissance ("based-knowledge systems") et donc essentiellement sur les techniques de l'intelligence artificielle. Plusieurs systèmes qui essayent d'intégrer l'intelligence artificielle aux environnements de programmation intégrés commencent à apparaître, citons par exemple [Waters 1982] et [Kant 1981].

Sans vouloir être trop pessimistes, nous croyons que le système décrit par Winograd ne sera réalisable que d'ici plusieurs années. Nous nous concentrerons donc sur des environnements de programmation bien plus modestes que ceux générés par le système A et qui font partie de la première solution proposée par Winograd pour traiter la complexité des systèmes générateurs d'EPI, c'est-à-dire des systèmes qui sont basés sur les grammaires à contexte libre et, dans certains cas, sur le calcul de prédicats (2) :

- Le système MENTOR [Donzeau-Gouge 1980]

- Le système CENTAUR [Clément 1986]
- Le système ALOE [Medina-Mora 1981]
- Le système CPS [Reps 1981]
- Le système CEPAGE [Meyer 1984]

1.3 LES ATELIERS DE GENIE LOGICIEL.

Avant de présenter les ateliers de génie logiciel, il nous semble pertinent de faire le point sur les boîtes à outils et les environnements de programmation intégrés. Les boîtes à outils nous proposent de nombreux outils pour le traitement de programmes, mais nous en avons mentionné les inconvénients (p.e. séquentialité, manque de cohérence, ...). Une première tentative d'avoir un ensemble d'outils ayant les avantages mais sans les inconvénients des boîtes à outils était représentée par les environnements de programmation intégrés. Il est clair que dans le cadre de la programmation et même dans celui de la maintenance et des testes unitaires et système, les EPI nous ont donné une satisfaction quasi complète. Mais, que peuvent nous offrir ces environnements dans le cadre de tout le cycle de vie d'un logiciel ? La réponse est certes dépendante du système générateur d'EPI utilisé, mais dans des termes généraux nous pouvons dire que ces environnements sont incapables de nous apporter une aide vraiment efficace dans des étapes du cycle de vie telles que la réalisation du cahier des charges, la spécification, la conception et même lors de la maintenance.

Les limites des EPI se font sentir surtout lorsqu'il s'agit de réaliser un "gros" projet où plusieurs personnes ineterviennent. Il est clair que l'on a besoin non seulement d'une aide automatique pour l' élaboration des logiciels, mais aussi pour pouvoir gérer correctement un ou plusieurs projets.

(2) Le fait de ne pas étudier les systèmes générateurs d'EPI utilisant la deuxième solution proposée par Winograd (i.e. l'utilisation de l'intelligence artificielle), n'est nullement un oubli de notre part. Cependant, ce type de systèmes ne sont pas du ressort de cette thèse et c'est pour quoi nous n'étudierons en détail que les systèmes basés sur la première solution proposée (i.e. l'utilisation de grammaires à contexte libre et du calcul de prédicats).

Afin de donner aux utilisateurs un environnement de programmation pouvant leur fournir une aide efficace pour la gestion des projets, il faut disposer d'un outil plus puissant que les environnements de programmation intégrés. Cet outil est l'*atelier de génie logiciel*. Pour citer quelques exemples, [Habermann 1979], [ALF 1987], [André 1986].

Pour donner des exemples plus pratiques, nous concentrerons notre attention sur un projet nous permettant de voir de manière générale les concepts de base des ateliers de génie logiciel. Il s'agit du Projet National de Génie Logiciel (PNGL) qui a démarré en 1979 sous les auspices de la Direction de la Recherche et du Transfert Technologique de l'Agence de l'Informatique.

L'objectif du PNGL(3) [Attard 1986] était de contribuer à améliorer de façon significative la productivité dans l'ensemble des opérations entrant dans le cycle de vie du logiciel, de la spécification à la maintenance, et de favoriser l'émergence d'une industrie de produits de génie logiciel œuvrant tant sur le marché national que sur le marché international.

Dans le cadre du PNGL, les ateliers de génie logiciel doivent avoir les caractéristiques suivantes:

- Ils doivent prendre en compte tout le cycle de vie du logiciel, depuis l'établissement du cahier des charges jusqu'à la maintenance;
- Ils doivent s'occuper de toutes les catégories d'utilisateurs : sociétés de services en ingénierie informatique, centres de recherches, constructeurs d'équipements informatiques, industries utilisatrices de l'informatique, administration et services;
- Ils doivent pouvoir traiter tous les types de logiciels : logiciels de base, applications scientifiques et industrielles, applications de gestion, temps réel, etc.
- Ils concernent tous les systèmes cibles, du micro-ordinateur bas de gamme au grand système temps réel et ceci quelque soit le constructeur des matériels cibles;

(3) La description que nous faisons du PNGL a été tirée de l'article "PNGL : Le projet de Génie Logiciel" de Roger Attard et Jean-Claude Rault, apparu dans le numéro 6 de la revue "Génie Logiciel" de l'Agence de l'Informatique en Novembre 1986.

- Ils concernent tout mode d'organisation et taille d'équipes (petites et grandes) de conception, développement et maintenance.

Dans le but d'atteindre ses objectifs, le PNGL a été organisé autour de deux volets complémentaires :

- **L'élaboration d'un noyau** réunissant le composant de base pour la construction, par intégration des outils appropriés, des ateliers de génie logiciel servant divers domaines techniques ou contextes de travail. Il s'agit de la structure d'accueil EMERAUDE [Bourguignon 1986] reposant sur le système UNIX;

- **Le développement d'un ensemble d'outils** de génie logiciel relatifs aux diverses phases du cycle de vie et à divers contextes d'utilisation; ces outils respectent des règles d'interfaçage avec la structure d'accueil. Il s'agit du volet "Outils" du PNGL.

Quatre éléments interviennent dans la définition des ateliers de génie logiciel pouvant résulter du PNGL : architecture, support matériel, logiciels et méthodes.

L'architecture des ateliers visés distingue :

- le poste de travail autonome de l'informaticien;
- le système de développement;
- le système d'exploitation.

Le support matériel des ateliers n'est pas supposé être limité à un seul type de machines : les différents matériels, actuels et à venir, les plus diffusés devront pouvoir supporter des systèmes de développement et d'exploitation et, donc, la structure d'accueil.

Les logiciels sont ceux correspondant aux fonctions suivantes :

- gestion des objets manipulés par les outils des ateliers;
- gestion des communications entre les différents processeurs logiciels et matériels et à

travers l'architecture répartie du système;
- gestion des interfaces entre le système et l'utilisateur;
- support pour la télédistribution ou la télémaintenance;
- support aux opérations entrant dans la construction et la maintenance des logiciels.
- aides à la formation des utilisateurs.

Les méthodes. Les ateliers construits à partir de la structure d'accueil devront permettre de mettre en œuvre une ou plusieurs méthodes de développement de logiciel existantes et largement utilisées au moment de la commercialisation des produits; aucune exclusive n'a été formulée quant à ces méthodes.

En ce qui concerne le volet **structure d'accueil**, celle-ci doit concourir aux objectifs suivants :

- assurer une compatibilité avec le système UNIX;
- permettre une intégration cohérente et une coopération entre les différents outils composant un atelier de logiciel;
- offrir une interface de dialogue homogène entre les utilisateurs et l'atelier de logiciel;
- assurer une gestion des différents objets manipulés dans l'atelier;
- assurer les communications entre les différentes composantes de l'architecture de l'atelier, notamment entre les postes de travail.

Quant au volet **"Outils"**, on considère les outils de la liste non-exhaustive suivante :

- Aide à la conception;
- Aide à la production;
- Aide à l'analyse;
- Aide à la maintenance;
- Aide à la gestion;
- Aide à la documentation.

On peut donc constater que le concept d'atelier de génie logiciel est bien plus vaste que

celui d'environnement de programmation intégré. Toutefois, il faut préciser que la frontière entre les deux est beaucoup plus floue que ne nous le laissent supposer les caractéristiques, des EPI intégrés et des ateliers de génie logiciel, citées précédemment. En effet, la structure de bloc du système A, concerne plus des fonctionnalités des ateliers de génie logiciel que des fonctionnalités des EPI.

2. LES MOTIVATIONS ET LES OBJECTIFS DU SYSTEME GEODE.

Lorsque nous nous sommes proposés de construire le système GEODE, nous avions l'intention de bâtir un système nous permettant de concevoir des environnements de programmation dont la vocation primaire était d'offrir aux programmeurs de moyens leur fournissant une aide efficace lors de la construction de programmes. Il est donc clair que GEODE appartient à la catégorie de systèmes générateurs d'environnements de programmation intégrés.

Or, depuis déjà quelques années nous constatons qu'il y a un certain nombre de ces systèmes, et que certains d'entre eux peuvent même être considérés comme des produits commerciaux. Pourquoi donc essayer de construire un nouveau système ?

Tout d'abord il faut constater que tous les générateurs d'EPI ne fonctionnent pas de la même manière. Certains générateurs se l''imitent'' à générer des éditeurs syntaxiques ALOE [Medina-Mora 1981], CEPAGE [Meyer 1984]. D'autres offrent la possibilité de construire des EPI plus généraux, mais il n'utilisent pas les mêmes formalismes : CENTAUR [Clément 1986] représente la syntaxe par une grammaire à contexte-libre et par une grammaire abstraite; la manière dont la sémantique sera représentée n'est pas encore complètement définie. CPS [Reps 1984] représente la syntaxe par une grammaire à contexte libre et par une grammaire abstraite, et la sémantique par des équations associées à cette dernière.

Néanmoins, on constate l'existence de certains inconvénients :

- Les outils constituant l'environnement généré ne sont pas tous décrits en utilisant un même formalisme;

- La description du langage de programmation concerné est faite en utilisant plusieurs formalismes;
- Lorsqu'il s'agit d'utiliser l'environnement généré, le programmeur n'est pas en mesure de choisir les outils dont il veut se servir;
- Dans certains cas les outils générés sont beaucoup plus restrictifs que les outils dits "classiques".

Sans avoir vraiment chercher à concurrencer des systèmes qui sont le fruit du travail de beaucoup de personnes et qui s'insèrent dans le cadre de projets d'envergure internationale, nous avons voulu essayer des formalismes, des mécanismes et des méthodes nous permettant de diminuer ou de remédier aux inconvénients mentionnés ci-dessus. Ceci est la motivation de la construction du système GEODE.

Les buts de GEODE sont donc les suivants :

- Avoir un seul formalisme pour la description des outils constituant l'environnement à générer;
- Avoir un unique formalisme pour la description du langage de programmation concerné;
- Offrir la possibilité de définir séparément les outils de l'environnement, tout en offrant la possibilité de les réunir ultérieurement;
- Donner aux utilisateurs la possibilité de choisir les outils de l'environnement dont il veulent se servir : concept de sous-environnement de programmation;
- Fournir au concepteur de l'environnement des outils de haut niveau;
- Construire des environnements hautement interactifs.

CHAPITRE II

Chapitre II.

GEODE : UN SYSTEME POUR LA GENERATION D'ENVIRONNEMENTS DE PROGRAMMATION INTEGRES.

1. PRESENTATION GENERALE.

1.1 INTRODUCTION.

Le système GEODE engendre des environnements de programmation à partir de la description d'un langage de programmation. Dans GEODE la description d'un langage de programmation est réalisée par une grammaire attribuée [Knuth 1968]. Une grammaire attribuée est constituée d'une grammaire à contexte libre et d'un ensemble d'équations sémantiques associées à cette dernière (c.f. Chapitres III et IV).

Dans GEODE une grammaire attribuée est utilisée de la manière suivante :

- La grammaire à contexte libre représente la description de la syntaxe du langage de programmation;
- L'ensemble d'équations sémantiques décrit l'interprétation que les différents outils de l'environnement donnent au langage.

Désormais, et sauf indication contraire, lorsque nous utiliserons le mot sémantique au sujet d'un outil de l'environnement, nous sous-entendrons que nous faisons référence à l'interprétation que les différents outils de l'environnement donnent au langage, et non à la sémantique du langage proprement dite.

Afin de mieux comprendre la notion d'interprétation, analysons quelques exemples :

Prenons d'abord un compilateur. Nous savons qu'un compilateur est un outil permettant de traduire un texte, dit programme source, en un texte, dit programme objet [Aho 1977]. Dans ce sens, nous pouvons dire qu'un compilateur est un outil de traduction. La sémantique ou interprétation qu'un compilateur donne au programme source, n'est que le programme objet issu de la traduction effectuée par le compilateur.

Analysons maintenant un éditeur syntaxique [Hansen 1971], [Donzeau-Gouge 1975], [[Medina-Mora 1982], [Reps 1981]. La caractéristique la plus importante de ce type d'éditeurs est qu'ils assurent la validité syntaxique des programmes construits par les utilisateurs. Certains de ces éditeurs assurent aussi des vérifications du style : "un identificateur ayant été déclaré en tant que variable entière ne peut être utilisé dans une expression portant sur les chaînes de caractères". Ce type de vérifications ne concernent plus la syntaxe d'un langage mais des contraintes liées à la sémantique statique du langage de programmation en question. Lorsque les éditeurs syntaxiques réalisent ce type de vérifications, nous préférons parler d'*éditeurs structurés*.

Dans les environnements d'édition structurée, la construction d'un programme est guidée par la grammaire du langage concerné et consiste à remplacer les symboles non-terminaux de la grammaire par les parties droites des productions où ceux-ci apparaissent en partie gauche (c.f. section 2.3 du chapitre III). Un programme n'est plus représenté par un texte mais par un arbre de dérivation ou de syntaxe abstraite.

Quelle serait donc la sémantique d'un programme dans un environnement d'édition structurée? Nous verrons dans le chapitre III que dans ce type d'environnements, la sémantique ou interprétation que l'éditeur donne au programme est liée à l'obtention d'une représentation textuelle du programme, à faire la liaison entre la représentation arborescente du programme et sa représentation textuelle, et à faire des vérifications des contraintes liées à la sémantique statique du langage de programmation.

Ces deux exemples, le compilateur et l'éditeur syntaxique nous montrent que la sémantique ou interprétation qu'un outil de l'environnement donne au programme dépend de

l'utilisation que l'on fait de ce programme.

Dans un environnement de programmation, **la sémantique ou signification d'un programme concerne l'interprétation que les différents outils de l'environnement donnent au programme.** Ainsi, un compilateur interprète un programme comme étant un texte à traduire, tandis qu'un éditeur syntaxique l'interprète soit, comme un arbre de dérivation, soit comme un texte où l'utilisateur effectuera les opérations classiques d'édition (i.e. insérer, supprimer, déplacer, etc.). Remarquons que les aspects syntaxiques d'un langage ne varient pas en fonction de l'outil de l'environnement utilisé.

Ceci constitue la philosophie de base du système GEODE : Dans un environnement de programmation, les outils partagent la définition syntaxique du langage, mais chacun lui associe une sémantique particulière. La description d'un langage de programmation est donc constituée d'une partie "syntaxe" et de plusieurs parties "sémantiques" (Figure 2.1). Il est à remarquer que dans cette figure nous n'avons représenté que trois outils : le compilateur, l'éditeur syntaxique et le débogueur.

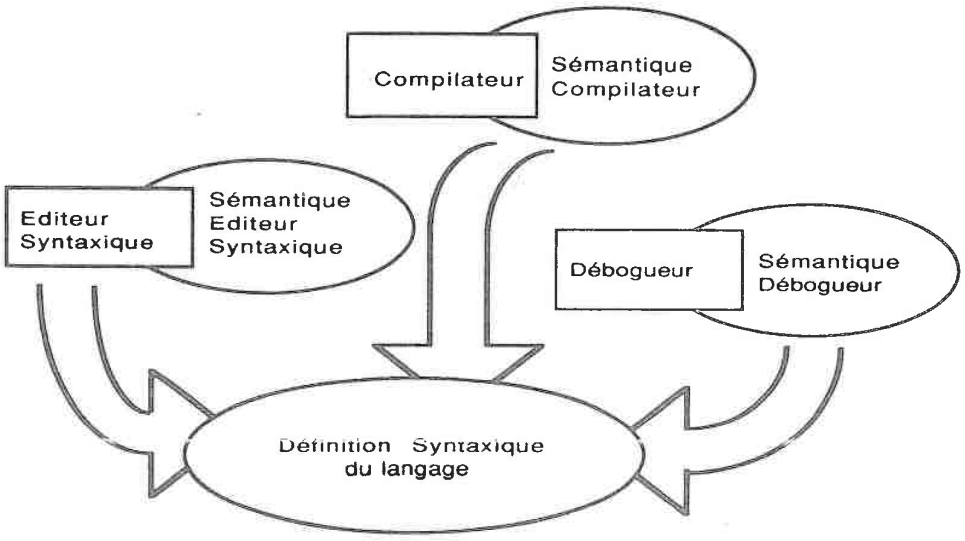


Figure 2.1 "La description syntaxique d'un langage et les descriptions des outils de l'environnement de programmation".

Etant donné que les descriptions des outils de l'environnement son dépendantes de la syntaxe du langage de programmation, toute modification du programme entraîne nécessairement des modifications concernant l'interprétation qu'un outil de l'environnement de programmation fait du programme. Ainsi, lorsque l'on modifie le programme, les outils devront le réinterpréter de façon à ce que leur vision du programme soit cohérente par rapport aux modifications effectuées. Ce processus de réinterprétation sera appelé évaluation.

Dans les sections qui suivent nous analyserons en détail l'architecture du système GEODE. Nous commencerons par une description générale du système qui pourra éventuellement paraître un peu confuse dans la mesure où dans cette description générale, les niveaux concepteur et utilisateur des environnements de programmation se mélangent. Nous pensons toute fois qu'il faut présenter le système GEODE dans son ensemble.

En revanche, dans le deuxième section du chapitre nous analyserons le système GEODE avec plus de détails. Nous ferons notamment la différence entre le système GEODE pour le concepteur d'un environnement de programmation et le système GEODE pour l'utilisateur de cet environnement. En fin, les détails techniques et théoriques de l'implantation de GEODE seront analysés d'une façon plus approfondie dans les chapitres III, IV et V.

1.2 LES ELEMENTS CONSTITUANT LE SYSTEME GEODE.

Le système GEODE est constitué de plusieurs outils ou modules; son architecture générale est présentée dans la figure 2.2.

GEODE est constitué de sept modules principaux :

- Un environnement de conception et définition de grammaires à contexte-libre;
- Un environnement de conception et description des outils;

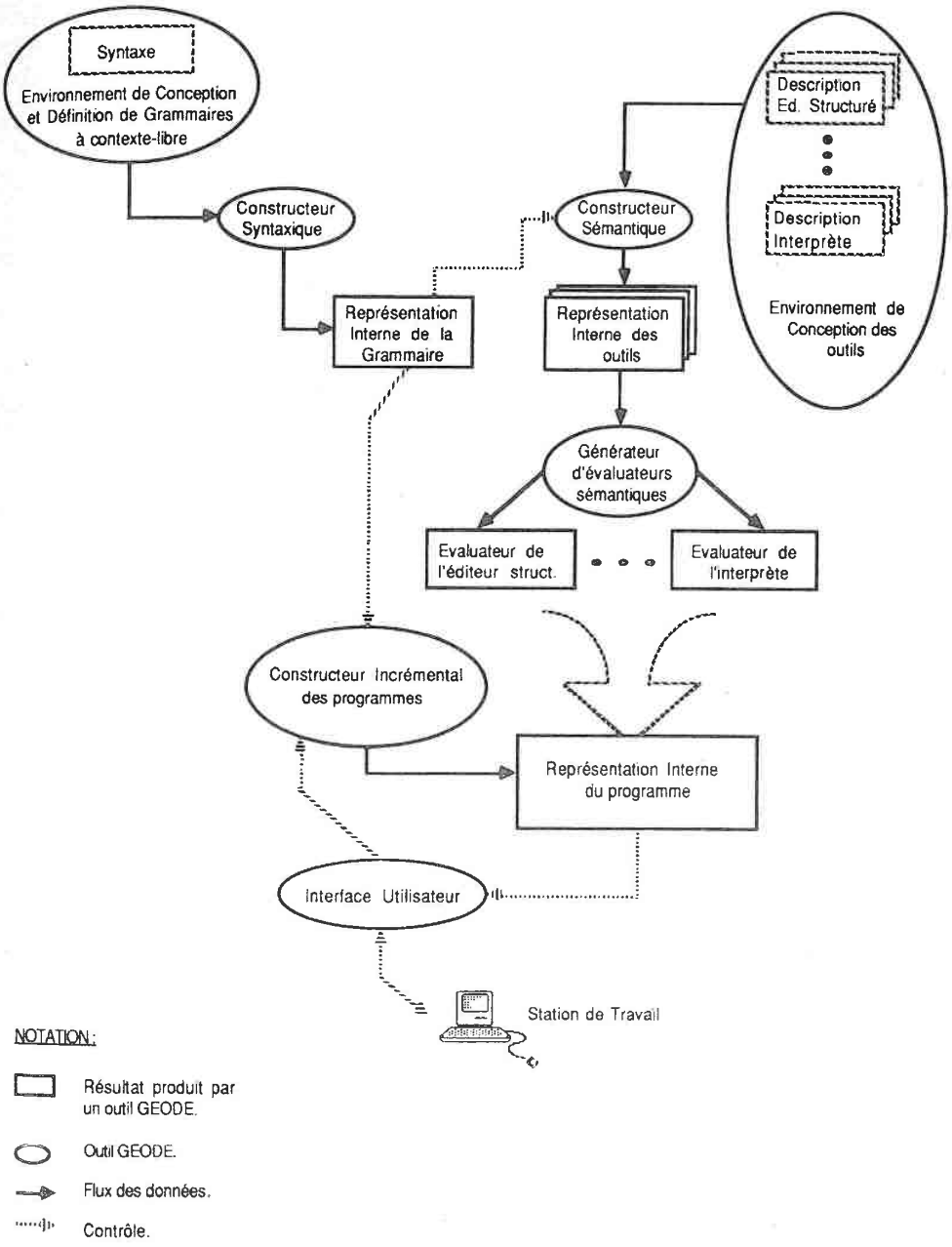


Figure 2.2 "L'architecture générale du système GEODE".

- Un module de construction de la représentation interne des grammaires à contexte-libre;
- Un module de construction de la représentation interne de la sémantique des outils;
- Un module de génération d'évaluateurs de la sémantique des outils;
- Un module de construction incrémentale de programmes;
- Un module d'interface utilisateur.

Afin de mieux comprendre quelles sont les fonctionnalités de ces modules, nous allons expliquer quelles sont les étapes à suivre pour générer un environnement de programmation avec GEODE : (Des explications plus précises seront données par la suite).

1. Décrire la syntaxe du langage de programmation en utilisant une grammaire à contexte libre. GEODE offre pour ceux-ci un environnement permettant d'écrire les règles ou productions de cette grammaire.

2. Chaque outil de l'environnement de programmation doit être décrit en utilisant l'environnement de conception des outils. Chaque outil est représenté par un ensemble d'équations sémantiques associées à la grammaire à contexte libre décrivant la syntaxe du langage de programmation en question.

3. Une fois la syntaxe du langage et les outils de l'environnement décrits, le système s'occupe de leur donner une représentation interne GEODE. Ceci est le rôle du constructeur syntaxique, pour la grammaire à contexte libre, et du constructeur sémantique pour ce qui concerne les outils.

4. GEODE construit en suite, à partir de la représentation interne des outils, un ensemble d'évaluateurs incrémentaux. Il existe un évaluateur pour chaque outil décrit dans la partie 2.

En ce qui concerne l'utilisation de l'environnement de programmation ainsi généré, on peut alors utiliser les menus et les options offerts par le module interface utilisateur. Ce

module fait appel au constructeur incrémental des programmes. Ce dernier s'occupe de générer, à partir de la représentation de la syntaxe du langage, et des évaluateurs des outils, une représentation interne du programme utilisateur. Chaque fois que le programme est modifié par l'utilisateur, les évaluateurs incrémentaux s'occupent de le rendre cohérent par rapport à la description des outils de l'environnement.

1.21 L'environnement de conception et définition de grammaires à contexte-libre.

Comme son nom l'indique, ce module permet au concepteur de l'environnement de programmation de concevoir et de définir la grammaire à contexte-libre qui génère le langage pour lequel on souhaite construire l'environnement .

Ce module constitue un véritable environnement de conception de grammaires qui a été construit par le système GEODE, à partir d'un langage qui a été appelé Langage de Description de Grammaires (LDG). LDG ainsi que cet environnement sont décrits dans [Canals 1987].

L'environnement de conception de grammaires s'occupe donc de fournir au concepteur de l'environnement de programmation un certain nombre d'outils lui permettant de définir facilement la grammaire du langage en question. Bien entendu, l'environnement de conception réalise des vérifications sémantiques sur la grammaire. Par exemple, tout symbole non-terminal doit apparaître au moins une fois en partie gauche d'une production ou règle de réécriture.

Lorsque le concepteur de l'environnement de programmation a mis au point la grammaire du langage concerné, l'environnement de conception génère un certain nombre de directives qui permettront au module de construction syntaxique d'obtenir la représentation interne GEODE de la grammaire.

L'environnement de conception de grammaires est constitué de trois outils principaux :

- Un outil d'édition structurée permettant au concepteur de décrire une grammaire à partir du langage LDG;
- Un outil de vérification sémantique qui est en fait intégré à l'éditeur structuré, et qui permet de réaliser des vérifications sémantiques sur la grammaire en cours de description;
- Un outil de génération qui s'occupe de produire les directives qui seront utilisées par le module de construction syntaxique.

1.22 L'environnement de conception et définition de la sémantique des outils.

De même que l'environnement de conception de grammaires, ce module est généré en utilisant le système GEODE. Cet environnement s'occupe de fournir à l'utilisateur un certain nombre d'outils qui l'aident à décrire plus facilement la sémantique qu'il associera aux outils de l'environnement de programmation à construire.

L'environnement de conception et définition de la sémantique des outils est basé sur le mécanisme connu sous le nom de grammaires attribuées (c.f. chapitre III) et il est généré à partir d'un langage qui a été appelé Langage de Description Sémantique (LDS). A l'heure actuelle cet environnement n'est pas complètement achevé.

L'environnement de conception de la sémantique des outils est constitué de quatre outils principaux :

- Un outil d'édition structurée qui permet au concepteur de décrire la sémantique des outils de l'environnement à construire;
- Un outil de vérification sémantique, intégré à l'éditeur structuré, qui s'occupe de vérifier un certain nombre de contraintes imposées par les grammaires attribuées;
- Un outil de génération s'occupant de produire les directives utilisées par le module de construction sémantique, qui se charge de construire la représentation interne GEODE de la sémantique des outils de l'environnement de programmation;

- Un interprète incrémental qui permet d'évaluer dynamiquement la sémantique d'un outil sur un programme utilisateur donné.

1.23 Le constructeur syntaxique.

Le constructeur syntaxique s'occupe de produire une représentation interne GEODE de la grammaire générant le langage de programmation en question. Cette représentation interne sera en suite utilisée par le constructeur sémantique et par l'environnement de conception et définition de la sémantique des outils de l'environnement de programmation.

La représentation interne GEODE des grammaires à contexte-libre sera analysée en détail dans les chapitres IV et V.

1.24 Le constructeur sémantique.

Le constructeur sémantique se charge de produire une représentation interne GEODE de la sémantique des outils de l'environnement de programmation. Le constructeur sémantique associe à chaque définition sémantique des outils la représentation interne de la syntaxe qui a été générée par le constructeur syntaxique.

La représentation interne GEODE de la sémantique des outils sera en suite utilisée par le module de génération d'évaluateurs sémantiques. Nous analyserons plus en détail cette représentation dans le chapitre IV.

1.25 Le générateur d'évaluateurs sémantiques.

Une fois que l'on a obtenu la représentation interne de la sémantique des outils de l'environnement de programmation, on doit être capable d'évaluer cette sémantique quelque soit le programme qui est en train d'être construit. Rappelons que lorsque nous évaluons la sémantique d'un outil sur un programme, nous sommes en fait en train d'interpréter le programme en fonction de l'outil en question.

Dans la philosophie du système GEODE à chaque outil est associé un évaluateur de sémantique ce qui permet la conception séparée et indépendante d'un outil par rapport aux autres. Cependant, étant donné que l'on évolue dans des environnements de programmation intégrés, les outils peuvent communiquer entre eux de façon à établir une collaboration inter-outils tout au long de la construction d'un programme. Cet aspect sera approfondi dans le chapitre IV.

1.26 Le constructeur incrémental des programmes.

Cet outil GEODE s'occupe de construire un programme utilisateur à partir de la représentation interne de la grammaire à contexte-libre générant le langage en question, et de certaines commandes que le programmeur utilise lorsqu'il se sert de l'éditeur structuré. Ces commandes peuvent être le développement d'un symbole non-terminal, le remplacement d'un symbole par un autre, etc.

Le constructeur incrémental des programmes s'occupe de donner au programme construit par l'utilisateur une forme interne pouvant être interprétée par tous les outils de l'environnement dont la sémantique a été définie au préalable. Cette forme interne est un arbre de dérivation qui a été construit à partir de la description syntaxique du langage, (c.f. chapitres III et IV).

Il ne faut pas confondre le rôle du constructeur incrémental des programmes et celui de l'éditeur structuré. Le premier se charge de construire la représentation interne du programme à partir des commandes issues du deuxième. L'éditeur structuré s'occupe en outre, de donner à l'utilisateur une représentation du programme facilement compréhensible en faisant abstraction de la représentation interne utilisée par le constructeur incrémental des programmes. Cet aspect sera analysé dans le chapitre IV.

1.27 L'interface utilisateur.

Comme son nom l'indique, ce module fournit à l'utilisateur de l'environnement de programmation une interface compréhensible et agréable à utiliser avec les outils construits au préalable. Ce module permet à l'utilisateur de travailler comme sous un éditeur de textes classique, c'est-à-dire qu'il cache complètement les représentations internes de la grammaire à contexte-libre et de la sémantique des outils de l'environnement de programmation. Ce module peut être vu comme un gestionnaire de fenêtres, où chaque fenêtre est associée à un ou à plusieurs outils de l'environnement de programmation. Des exemples sont donnés dans les annexes.

2. FONCTIONNALITES DU SYSTEME GEODE.

2.1 INTRODUCTION.

Dans la section précédente nous avons vu l'architecture générale du système GEODE. Cependant, lorsque nous avons brièvement expliqué le fonctionnement des modules, les rôles joués par le concepteur et l'utilisateur n'ont pas été suffisamment différenciés. Pour ce faire nous analyserons le système GEODE à deux niveaux différents :

- Le niveau conception de l'environnement de programmation.
- Le niveau utilisation de l'environnement de programmation.

Du point de vue de la conception, le système GEODE est concerné, principalement, par la mise au point de la grammaire à contexte-libre générant le langage en question (i.e. syntaxe), ainsi que par la mise au point de la sémantique des outils qui feront partie de l'environnement de programmation à construire. A partir des représentations internes de la syntaxe et de la sémantique du langage, le système GEODE génère des évaluateurs sémantiques pour les différents outils.

Du point de vue de l'utilisation, le système GEODE s'occupe de fournir à l'utilisateur

un moyen de construire son programme et de donner à celui-ci une représentation interne qui sera partagée par les différents outils. Tout au long de la construction du programme, les évaluateurs sémantiques générés au préalable vont se charger de l'interpréter d'après la sémantique associée à chacun des outils. Il est clair que les utilisateurs n'ont pas à s'occuper ni de la représentation interne des programmes, ni des évaluateurs sémantiques. Le but est de faire en sorte que les utilisateurs construisent leurs programmes comme s'ils utilisaient un éditeur de textes classique. Nous ne voulons pas bouleverser les habitudes prises par les programmeurs depuis longtemps déjà. Nous voulons leur donner des nouveaux outils pour les aider à construire des programmes plus efficaces.

Dans les sections suivantes nous expliquerons avec plus de détails les fonctionnalités du système GEODE des points de vue conception et utilisation des environnements de programmation.

2.2 LA CONCEPTION D'ENVIRONNEMENTS DE PROGRAMMATION DANS GEODE.

2.21 INTRODUCTION.

Le concepteur d'un environnement de programmation doit utiliser deux modules GEODE, à savoir : l'environnement de conception et définition de grammaires à contexte-libre, et l'environnement de conception et définition de la sémantique des outils. Ces environnements s'occupent de générer un ensemble de directives qui permettront aux constructeurs syntaxique et sémantique d'obtenir les représentations internes correspondantes. La figure 2.3 montre l'architecture du système GEODE permettant de concevoir des environnements de programmation.

Lorsque le concepteur a défini la sémantique des outils de l'environnement de programmation, le système GEODE génère des évaluateurs qui s'occupent d'interpréter les programmes des utilisateurs selon la sémantique des différents outils. Ces évaluateurs travaillent simultanément sur le programme ce qui signifie que les outils constituent un

environnement de programmation intégré.

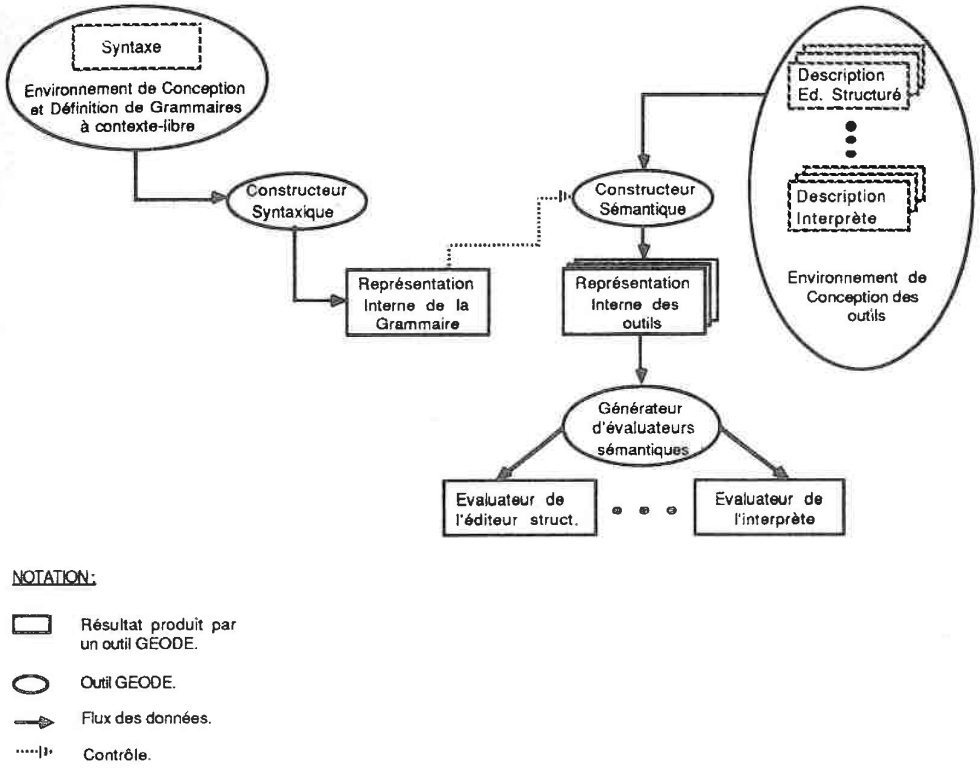


Figure 2.3 "Le système GEODE pour la conception d'environnements de programmation".

2.22 L'ENVIRONNEMENT DE CONCEPTION ET DEFINITION DES GRAMMAIRES A CONTEXTE LIBRE.

La description du langage de programmation pour lequel GEODE générera un environnement de programmation est exclusivement une description syntaxique faite par une grammaire à contexte libre. Cette grammaire est mise au point en utilisant un environnement de conception qui a été généré par GEODE. Cet environnement permet une saisie interactive de la grammaire et il propose différentes facilités (i.e. traitement des listes, contrôles de cohérence, etc.).

L'environnement a été construit au tour du **L**angage de **D**escription de **G**rammaires (LDG) défini par [Canals 1987]. LDG et les outils de son environnement sont présentés dans les annexes.

2.23 L'ENVIRONNEMENT DE CONCEPTION DES OUTILS.

GEODE propose un langage de définition d'outils unique basé sur les grammaires attribuées : le langage LDS (**L**angage de **D**escription d'équations **S**émantiques). La définition d'un outil se fait en deux parties :

- déclaration des attributs attachés aux différents symboles de la grammaire à contexte libre décrivant la syntaxe du langage de programmation en question;
- écriture des équations sémantiques décrivant le comportement des outils vis à vis de cette grammaire.

Des exemples de définition d'outils d'un environnement de programmation sont donnés dans le chapitre IV.

De même que LDG, LDS sera doté d'un environnement de programmation composé d'un éditeur structuré et d'un interprète-metteur au point. A l'heure actuelle cet environnement n'a pas encore été implanté.

L'éditeur structuré de l'environnement possède les fonctionnalités suivantes :

- **Rappel des règles de la grammaire à contexte libre** : l'éditeur rappelle à l'utilisateur les règles de la grammaire au fur et à mesure, facilitant ainsi l'association des équations sémantiques à ces règles;
- **Déclaration automatique d'attributs** : chaque attribut utilisé est automatiquement déclaré par le système;
- **Mémorisation et édition des fonctions sémantiques** : le système mémorise le nom des

fonctions sémantiques que l'utilisateur devra écrire pour compléter la définition d'un outil. Il peut à tout moment demander leur édition pour les construire, les modifier ou les tester.

L'interprète-metteur au point est constitué de trois parties :

- Un constructeur d'arbres syntaxiques;
- Un évaluateur incrémental d'attributs;
- Des outils de visualisation.

Le constructeur d'arbres syntaxiques permet au concepteur de fabriquer rapidement des arbres, à partir de la grammaire à contexte libre, sans avoir à utiliser l'éditeur. Grâce à l'évaluateur incrémental, il pourra effectuer l'évaluation d'attributs sur ces arbres, éventuellement en mode pas à pas. Les outils de visualisation affichent alors les résultats : valeurs des instances d'attributs ou graphe de dépendance (c.f. chapitre III).

2.24 LA NOTION DE SOUS-ENVIRONNEMENT DE PROGRAMMATION.

Supposons que pour un projet déterminé le concepteur de l'environnement ne souhaite pas se servir d'un outil particulier. Par exemple, imaginons un outil qui "oblige" un programmeur à insérer des commentaires dans son programme. Il serait peut-être indésirable d'utiliser un tel outil lorsque l'on construit un programme dans le but de se familiariser avec un nouveau langage de programmation. Comment faire en sorte que cet outil soit temporairement "mis à l'écart" d'un environnement de programmation intégré? La réponse est le concept de sous-environnement de programmation.

Un sous-environnement de programmation est un sous-ensemble de l'environnement de programmation originalement construit. Dans un sous-environnement de programmation l'utilisation d'un ou plusieurs outils faisant partie de l'environnement original est inhibée. Un environnement de programmation peut avoir plusieurs sous-environnements associés. Les sous-environnements de programmation ne peuvent être définis en GEODE, qu'au niveau de

la conception.

Toute fois, il faudra noter que certains outils ne pourront pas être "éliminés" des environnements de programmation. C'est notamment le cas de l'outil d'édition structurée, car celui-ci constitue le seul moyen dont disposent les utilisateurs pour construire leurs programmes.

En fin, un sous-environnement de programmation peut-être crée non seulement dans le but de rendre un environnement plus souple à utiliser, mais aussi dans le but de suivre des consignes particulières concernant la construction des programmes. Nous pourrions ainsi rendre l'adjonction des commentaires dans les programmes obligatoire.

2.3 L'UTILISATION DES ENVIRONNEMENTS DE PROGRAMMATION GENERES PAR GEODE.

2.31 INTRODUCTION.

Au niveau de l'utilisation d'un environnement de programmation, le système GEODE se charge d'abord, de fournir à l'utilisateur un moyen simple et compréhensible lui permettant de construire ses programmes, et ensuite de faire en sorte que tous les outils de l'environnement ayant été définis par le concepteur collaborent et coopèrent tout au long de la construction de ces programmes. La figure 2.4 montre le système GEODE du point de vue de l'utilisation des environnements de programmation.

Un programme utilisateur est considéré par le système GEODE, comme un arbre de dérivation auquel on a attaché un certain nombre d'informations sémantiques d'après la définition des outils de l'environnement. Chaque fois que cet arbre est modifié, les évaluateurs sémantiques, générés lors de la conception de l'environnement, s'occupent d'interpréter le nouvel arbre selon la sémantique de différents outils.

De même que la définition syntaxico-sémantique d'un langage, les programmes des

utilisateurs sont représentés par un arbre de dérivation qui a été implanté en suivant une philosophie orientée objet (c.f. chapitre IV). Les utilisateurs ne s'occupent que de construire leurs programmes en utilisant l'éditeur structuré de l'environnement de programmation, mais éventuellement, ils peuvent faire appel à d'autres outils tels que la documentation automatique, le débogage ou même l'exécution. Néanmoins, l'intégration des différents composants fait apparaître, pour l'utilisateur, un environnement de programmation non pas comme une collection d'outils distincts, mais plutôt comme un outil unique disposant de fonctionnalités diverses s'appliquant à différents moments du processus de construction du programme. Ceci se traduit au niveau de l'interface utilisateur qui est unique et commune à tous les outils.

Les sections qui suivent décrivent de manière détaillée le fonctionnalités offertes par le système GEODE aux utilisateurs des environnements générés [Canals 1987].

2.32 L'INTERFACE UTILISATEUR.

L'interface utilisateur qui offre GEODE est hautement interactive. Elle est multifenêtres et la communication avec l'utilisateur est réalisée à travers des menus déroulants qui sont actionnés par un dispositif de pointage (i.e. la souris). Des exemples de cette interface utilisateur seront donnés dans les annexes.

Les menus de l'interface permettent donc à l'utilisateur d'effectuer un certain nombre d'opérations générales telles que le chargement et la sauvegarde de documents, la création et destruction de fenêtres, l'appel de différents outils de l'environnement, etc.

L'interface offre aussi des fenêtres où les documents de l'utilisateur sont construits et modifiés de façon interactive. L'utilisateur peut créer ces fenêtres explicitement et il peut en créer un nombre arbitraire. Néanmoins, une seule fenêtre est active à la fois et c'est sur elle qu'agissent les différentes commandes de l'utilisateur. Chaque fenêtre représente l'application d'un outil sur le programme. Cependant, comme nous l'avons mentionné, l'intégration de tous ces outils, donne l'impression à l'utilisateur de travailler avec un outil

unique.

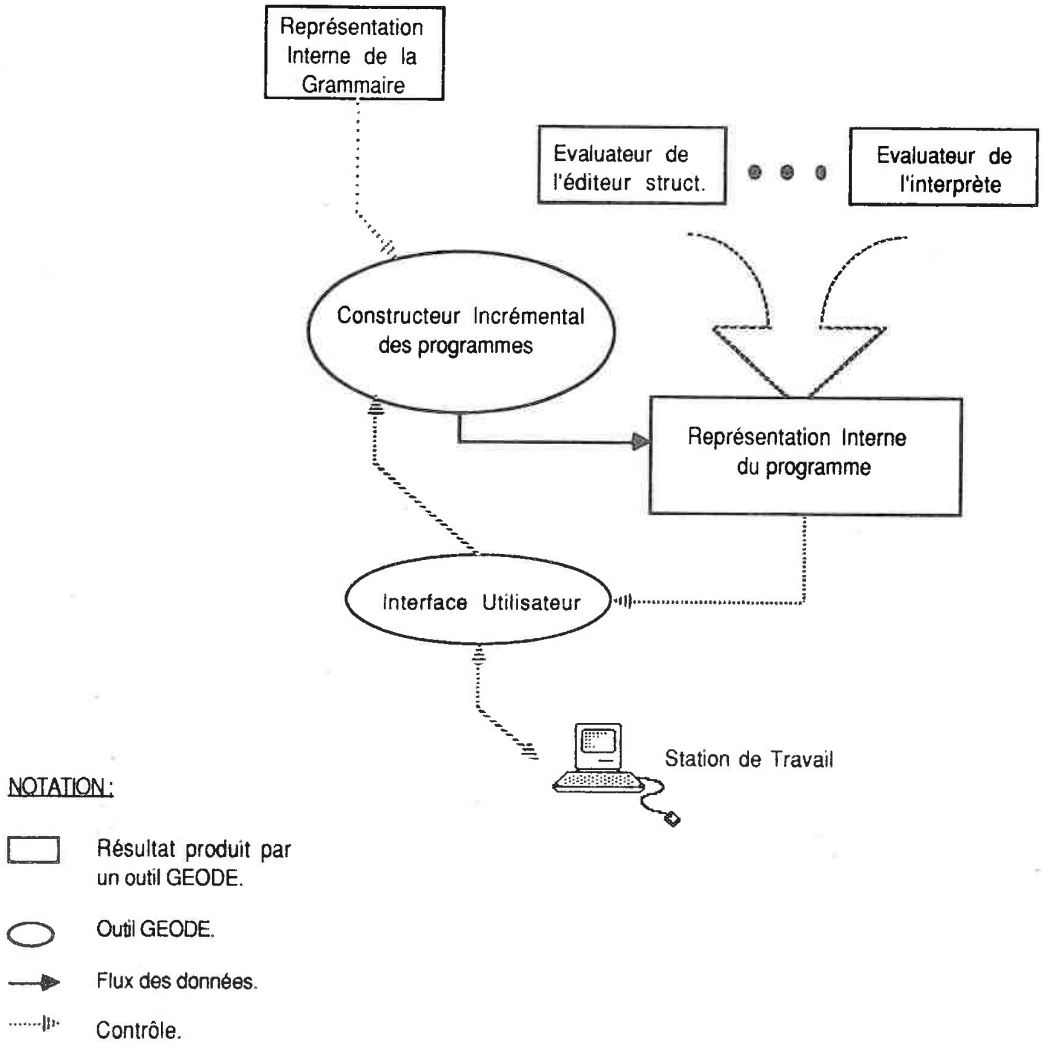


Figure 2.4 "L'utilisation d'un environnement de programmation généré par le système GEODE".

La manipulation du contenu des fenêtres est réalisée en utilisant la notion de *sélection*. Une sélection représente la partie du programme où vont agir les commandes de l'utilisateur.

Au niveau de la représentation interne du programme, une sélection représente soit un nœud de l'arbre de dérivation, soit un sous-arbre; au niveau de l'écran, une sélection est représentée par une zone de texte apparaissant en inverse vidéo. Il est clair que l'utilisateur fait sa sélection sur le texte du programme, soit en utilisant le curseur, soit en utilisant la souris.

Une sélection représente toujours une structure syntaxique complète, que ce soit une unité lexicale simple ou bien une phrase entière du langage. Si la sélection est faite à travers le curseur, l'utilisateur ne peut sélectionner qu'une unité lexicale simple, mais il peut se servir de la commande "englober" pour accéder à la structure syntaxique immédiatement supérieure par rapport à l'arbre représentant le programme. En revanche, si l'utilisateur se sert de la souris, il peut sélectionner soit une unité lexicale simple, soit une phrase. Pour sélectionner une phrase entière, on noircit une zone du texte et le système s'occupe de compléter cette zone, selon la structure syntaxique correspondante.

2.33 LA CONSTRUCTION DES PROGRAMMES.

En ce qui concerne l'éditeur structuré, l'utilisateur dispose de deux modes de création pour ses programmes : le mode "guidé par les menus" et le mode "à la frappe". Ces deux modes sont disponibles à tout moment.

Le mode "guidé par les menus" consiste à demander le développement de la sélection courante. Le système propose alors les différentes structures syntaxiques valables. L'utilisateur peut donc choisir la structure syntaxique qui lui convient sans avoir à se préoccuper de taper du texte. La construction du programme est réalisée en effectuant une série de développements successifs et aucune analyse syntaxique n'est nécessaire. Ce mode est particulièrement utile pour l'apprentissage d'un langage.

Or, lorsque l'on a l'habitude de travailler avec un langage de programmation, ce mode est particulièrement lourd à utiliser étant donnée les développements qu'il faut effectuer séquentiellement. C'est la raison pour laquelle le système GEODE offrira le mode "à la

frappe". Ce mode de construction de programmes permet à l'utilisateur de construire son texte de la même manière que sous un éditeur de textes classique. L'utilisateur s'occupe donc de taper son texte et le système doit donc effectuer une analyse syntaxique afin de construire l'arbre de dérivation représentant le programme. Le système peut, à la demande, insérer des symboles non-terminaux, ce qui permettra à l'utilisateur de construire son programme par raffinements successifs. A l'heure actuelle, le mode "à la frappe" n'est pas encore opérationnel.

D'autres commandes seront également disponibles (i.e. commandes d'insertion, commandes de déplacement, etc.).

2.34 LA MODIFICATION DES PROGRAMMES.

En ce qui concerne la modification de programmes, le système GEODE offre à l'utilisateur deux possibilités :

- La modification manuelle
- La modification automatique.

La modification manuelle consiste à utiliser des commandes agissant sur la sélection courante. Outre les commandes offertes par un éditeur de textes classique (i.e. insertion, élimination, etc.) le système GEODE permet d'utiliser la commande "réduction". Cette commande effectue l'opération inverse de l'opération de développement (c.f. chapitre IV). L'opération de réduction consiste donc à remplacer toute une phrase du programme par le symbole non-terminal duquel celle-ci est issue. Par rapport à l'arbre de dérivation qui représente le programme de l'utilisateur, cette commande effectue un élagage permettant de supprimer le sous-arbre concernant la phrase en question.

Des commandes de recherche sont également disponibles afin de permettre un positionnement rapide sur une partie du texte. Les recherches peuvent être effectuées, soit sur des chaînes de caractères, soit sur des schémas d'arbre.

La modification automatique concerne des modifications importantes ou systématiques sur le programme de l'utilisateur. Ces modifications peuvent être programmées en utilisant les primitives de base du constructeur incrémental de programmes. Ces primitives permettent de rechercher, de créer ou de supprimer des nœuds ou des schémas d'arbre.

Le langage utilisé pour programmer ce type de modifications est le langage Le_Lisp 15.2 [Chailloux 1986]. Les fonctions Le_Lisp réalisant les modifications souhaitées peuvent être liées, soit à une clé, afin d'en faire de nouvelles commandes de l'éditeur, soit à un item de menu qui sera déclenché par la souris.

2.35 LA VERIFICATION ET LA DOCUMENTATION DES PROGRAMMES.

Mises à part le création et la modification des programmes, le système GEODE donne à l'utilisateur la possibilité d'avoir des opérations qui concernent l'analyse statique des programmes. L'ensemble de ces opérations sera référencé sous le terme "vérification et documentation de programmes". Ces opérations se chargent de réaliser de vérifications de la sémantique statique des programmes, mais elles offrent aussi la possibilité d'avoir des renseignements sur les programmes eux-mêmes. Par exemple, dans le cadre d'un langage de programmation à structure de bloques (i.e. Pascal, Ada, Algol, etc.), on peut disposer à tout moment d'informations concernant les différents identificateurs utilisés dans un programme.

L'outil de vérification et documentation de programmes, que nous appelerons "documenteur" par la suite, fournit des renseignements sur les identificateurs utilisés dans le programme. Les informations qu'il fournit sont celles qui sont contenues dans la table des symboles : type, caractéristiques; ou calculées à partir de cette table : lieu de déclaration, lieux de modification et d'utilisation des identificateurs. Lorsque la sélection courante dans le texte correspond à un identificateur, la commande de vérification et documentation automatique fait apparaître une nouvelle fenêtre dans laquelle sont inscrites les informations. Cette fenêtre peut disposer en plus d'un petit cadre d'édition permettant à l'utilisateur d'associer à l'identificateur des informations supplémentaires qui seront éditées à l'activation suivante du documenteur.

La documentation de programmes sera analysée de manière plus détaillée dans le chapitre IV.

Chapitre III.

LES GRAMMAIRES ATTRIBUEES.

1. DEFINITIONS ET NOTATIONS.

Dans cette section nous décrivons les grammaires attribuées et nous présentons la notation qui sera utilisée dans les chapitres suivants.

Une *grammaire attribuée*, notée GA, est constituée par [Knuth 1968] :

i) Une grammaire à contexte libre G = (Vt, Vn, St, P), où
   + Vt est le vocabulaire terminal ;
   + Vn est le vocabulaire non-terminal ;
   + St est un élément distingué de Vt appelé axiome ;
   + P est l'ensemble de productions de G, où chaque production $p \in P$ a la forme :

   $$p : X_0 \rightarrow X_k, \qquad \text{où } X_k \in (Vt \cup Vn)^* \ \forall k = 1, \ldots, n \text{ et } X_0 \in Vn.$$

ii) A chaque symbole $X \in Vt \cup Vn$ sont associés deux ensembles disjoints : un ensemble d'*attributs synthétisés* S(X) et un ensemble d'*attributs hérités* I(X). L'union de ces ensembles sera notée $A(X) = S(X) \cup I(X)$. Nous imposons que $I(St) = \varnothing$ et $S(X) = \varnothing$ si $X \in Vt$.

   Pour chaque production p et chaque attribut α de X, c'est-à-dire, $\alpha \in A(X)$, nous disons que p possède une instance d'attribut X.α. Si $\alpha \in S(X)$ nous avons une *instance d'attribut synthétisé*, et si $\alpha \in I(X)$ nous avons une *instance d'attribut hérité*.

iii) A chaque production $p : X_0 \rightarrow X_1, \ldots, X_n$, est associé un ensemble d'*équations sémantiques* Ep, tel que :

Pour chaque attribut synthétisé $\alpha \in S(X_0)$, il existe dans Ep une et seulement une équation sémantique définissant $X_0.\alpha$. Cette équation a la forme :

$X_0.\alpha = \varphi (X_k.\beta_i)$, où φ est une fonction ou expression symbolique, et $\beta_i \in A(X_k)$, $\forall k = 0, ..., n$.

Pour chaque attribut hérité $\alpha \in I(X_k)$, $\forall k = 1, ..., n$, il existe dans Ep une et seulement une équation sémantique définissant $X_k.\alpha$. Cette équation a la forme :

$X_k.\alpha = \varphi (X_k.\beta_i)$, où φ est une fonction ou expression symbolique, et $\beta_i \in A(X_k)$, $\forall k = 0, ..., n$.

Dans une équation sémantique, l'instance d'attribut synthétisée $X_0.\alpha$ et les instances d'attributs héritées $X_k.\alpha$, $\forall k = 1, ..., n$, sont définies en fonction des autres instances d'attribut de p. Ces dernières constituent ce que l'on appelle l'*ensemble de dépendances* de l'équation sémantique et il sera noté $DS(X_k.\alpha)$.

Ainsi, lorsque nous notons $DS(X_0.\alpha) = \{ X_k.\beta_i \}$ nous indiquons que l'instance d'attribut $X_0.\alpha$ est calculée à partir des instances d'attributs $X_k.\beta_i$, $\forall k = 1, ..., n$.

Une grammaire attribuée est en *forme normale*, si les arguments des fonctions dans la partie droite des équations sémantiques sont limités à $I(X_0)$ et $S(X_k)$, $\forall k = 1, ..., n$. (c.f. section 1.1)

Un *graphe de dépendances* pour une production p, noté DGp, donne les relations de dépendance entre les différentes instances d'attribut de p :

DGp = (Np , Ap)

où l'ensemble de nœuds Np représente l'ensemble d'instances d'attributs dans p et l'ensemble d'arcs Ap représente l'ensemble de pairs de dépendance ayant la forme :

$(X_i.\beta \rightarrow X_k.\alpha)$ où $X_i.\beta \in DS(X_k.\alpha)$

Un *arbre de dérivation attribué* T est un arbre de dérivation pour une GA. (i.e. à chaque nœud de T sont associées des instances d'attributs héritées et synthétisées). Les instances d'attributs synthétisées ramènent l'information vers la racine de T, tandis que les instances d'attributs hérités la ramènent vers les feuilles de T.

On définit un *graphe de dépendances* pour T, noté DG_T, pour représenter les dépendances entre les différentes instances d'attributs dans T. DG_T est construit en mettant ensemble les diverses DGp, d'après la structure syntaxique de T.

Une instance d'attribut $X_k.\alpha$ est appelée *successeur* d'une instance d'attribut $X_i.\beta$, s'il existe une arête les connectant dans n'importe quel arbre de dérivation T.

Une grammaire attribuée est dite *non-circulaire* si DG_T ne contient aucun circuit quelque soit l'arbre de dérivation T.

Dans une grammaire attribuée GA, les équations sémantiques associées à une production p de la forme $X_0 \rightarrow X_1 \ldots X_n$ définissent la manière dont les instances d'attribut synthétisées $X_0.\alpha$ et les instances d'attribut hérités $X_k.\alpha$ ($\forall k = 1, \ldots, n$) seront calculées. Dans un arbre de dérivation attribué T, généré à partir de GA, plusieurs productions y sont représentées. Le problème consiste alors à évaluer toutes les instances d'attributs attachées aux nœuds de T. Ce processus d'évaluation, qui sera traité dans la section 2.4 de ce chapitre, est connu sous le nom d'*évaluation d'attributs*. Pour que cette évaluation soit optimum, elle doit suivre, implicitement ou explicitement, le graphe de dépendances DG_T.

Pour définir plus formellement l'évaluation d'attributs, il faut constater qu'un arbre de dérivation attribué T définit un système d'équations sémantiques où les inconnues sont toutes les instances d'attribut que T contient (à l'exception de celles initialisées par des constantes). L'évaluation d'attributs est donc la résolution de ce système d'équations. Lors de la résolution de ce système d'équations on peut avoir deux cas : [Courcelle 1984]

- Soit la grammaire GA est circulaire, le système peut alors avoir plusieurs solutions ou éventuellement ne pas avoir de solution du tout ;

- Soit la grammaire GA n'est pas circulaire, le système possède alors une solution unique.

Dans la pratique, l'évaluation d'attributs est souvent limitée à calculer la *valeur sémantique d'un arbre de dérivation attribué T*. On définit cette valeur comme l'ensemble d'instances d'attribut synthétisées de la racine de T [Knuth 1968].

Une grammaire attribuée est dite *grammaire attribuée bien formée* lorsqu'elle réunit les caractéristiques suivantes :

- La grammaire est non-circulaire;
- Chaque production de la forme :

$$p : X_0 \rightarrow X_1 X_2 \ldots X_n$$

inclut une équation sémantique pour tous les attributs synthétisés de X_0 et pour tous les attributs hérites de X_i, $\forall i = 1, \ldots, n$.

Cette thèse ne concerne que les grammaires attribuées bien formées et, sans indication contraire, nous ne ferons aucune supposition par rapport à la forme normale. Aussi, lorsque nous parlerons d'évaluation d'attributs, nous supposerons que l'on se limite à calculer la valeur sémantique d'un arbre de dérivation attribué.

1.1 LES GRAMMAIRES ATTRIBUEES EN FORME NORMALE.

Pour mieux comprendre ce que la forme normale implique, reconsidérons la production : $p : X_0 \rightarrow X_1 X_2 \ldots X_n$. Dans un arbre de dérivation cette production est liée au sous-arbre de la figure 3.1. Le flux d'information associé à ce sous-arbre y est aussi représenté. Ainsi nous voyons que l'information d'entrée est constituée de $I(X_0) \cup S(X_i)$, $\forall i = 1, \ldots, n$ et que l'information de sortie est constituée de $S(X_0) \cup I(X_i)$, $\forall i = 1, \ldots, n$.

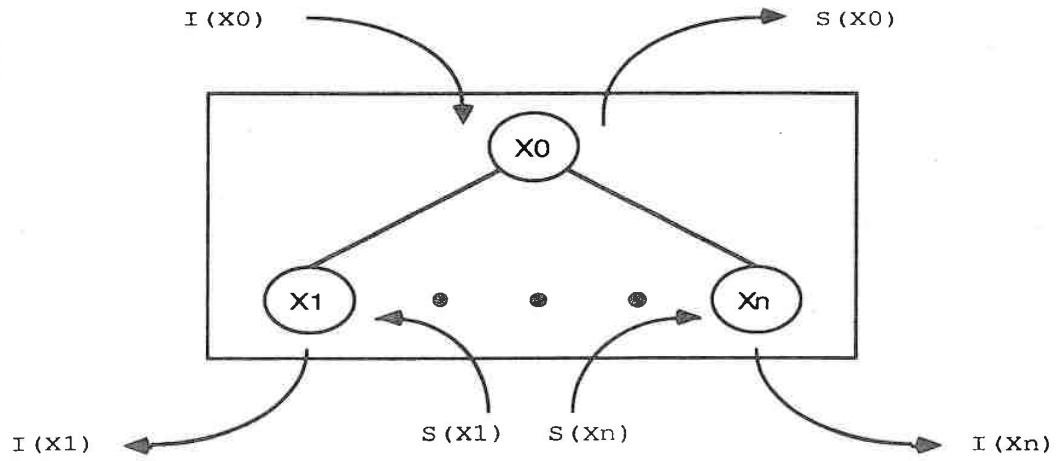


Figure 3.1 "Le flux d'information associé à un sous-arbre de dérivation attribué".

Si nous considérons maintenant les contraintes imposées par la forme normale, nous constatons qu'elles signifient que l'information est propagée vers la racine par les attributs synthétisés et vers les feuilles par les attributs hérités, c'est-à-dire que :

$$S(X_0) = \varphi(I(X_0) \cup S(X_i)), \forall i = 1, ..., n$$
$$I(X_i) = \varphi(I(X_0) \cup S(X_i)), \forall i = 1, ..., n$$

Alors que si la grammaire n'était pas en forme normale, nous aurions que :

$$S(X_0) = \varphi(I(X_0) \cup S(X_i) \cup S(X_0) \cup I(X_i)), \forall i = 1, ..., n$$
$$I(X_i) = \varphi(I(X_0) \cup S(X_i) \cup S(X_0) \cup I(X_i)), \forall i = 1, ..., n$$

Comparons les graphes de dépendances de ces deux approches (c.f. figures 3.2 et 3.3).

Le fait d'imposer la forme normale n'entraîne pas de restrictions quant à la puissance d'une grammaire attribuée. Cette contrainte définit l'utilisation des attributs selon le but pour lequel ils ont été crées : les attributs synthétisés propagent l'information des feuilles vers la racine de l'arbre de dérivation, tandis que les attributs hérités propagent l'information de la racine vers les feuilles [Jourdan 1984].

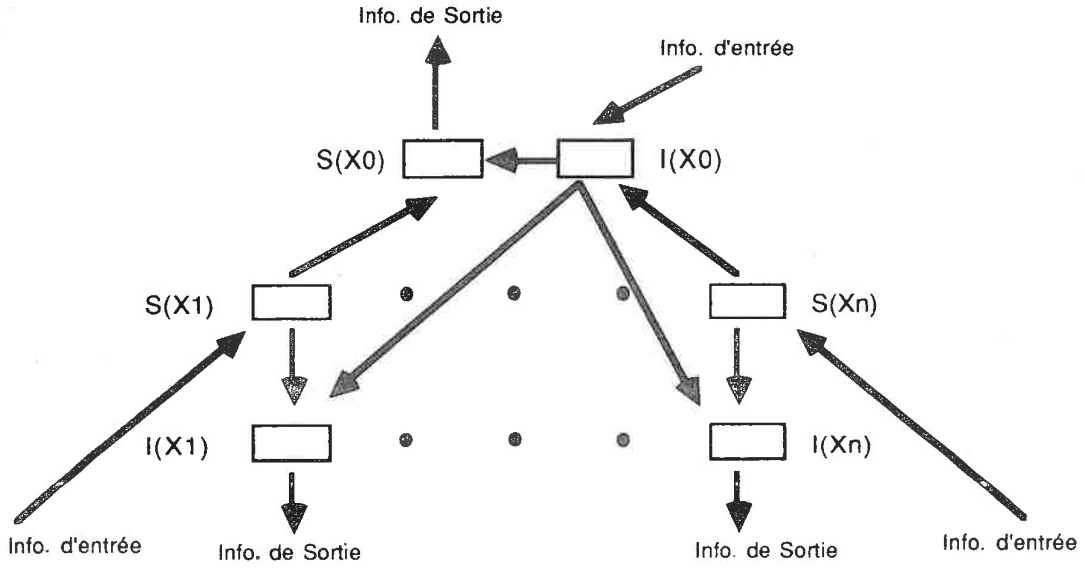


Figure 3.2 "Le graphe de dépendances d'un sous-arbre de dérivation respectant la forme normale".

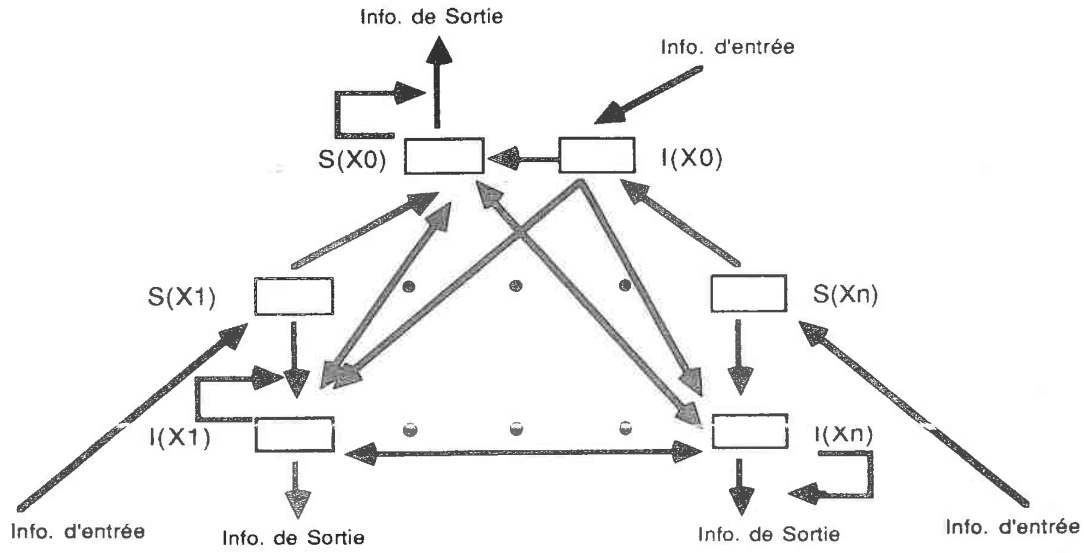


Figure 3.3 "Le graphe de dépendances d'un sous-arbre de dérivation ne respectant pas la forme normale"

2. NOTIONS DE BASE DES GRAMMAIRES ATTRIBUEES.

2.1 INTRODUCTION.

Il est bien connu que les grammaires à contexte libre constituent un outil simple et très puissant pour définir la syntaxe d'un langage de programmation, mais qu'elles s'avèrent incapables d'en définir la sémantique [Aho 1973].

Dans la définition d'un langage de programmation, la sémantique s'occupe de donner un sens complet et non ambigu à tout programme du langage, ce qui permet de l'interpréter, mais aussi de contribuer à la résolution des problèmes de construction automatique et de preuve de correction de compilateurs [LNCS 1980].

Force est de constater que la définition formelle de la sémantique constitue un problème difficile comme le montrent la diversité des approches proposées [Livercy 1978].

Parmi les formalismes proposés, les GA constituent un des mécanismes les plus puissants permettant de définir d'une façon simple et précise la sémantique d'un langage de programmation [Knuth 1968]. Les GA ont déjà été utilisées dans des domaines tels que la génération automatique de traducteurs [Farrow 1982] [Lorho 1977], l'optimisation du code généré par un compilateur [Néel 1974], les preuves de descriptions de traitements de textes [Pair 1976], la détection d'anomalies dans les programmes [Arthur 1981] et dernièrement les GA ont été utilisées pour la spécification d'éditeurs syntaxiques [Reps 1983]. On pourra trouver dans [Deransart 1985] une bibliographie très complète sur l'utilisation des GA dans divers domaines.

2.2 SPECIFICATION DE LANGAGES A L'AIDE DES GRAMMAIRES ATTRIBUEES.

Les GA représentent une notation descriptive permettant de spécifier un langage de façon modulaire : la syntaxe est définie par une grammaire à contexte libre et la sémantique est définie par un ensemble d'équations associées aux productions de cette grammaire. Nous définissons la valeur sémantique d'un arbre de dérivation attribuée T, comme celle composée

par l'ensemble des valeurs des attributs synthétisés de la racine de T. La propagation des valeurs des attributs à travers cet arbre n'est pas spécifiée explicitement par la GA, elle est implicitement définie par les équations et par la forme de l'arbre.

2.21 La sémantique décimale des nombres binaires.

Dans cet exemple, il s'agit de donner à chaque nombre binaire généré par la grammaire à contexte libre, son équivalent décimal. Pour montrer les différentes manières de réaliser ceci, nous donnerons deux GA, la première ne fait intervenir que des attributs synthétisés et la deuxième utilise des attributs synthétisés et des attributs hérités.

Grammaire 3.1. [Knuth 1968]

1) Nombre → Nombre Chiffre
   Nombre1.valeur = 2 * Nombre2.valeur + Chiffre.valeur;

2) Nombre → Chiffre
   Nombre.valeur = Chiffre.valeur;

3) Chiffre → 0
   Chiffre.valeur = 0;

4) Chiffre → 1
   Chiffre.valeur = 1;

Cette GA utilise un attribut synthétisé "valeur" pour les non-terminaux Nombre et Chiffre. L'attribut "valeur" d'un non-terminal X représente la valeur du sous-arbre dont la racine est X. La valeur décimale associée au nombre binaire généré est donc celle de l'attribut "valeur" du nœud racine. La figure 3.4 montre comment obtenir la valeur décimale associée au nombre binaire **110**.

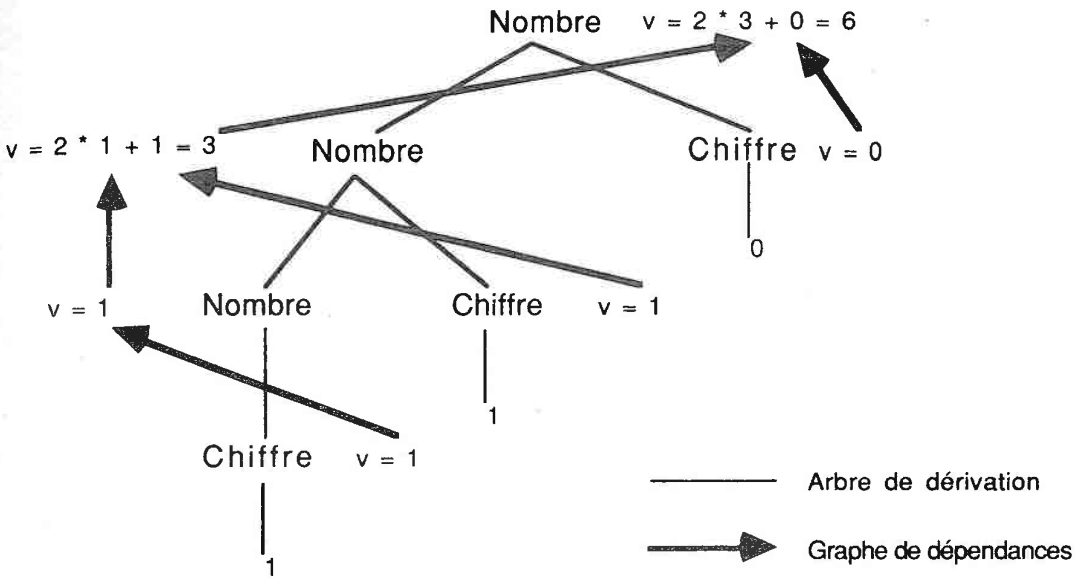


Figure 3.4 "L'arbre de dérivation attribué et le graphe de dépendances pour le nombre binaire **110**".

La Grammaire 3.1 n'est pas la seule GA à permettre d'obtenir la valeur décimale associée aux nombres binaires, comme nous le montre la Grammaire 3.2.

Grammaire 3.2. [Reps 1983]

1) Axiome → Nombre
   Axiome.valeur = Nombre.valeur;
   Nombre.position = 0;

2) Nombre → Nombre Chiffre
   Nombre1.valeur = Nombre2.valeur + Chiffre.valeur;
   Nombre2.position = Nombre1.position + 1;
   Chiffre.position = Nombre1.position;

3) Nombre → Chiffre
   Nombre.valeur = Chiffre.valeur;
   Chiffre.position = Nombre1.position;

4) Chiffre → 0
Chiffre.valeur = 0;

5) Chiffre → 1
Chiffre.valeur = 2 ** Chiffre.position;

Dans la Grammaire 3.2, les non-terminaux Axiome, Nombre et Chiffre possèdent chacun un attribut synthétisé "valeur"; Nombre et Chiffre possèdent aussi un attribut hérité nommé "position". L'attribut "valeur" d'un non-terminal X représente la valeur du sous-arbre dont la racine est X; l'attribut "position" d'un non-terminal X représente la "position binaire" du chiffre le plus à droite dans le sous-arbre dont la racine est X. La valeur décimale d'un nombre binaire est donc représentée par l'attribut "valeur" du symbole Axiome. La figure 3.5 montre comment obtenir, à partir de cette grammaire, la valeur décimale associée au nombre binaire **110**.

Remarquons que l'on a rajouté une nouvelle production, Axiome → Nombre, afin de pouvoir donner une valeur initiale à l'attribut hérité Nombre.position.

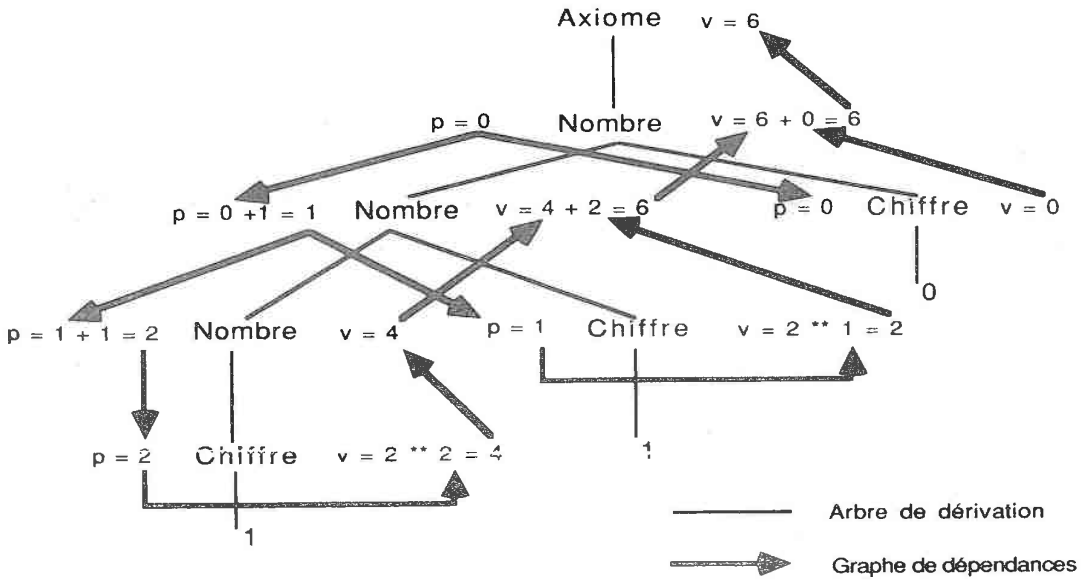


Figure 3.5 "L'arbre de dérivation attribué et le graphe de dépendances pour le nombre binaire **110**".

2.22 Les vérifications sémantiques des programmes.

Dans cette section nous montrerons comment utiliser les GA pour effectuer un certain nombre de vérifications sémantiques sur des programmes générés par la grammaire suivante :

Grammaire 3.3. "Le langage de programmation EXEMPLE".

1) Programme → Block
2) Block → Decl_liste "BEGIN" Inst_liste "END"
3) Decl_liste → "VAR" ident
4) Decl_liste → "VAR" ident Decl_liste
5) Inst_liste → Instruction
6) Inst_liste → Instruction Inst_liste
7) Instruction → ident " :=" exp
8) Instruction → "IF" cond "THEN" Instruction
9) Instruction → "WHILE" cond "DO" Instruction

NOTA : Pour des questions de facilité les productions associées aux non-terminaux exp et cond ne seront pas prises en compte. Nous nous passerons aussi des considérations d'ordre lexicographique associées au terminal générique ident.

Pour illustrer quelles sont les vérifications de sémantique statique auxquelles nous nous intéressons, considérons le programme suivant :

```
VAR J
BEGIN
    WHILE cond DO
        K := 0
END
```

Avec un programme de ce type, nous voudrions disposer d'un outil nous signalant

que la variable "J" n'a jamais été utilisée et que la variable "K" n'a pas été déclarée. Un tel outil peut facilement être construit à l'aide d'une GA, comme nous le montre la Grammaire 3.4.

Grammaire 3.4. "Vérification de la sémantique statique des programmes".

1) Programme → Block
   Programme.tsym = Block.tsym;

2) Block → Decl_liste "BEGIN" Inst_liste "END"
   Block.tsym = TabSym (Decl_liste.decl, Inst_liste.util);
   Inst_Liste.decl = Decl_Liste.decl;
   ASSERT : Decl_liste.decl = Inst_liste.util;

3) Decl_liste → "VAR" ident
   Decl_liste.decl = {ident.valeur};

4) Decl_liste → "VAR" ident Decl_liste
   ASSERT : {id.valeur} $\notin$ Decl_liste1.decl;
   Decl_liste1.decl = Decl_liste2.decl $\cup$ {ident.valeur};

5) Inst_liste → Instruction
   Instruction.decl = Inst_liste.decl;
   Inst_liste.util = Instruction.util;

6) Inst_liste → Instruction Inst_liste
   Instruction.decl = Inst_liste1.decl;
   Inst_liste2.decl = Inst_liste1.decl;
   Inst_liste1.util = Instruction.util $\cup$ Inst_liste2.util;

7) Instruction → ident " :=" exp
   Instruction.util = {ident.valeur};
   ASSERT : {ident.value} $\in$ Instruction.decl;

8) Instruction → "IF" cond "THEN" Instruction
Instruction1.util = Instruction2.util;
Instruction2.decl = Instruction1.decl;

9) Instruction → "WHILE" cond "DO" Instruction
Instruction1.util = Instruction2.util;
Instruction2.decl = Instruction1.decl;

Dans la Grammaire 3.4 nous avons utilisé la notation suivante :

- L'attribut "tsym" est un attribut synthétisé pour les non-terminaux "Block" et "Programme". Cet attribut représente l'information que l'on trouve généralement dans les tables des symboles. La fonction "TabSym" est supposée construire une de ces tables.

- L'attribut "decl" est un attribut hérité pour le non-terminal "Inst_liste" et un attribut synthétisé pour le non-terminal "Decl_liste". La valeur de cet attribut représente l'ensemble des identificateurs ayant été déclarés dans un programme.

- L'attribut "util" est un attribut synthétisé pour les non-terminaux Inst_liste et Instruction. La valeur de cet attribut représente l'ensemble des identificateurs utilisés dans les instructions d'un programme.

- L'attribut "valeur" est un attribut synthétisé pour le terminal générique "ident". La valeur de cet attribut représente la chaîne de caractères, lexicographiquement parlant, associée au symbole "ident".

- ASSERT représente une assertion devant être vérifiée. Si l'assertion concernée n'est pas vérifiée, c'est que l'on a détecté la violation d'une contrainte.

La Grammaire 3.4 permet de vérifier que tous les identificateurs apparaissant dans un programme ont été déclarés, qu'ils ont été utilisés au moins une fois dans le programme, et qu'aucun n' a été déclaré plusieurs fois.

2.3 L'UTILISATION DES GRAMMAIRES ATTRIBUEES DANS LES ENVIRONNEMENTS D'EDITION STRUCTUREE.

Dans cette section nous aborderons l'utilisation des grammaires attribuées dans des éditeurs dirigés par la syntaxe, c'est-à-dire, des éditeurs où la création d'un programme implique une croissance graduelle de l'arbre de dérivation.

Remarquons que la notion de croissance graduelle d'un arbre de dérivation signifie que celui-ci peut avoir des symboles non-terminaux au niveau de feuilles (i.e. des symboles non-terminaux non développés). Ceci pose un problème car nous n'avons aucun moyen de donner des valeurs aux attributs synthétisés de ces symboles, voire à aucun de leurs successeurs.

Pour éviter ce problème, il suffit d'introduire dans la grammaire la notion de *production complémentaire*, X → ¥, pour chaque symbole non-terminal X. Le symbole ¥ signifie non développé, et les équations sémantiques de la production complémentaire définissent les valeurs des attributs synthétisés de X [Reps 1983].

Si nous modifions la Grammaire 3.4, en introduisant les productions complémentaires, nous obtiendrons la grammaire suivante :

Grammaire 3.5. "La grammaire du langage EXEMPLE avec les productions complémentaires".

1) Programme → ¥
2) Programme → Block
3) Block → ¥
4) Block → Decl_liste "BEGIN" Inst_liste "END"
5) Decl_liste → ¥
6) Decl_liste → "VAR" ident
7) Decl_liste → "VAR" ident Decl_liste

8) Inst_liste → ¥

9) Inst_liste → Instruction

10) Inst_liste → Instruction Inst_liste

11) Instruction → ¥

12) Instruction → ident " :=" exp

13) Instruction → "IF" cond "THEN" Instruction

14) Instruction → "WHILE" cond "DO" Instruction

2.31 Vérification sémantique des programmes dans un environnement d'édition structurée.

Pour réaliser les vérifications de sémantique statique des programmes dans un environnement d'édition structurée, il suffit d'insérer dans la Grammaire 3.4, les productions complémentaires et les équations sémantiques qui leur sont associées, comme nous le montre la Grammaire 3.6.

Grammaire 3.6. "Vérification de la sémantique statique des programmes dans un environnement d'édition structurée".

1) Programme → ¥
   Programme.tsym = {∅};

2) Programme → Block
   Programme.tsym = Block.tsym;

3) Block → ¥
   Block.tsym = {∅};

4) Block → Decl_liste "BEGIN" Inst_liste "END"
   Block.tsym = TabSym (Decl_liste.decl, Inst_liste.util);
   Inst_Liste.decl = Decl_Liste.decl;
   ASSERT : Decl_liste.decl = Inst_liste.util;

5) Decl_liste → ¥
Decl_liste.decl = {∅};

6) Decl_liste → "VAR" ident
Decl_liste.decl = {ident.valeur};

7) Decl_liste → "VAR" ident Decl_liste
ASSERT : {id.valeur} ∉ Decl_liste1.decl;
Decl_liste1.decl = Decl_liste2.decl ∪ {ident.valeur};

8) Inst_liste → ¥
Inst_liste.util = {∅};

9) Inst_liste → Instruction
Instruction.decl = Inst_liste.decl;
Inst_liste.util = Instruction.util;

10) Inst_liste → Instruction Inst_liste
Instruction.decl = Inst_liste1.decl;
Inst_liste2.decl = Inst_liste1.decl;
Inst_liste1.util = Instruction.util ∪ Inst_liste2.util;

11) Instruction → ¥
Instruction.util = {∅};

12) Instruction → ident " :=" exp
Instruction.util = {ident.valeur};
ASSERT : {ident.value} ∈ Instruction.decl;

13) Instruction → "IF" cond "THEN" Instruction
Instruction1.util = Instruction2.util;
Instruction2.decl = Instruction1.decl;

14) Instruction → "WHILE" cond "DO" Instruction
Instruction1.util = Instruction2.util;
Instruction2.decl = Instruction1.decl

2.4 L'EVALUATION DES GRAMMAIRES ATTRIBUEES.

2.41 Introduction.

L'évaluation d'attributs† est un processus par lequel on assigne des valeurs à toutes les instances d'attributs d'un arbre de dérivation. L'ordre dans lequel les instances d'attributs sont évaluées n'est pas imposé. La seule contrainte est que celles-ci ne sont évaluées que lorsque tous les arguments de l'équation sémantique les définissant ont été évalués. C'est-à-dire que si nous avons la production :

$p : X_0 \rightarrow X_1 X_2 ... X_n$

et une équation sémantique de la forme :

$S(X_0) = \varphi(I(X_0) \cup S(X_i) \cup S(X_0) \cup I(X_i)), \forall i = 1, ..., n$

l'instance d'attribut $X_0.\alpha \in S(X_0)$, n'est évaluée que lorsque toutes les instances d'attribut, $X_0.\beta \in I(X_0)$ et $X_i.\delta \in S(X_i)$, $\forall i = 1, ..., n$, dont elle dépend ont été évaluées.

En fait, lorsque nous appliquons l'évaluation d'attributs à un arbre de dérivation T, ses attributs sont évalués dans un ordre respectant l'ordre partiel induit par le graphe de dépendances DG_T. Ceci signifie que le processus d'évaluation d'attributs est lié au problème du tri topologique.

Plusieurs méthodes ont été suggérées pour réaliser cette évaluation, e.g. [Lewis 1974], [Bochman 1976], [Kennedy 1976], [Lorho 1977], [Kastens 1980], [Jourdan 1984].

† NOTA : Bien que les termes "attributs" et "instance d'attributs" soient fondamentalement différents, lorsque nous parlerons d'évaluation d'attributs nous les confondrons quelques fois. Il est entendu que lors de ce processus d'évaluation il s'agit de donner une valeur à toutes les instances d'attributs apparaissant dans un arbre de dérivation.

Certains évaluateurs, e.g. [Lewis 1974], [Bochman 1976], [Kennedy 1976], [Kastens 1980], essayent de déterminer un ordre d'évaluation de manière statique, c'est-à-dire que l'ordre d'évaluation n'est pas déterminé par la forme de l'arbre de dérivation mais il est défini à partir des dépendances DGp de la grammaire. Malheureusement, ces méthodes ont l'inconvénient de restreindre le type de grammaires attribuées qu'elles peuvent traiter.

D'autres évaluateurs construisent le graphe de dépendances DG_T et ils le parcourent ensuite, en suivant une stratégie "profondeur d'abord" [Lorho 1977]. Malheureusement, et étant donné que l'ordre dans lequel les attributs sont évalués dépend de la forme de T, on doit recalculer DG_T chaque fois que l'on modifie T.

Enfin, il existe d'autres évaluateurs qui incluent le flux de contrôle des équations sémantiques dans celui du processus d'évaluation, en transformant chaque attribut en une fonction. Ces méthodes simulent le parcours du graphe de dépendances DG_T, en utilisant la pile des appels entre les différentes fonctions. Mis à part le fait qu'ils ne restreignent pas le type de grammaires attribuées à traiter, ces évaluateurs possèdent l'avantage d'effectuer une évaluation par nécessité, c'est-à-dire que si l'on a une équation sémantique de la forme :

$X_i.\delta$ = Si cond alors $X_m.\alpha$ sinon $X_k.\beta$

une seule des instances d'attribut, $X_m.\alpha$ ou $X_k.\beta$, sera calculée.

Un autre avantage important de ce type de méthodes est leur simplicité. C'est pour quoi, nous avons choisi, dans un premier temps, d'implanter dans GEODE un évaluateur d'attributs de ce type, en utilisant l'algorithme proposé par [Jourdan 1984]. Dans les sections qui suivent nous verrons les principes fondamentaux de cet algorithme.

2.42 La transformation des grammaires attribuées en un ensemble de fonctions.

La méthode d'évaluation des attributs proposée par [Jourdan 1984], consiste à représenter chaque attribut, hérité ou synthétisé, par une fonction. Cette évaluation est alors faite à travers des appels récursifs entre les fonctions représentant les différentes instances

d'attributs de l'arbre de dérivation à évaluer. L'évaluation s'arrête lorsque toutes les instances d'attributs synthétisées de la racine d'un arbre T ont été évaluées. Ceci signifie que si une instance d'attribut quelconque n'est pas nécessaire pour calculer la valeur sémantique de T, elle ne sera pas évaluée. Autrement dit, cette méthode assume que DG_T est un graphe connexe.

Pour un attribut synthétisé, la fonction possède un seul argument : un nœud de l'arbre de dérivation. Le nœud est étiqueté par une production dont la partie gauche est le symbole non-terminal auquel est attaché cet attribut. La fonction retourne la valeur de l'attribut synthétisé du nœud, après l'avoir calculé (i.e. lors du premier appel) ou après avoir consulté sa valeur (i.e. lors des appels subséquents).

Pour un attribut hérité, la fonction possède trois arguments, un nœud de l'arbre de dérivation comme précédemment, le nœud père de ce nœud dans l'arbre de dérivation complet et la position qu'occupe le symbole auquel est attaché l'attribut dans la partie droite de la production étiquetant ce nœud.

Soit la production p : $X_0 \rightarrow X_1, X_2, \ldots, X_n$

Si l'on construit une fonction pour un attribut synthétisé $X_0.\alpha \in S(X_0)$ définit par l'équation sémantique $X_0.\alpha = \varphi (X_0.\beta_0, X_1.\beta_1, X_2.\beta_2, \ldots, X_n.\beta_m)$:

FONCTION $X_0.\alpha$ (nœud) ;
DEBUT
    SI nœud possède déjà une valeur pour l'attribut valeur ALORS
        retourner cette valeur
    SINON
        CAS règle (nœud) DE
            .
            .
            .
            p : $X_0.\alpha = \varphi$ (
                $X_k.\beta_i$ (fils(nœud, k)) ,

{ Si $X_k.\beta_i$ est synthétisé.
On définit (fils(0,nœud) = nœud }
.
.
.
$X_k.\beta_i$ (fils(k,nœud) ,
père(fils(k,nœud)) ,
instance(fils(k,nœud)))
{ Si $X_k.\beta_i$ est hérité }
.
.
.
)
FIN CAS ;
STOCKER $X_0.\alpha$ dans nœud ;
RETOURNER $X_0.\alpha$;
FIN;

Si l'on construit une fonction pour un attribut hérité $X_k.\alpha \in I(X_k)$ définit par l'équation sémantique $X_k.\alpha = \varphi (X_0.\beta_0, X_1.\beta_1, X_2.\beta_2, \ldots , X_n.\beta_m)$:

FONCTION $X_k.\alpha$ (nœud, npère, instance) ;
DEBUT
SI nœud possède déjà une valeur pour l'attribut valeur ALORS
retourner cette valeur
SINON
FAIRE nœud := npère;
CAS [règle (nœud), instance] DE
.
.
.

[p, j]: $X_k.\alpha = \varphi$ (

$X_k.\beta_i$ (fils(k,nœud)) ,

{ Si $X_k.\beta_i$ est synthétisé.

On définit (fils(0,nœud) = nœud }

.

.

.

$X_k.\beta_i$ (fils(k,nœud) ,

père(fils(k,nœud)) ,

instance(fils(k,nœud)))

{ Si $X_k.\beta_i$ est hérité }

.

.

.

)

FIN CAS ;

STOCKER $X_k.\alpha$ dans nœud ;

RETOURNER $X_k.\alpha$;

FIN;

Afin de mieux comprendre la méthode reconsidérons la Grammaire 3.1, et construisons les fonctions représentant ses attributs : (Rappelons que cette grammaire ne contient que des attributs synthétisés)

Grammaire 3.1. [Knuth 68]

1) Nombre → Nombre Chiffre

Nombre1.valeur = 2 * Nombre2.valeur + Chiffre.valeur

2) Nombre → Chiffre

Nombre.valeur = Chiffre.valeur

3) Chiffre → 0

Chiffre.valeur = 0

4) Chiffre → 1
Chiffre.valeur = 1

Les fonctions associées aux attributs de cette grammaire seraient donc :

```
FONCTION Nombre.valeur (nœud) : ENTIER;
DEBUT
        SI nœud possède déjà une valeur pour l'attribut valeur ALORS
           retourner cette valeur
        SINON
           CAS règle(nœud) DE
              1 : Nombre.valeur := 2 * Nombre.valeur (fils (1, nœud)) +
                                   Chiffre.valeur (fils (2, nœud));
              2 : Nombre.valeur := Chiffre.valeur (fils (1, nœud))
           FIN
FIN;

FONCTION Chiffre.valeur (nœud) : ENTIER;
DEBUT
        SI nœud possède déjà une valeur pour l'attribut valeur ALORS
           retourner cette valeur
        SINON
           CAS règle(nœud) DE
              3 : Chiffre.valeur := 0;
              4 : Chiffre.valeur := 1
           FIN
FIN;
```

La fonction règle(nœud) retourne le numéro de la production associée au nœud passé en paramètre; la fonction (fils (i, nœud)) retourne l'i-ème fils du nœud passé en paramètre, si i = 0, la fonction retourne le nœud lui-même. L'évaluation d'un arbre de dérivation est donc réalisée lors des appels entre les différentes fonctions représentant les attributs de la grammaire.

Considérons à nouveau la Grammaire 3.2, et voyons comment construire les fonctions associées à ses attributs :

Grammaire 3.2. [Reps 1983]

1) Axiome → Nombre
   Axiome.valeur = Nombre.valeur
   Nombre.position = 0

2) Nombre → Nombre Chiffre
   Nombre1.valeur = Nombre2.valeur + Chiffre.valeur
   Nombre2.position = Nombre1.position + 1
   Chiffre.position = Nombre1.position

3) Nombre → Chiffre
   Nombre.valeur = Chiffre.valeur
   Chiffre.position = Nombre1.position

4) Chiffre → 0
   Chiffre.valeur = 0

5) Chiffre → 1
   Chiffre.valeur = 2 ** Chiffre.position

Les fonctions associées aux attributs de cette grammaire seraient donc :

```
FONCTION Axiome.valeur (nœud) : ENTIER;
DEBUT
        SI nœud possède déjà une valeur pour l'attribut valeur ALORS
           retourner cette valeur
        SINON
           CAS règle(nœud) DE
              1 : Axiome.valeur  := Nombre.valeur (fils (1, nœud))
           FIN
FIN;
```

```
FONCTION Nombre.valeur (nœud) : ENTIER;
DEBUT
        SI nœud possède déjà une valeur pour l'attribut valeur ALORS
           retourner cette valeur
        SINON
           CAS règle(nœud) DE
              2 : Nombre.valeur  := Nombre.valeur (fils (1, nœud)) +
                                    Chiffre.valeur (fils (2, nœud));
              3 : Nombre.valeur  := Chiffre.valeur (fils (1, nœud))
           FIN
FIN;

FONCTION Chiffre.valeur (nœud) : ENTIER;
DEBUT
        SI nœud possède déjà une valeur pour l'attribut valeur ALORS
           retourner cette valeur
        SINON
           CAS règle(nœud) DE
              4 : Chiffre.valeur  := 0;
              5 : Chiffre.valeur  := 2 ** (Chiffre.position (fils(0, nœud),
                                                             père(fils(0, nœud)),
                                                             instance(fils(0, nœud))))
           FIN
FIN;

FONCTION Nombre.position (nœud, fnœud, instance) : ENTIER;
DEBUT
        SI nœud possède déjà une valeur pour l'attribut position ALORS
           retourner cette valeur
        SINON
           CAS règle(nœud) DE
              1 : CAS instance DE
                     1 : Nombre.position  := 0
```

```
                FIN;
            2 : CAS instance DE
                2 : Nombre.position := Nombre.position (fils(0, fnœud),
                                                        père(fils(0, fnœud)),
                                                        instance(fils(0, fnœud))) + 1;
                FIN
        FIN
FIN;

FONCTION Chiffre.position (nœud, fnœud, instance) : ENTIER;
DEBUT
    SI nœud possède déjà une valeur pour l'attribut position ALORS
       retourner cette valeur
    SINON
       CAS  règle (fnœud) DE
          2 : CAS instance DE
              1 : Chiffre.position := Nombre.position (fils(0, fnœud),
                                                       père(fils(0, fnœud)),
                                                       instance(fils(0, fnœud)))
              FIN;
          3 : CAS instance DE
              1 : Chiffre.position := Nombre.position (fils(0, fnœud),
                                                       père(fils(0, fnœud)),
                                                       instance(fils(0, fnœud)))
              FIN
        FIN
FIN;
```

La fonction père(nœud) retourne le nœud père du nœud passé en paramètre; l'appel instance(nœud) retourne l'instance d'attribut hérité devant être calculée. Rappelons que du fait qu'un symbole X peut apparaître plusieurs fois en partie droite d'une production, il est nécessaire, dans une équation sémantique, de distinguer laquelle des occurrences de X possède l'instance d'attribut hérité à calculer.

Il n'est pas difficile de voir que le temps de traitement de cette méthode est optimal, car on n'évalue que les attributs strictement nécessaires pour calculer la valeur sémantique de l'arbre de dérivation T [Jourdan 1984]. Remarquons aussi que pendant l'évaluation des attributs, la pile des appels entre les différentes fonctions représente le graphe de dépendances DG_T, comme nous le montre la figure 3.6. (Les nœuds de l'arbre 3.5 ont été numérotés en suivant une stratégie "largeur d'abord")

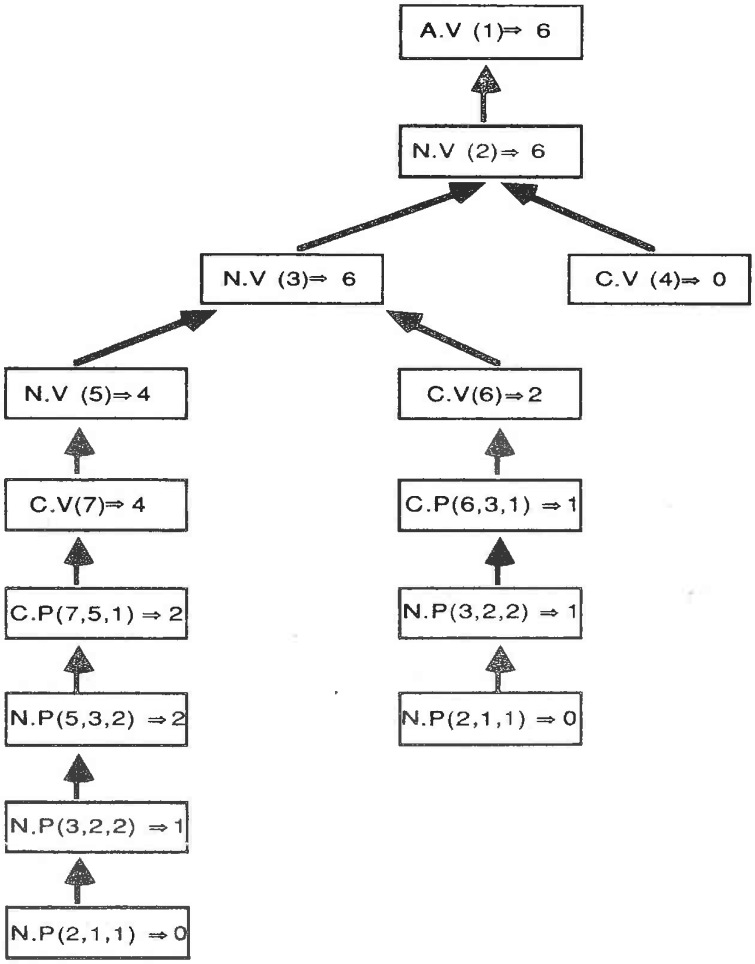


Figure 3.6 "Le graphe d'appels des fonctions réalisant l'évaluation d'attributs sur l'arbre de dérivation de la figure 3.5".

La notation utilisée dans la figure 3.6 est la suivante :

- Pour les attributs synthétisés :

X.V(1) → 6; signifie que la fonction représentant l'instance d'attribut X.V reçoit comme paramètre le nœud 1 et qu'une fois l'évaluation réalisée, la fonction renvoie comme résultat le numéro 6.

- Pour les attributs hérités :

X.V(2,1,1) → 0; signifie que la fonction représentant l'instance d'attribut X.V reçoit comme paramètres le nœud 2, le nœud 1 (i.e. le père du nœud 2) et le numéro 1 qui indique qu'il s'agit de calculer l'instance d'attribut associée à la première instance du symbole X. Une fois l'évaluation faite, la fonction retourne le résultat 0.

2.43 La notion d'évaluation incrémentale.

Dans un environnement d'édition structurée, modifier un programme implique la restructuration de l'arbre de dérivation à travers l'élimination ou la greffe d'un ou plusieurs sous-arbres. Ce processus d'élimination et/ou de greffe sera appelée *remplacement de sous-arbre.*

Lorsque l'on effectue un remplacement de sous-arbre il est fréquent que plusieurs instances d'attributs aient besoin d'une mise à jour, notamment celles du sous-arbre remplacé, ainsi que toutes les instances d'attributs les succédant. On peut effectuer cette mise à jour de deux manières :

- Soit, on relance l'évaluation d'attributs sur tout l'arbre de dérivation, ce que l'on appellera *évaluation globale*;
- Soit, on ne réévalue que les instances d'attributs touchées par le remplacement de sous-arbre, ce que l'on appellera *évaluation incrémentale.*

Il est évident que dans le cadre d'un éditeur structuré, où les remplacements de sous-arbre peuvent être très nombreux, la méthode d'évaluation incrémentale s'avère la plus

intéressante.

Afin de mieux comprendre la manière dont un évaluateur incrémental fonctionne, considérons un remplacement effectué sur un arbre de dérivation issu de la grammaire 3.6.

Lorsque le remplacement de sous-arbre illustré par la figure 3.7 est effectué, (voir figure 3.8) les instances d'attributs du type "Instruction.util" doivent être mises à jour. En effet, après ce remplacement il n'y a plus aucun identificateur utilisé. Par contre, les instances d'attributs du type "Instruction.decl" ne doivent pas être réévaluées, car elles conservent les mêmes valeurs qu'avant le remplacement de sous-arbre : les identificateurs déclarés sont toujours "A" et "B".

Le but de l'évaluation incrémentale d'attributs est donc de réduire au maximum le nombre d'instances d'attributs réévaluées après avoir fait un remplacement de sous-arbre.

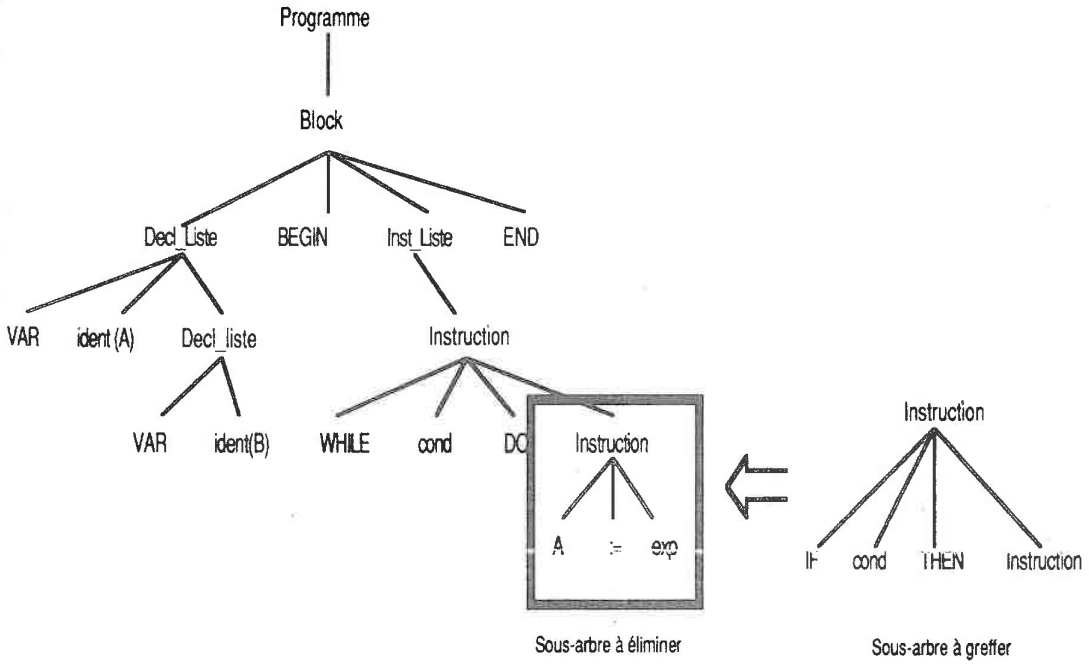


Figure 3.7 "Un remplacement de sous-arbre".

D'avantage de précisions sur l'évaluation incrémentale d'attributs seront données dans le chapitre IV.

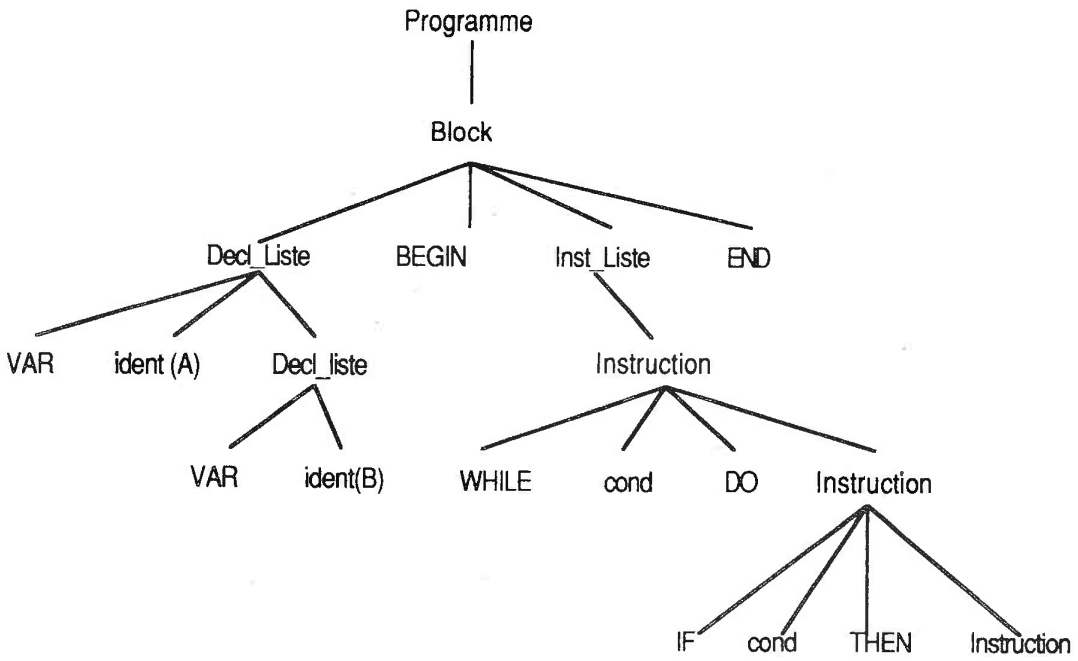


Figure 3.8 "L'arbre de dérivation résultant du remplacement de sous-arbre".

Chapitre IV.

LE SYSTEME GEODE ET LES GRAMMAIRES ATTRIBUEES.

1. INTRODUCTION.

Dans le chapitre précédent, nous avons présenté un mécanisme nous permettant de définir syntaxiquement et sémantiquement un langage : les grammaires attribuées. Dans le chapitre IV nous montrons comment ce mécanisme est utilisé par le système GEODE lors de la construction d'outils d'un environnement de programmation intégré.

La première partie est consacrée à l'évaluation incrémentale d'attributs dans GEODE; nous présentons l'algorithme d'évaluation incrémentale implanté dans GEODE et nous réalisons une évaluation de sa complexité. La deuxième partie présente la construction d'un outil d'édition structurée et d'un outil de documentation automatique des programmes. Nous analysons aussi la construction éventuelle d'autres types d'outils.

2. L'EVALUATION D'ATTRIBUTS DANS LE SYSTEME GEODE.

Nous avons vu dans le chapitre III un certain nombre de méthodes nous permettant d'évaluer un arbre de dérivation attribué. Dans le système GEODE nous voulions un évaluateur d'attributs ayant les caractéristiques suivantes :

- Nous ne voulions pas avoir à restreindre le type de grammaires attribuées pouvant être traitées par le système. Il n'était donc pas question d'utiliser des méthodes définissant un ordre d'évaluation de manière statique [Lewis 1974], [Kennedy 1976], [Kastens 1980];
- Il est évident que l'algorithme d'évaluation choisi devait avoir la capacité de réaliser l'évaluation incrémentale, mais nous voulions aussi disposer d'un algorithme d'évaluation globale simple et rapide. Rappelons que GEODE est un système

générateur d'environnements de programmation et que s'il est vrai que pour l'éditeur structuré nous avions besoin de l'évaluation incrémentale, rien ne nous permettait d'affirmer que ce type d'évaluation était le meilleur quel que fût l'outil de l'environnement à générer;
- Nous voulions aussi, un algorithme simple et facile à mettre en œuvre.

Ainsi, nous nous sommes proposés de faire un nouvel algorithme d'évaluation† en nous basant sur la méthode de [Jourdan 84] et en y apportant des modifications nous permettant de réaliser l'évaluation incrémentale. Ceci sera traité dans la section suivante.

2.1 PRESENTATION DE LA METHODE D'EVALUATION D'ATTRIBUTS DANS GEODE.

L'évaluation incrémentale en GEODE est réalisée de la manière suivante :

- Lorsqu'on construit le programme on construit, de manière dynamique, le graphe de dépendances associé à l'arbre de dérivation (i.e. le graphe de dépendances est construit au fur et à mesure des appels entre les différentes fonctions);
- Lorsqu'il y a une modification de cet arbre (i.e. un remplacement de sous-arbre), on effectue les actions suivantes :
    - tous les attributs du sous-arbre reçoivent la valeur spéciale "null", à l'exception des attributs hérités de sa racine;
    - tous les attributs hérités de la racine du sous-arbre reçoivent les valeurs qui correspondent d'après le graphe de dépendances;
    - lancement de l'évaluation globale sur les attributs synthétisés -notés S(Sr)- de la racine du sous-arbre;
    - propagation dans l'arbre des changements introduits par le remplacement du sous-arbre. La propagation se termine lorsque la nouvelle valeur d'une instance d'attribut successeur de S(Sr) est égale à celle qui était stockée dans l'arbre, ou bien, lorsque toutes les instances d'attributs synthétisés de la racine de l'arbre ont été réévaluées.

Situons nous d'abord dans le contexte où l'algorithme d'évaluation d'attributs est utilisé [Reps 1983]. Comme nous avons vu précédemment, la modification d'un programme consiste à restructurer l'arbre de dérivation qui le représente à travers des greffes et des élagages de sous-arbres (i.e. remplacements de sous-arbre). Soit T un arbre de dérivation et U un sous-arbre de T où la racine est le nœud r etiquetté par X. U est élagué de T en enlevant le sous-arbre dont la racine est r. Soit U' un sous-arbre où la racine est le nœud r' etiquetté par X. U' est greffé sur T en remplaçant le nœud r etiquetté par X et en assignant les valeurs des attributs hérités de r aux instances d'attributs hérités de r'. (Voir figure 4.1).

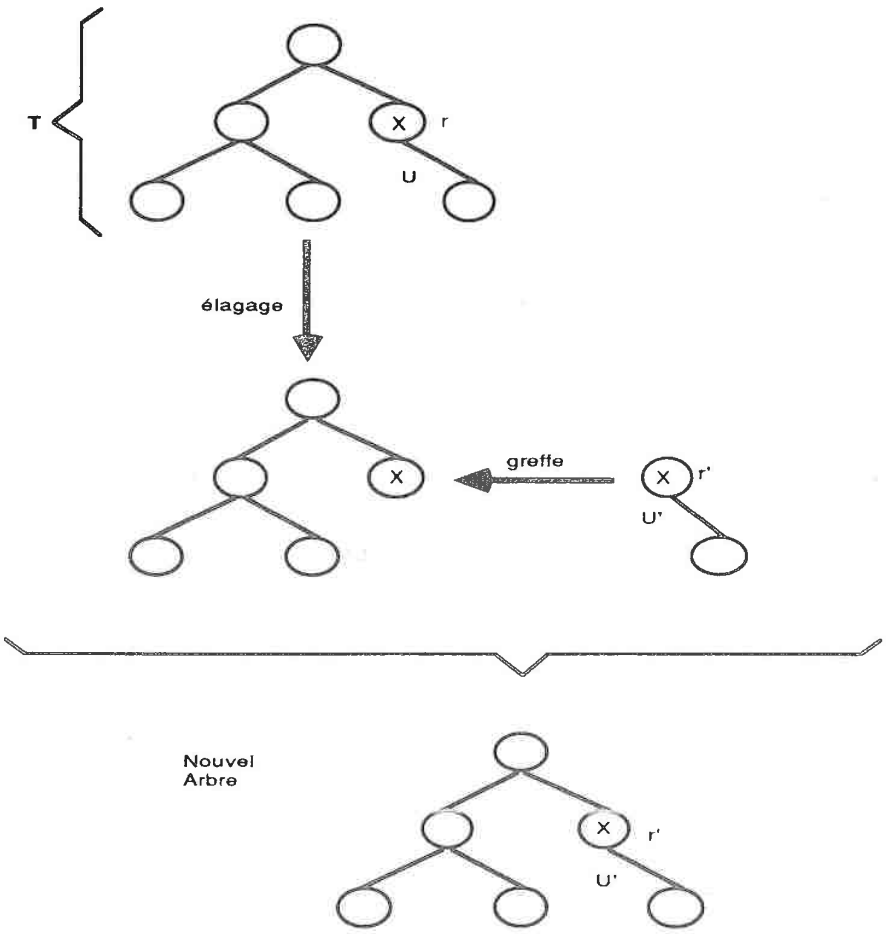


Figure 4.1 "La restructuration d'un arbre de dérivation par greffe et élagage".

La restructuration de l'arbre de dérivation T consiste donc à élaguer un sous-arbre U, à greffer un sous-arbre U' et à exécuter l'algorithme de propagation afin de rétablir la consistance de T.

En général, un arbre de dérivation où une greffe a été effectuée n'est pas consistant, c'est-à-dire que les valeurs de ses instances d'attributs ne correspondent pas aux équations sémantiques qui les définissent.

Pour obtenir un arbre de dérivation consistant, on pourrait relancer l'évaluation globale d'attributs à partir de la racine de cet arbre, mais le but est de ne réévaluer que les instances d'attributs touchées par la greffe réalisée. Il faut donc effectuer une évaluation incrémentale à partir du nœud où la greffe a été faite, en utilisant un algorithme capable de propager les modifications sans changer pour autant les instances d'attributs consistentes. Cet algorithme est connu sous le nom d'algorithme de propagation.

Regardons avec l'exemple traité dans les figures 3.7 et 3.8, comment fonctionne l'algorithme d'évaluation incrémentale implanté dans GEODE : La première étape de l'évaluation incrémentale consiste donc à lancer l'évaluation globale sur le sous-arbre venant d'être remplacé (voir figure 4.2).

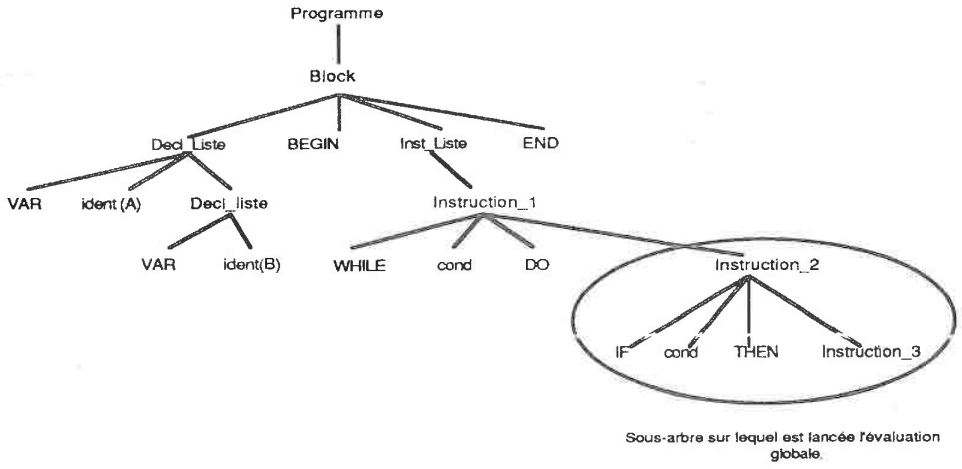


Figure 4.2 "La première étape de l'évaluation incrémentale d'attributs".

Le lancement de l'évaluation globale sur le sous-arbre a un double effet, d'abord mettre à jour le graphe de dépendances de l'arbre de dérivation complet (voir figures 4.3 et 4.4) et ensuite de rendre consistentes les instances d'attributs du sous-arbre. Cette première étape de l'évaluation incrémentale modifie les valeurs des instances d'attributs du sous-arbre, c'est-à-dire que :

- L'instance "Instruction_2.decl" possède la valeur {A, B}, car elle hérite directement la valeur de l'instance "Instruction_1.decl";
- Une fois cette première étape terminée, on obtient que :
  "Instruction_2.util" = {∅}, "Instruction_3.decl" = {A, B},
  "Instruction_3.util" = {∅};

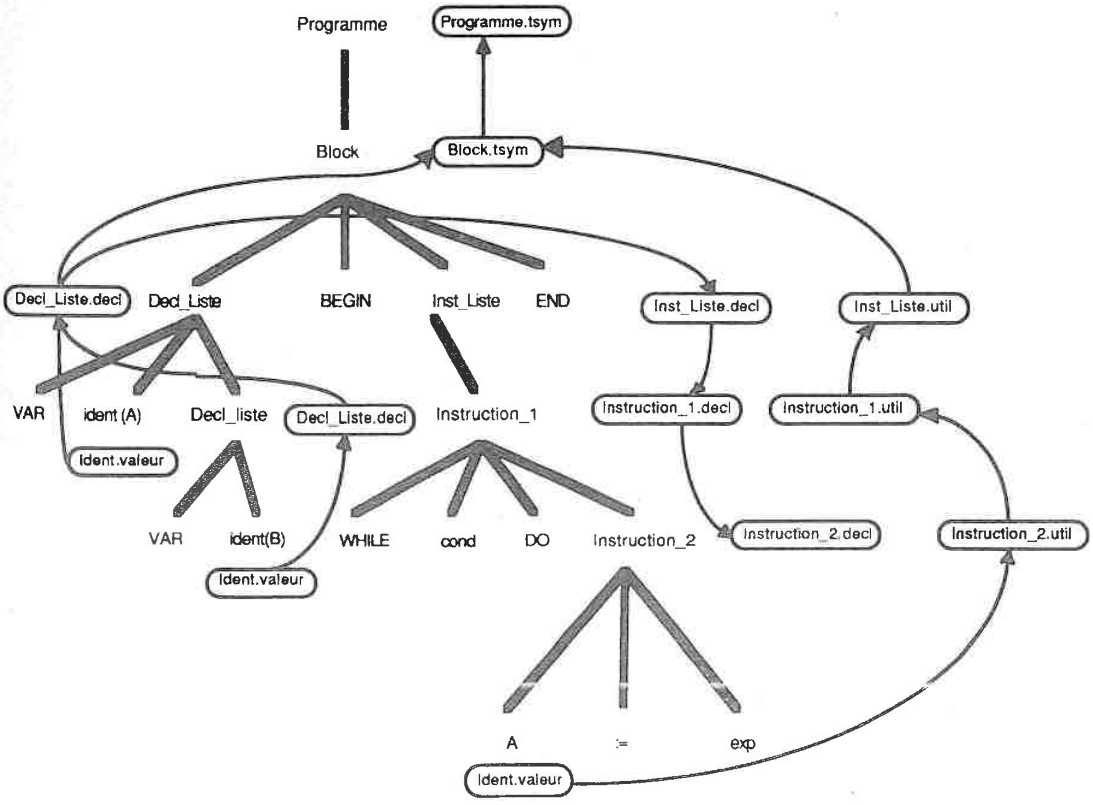


Figure 4.3 "Le graphe de dépendances DG_T avant le remplacement de sous-arbre".

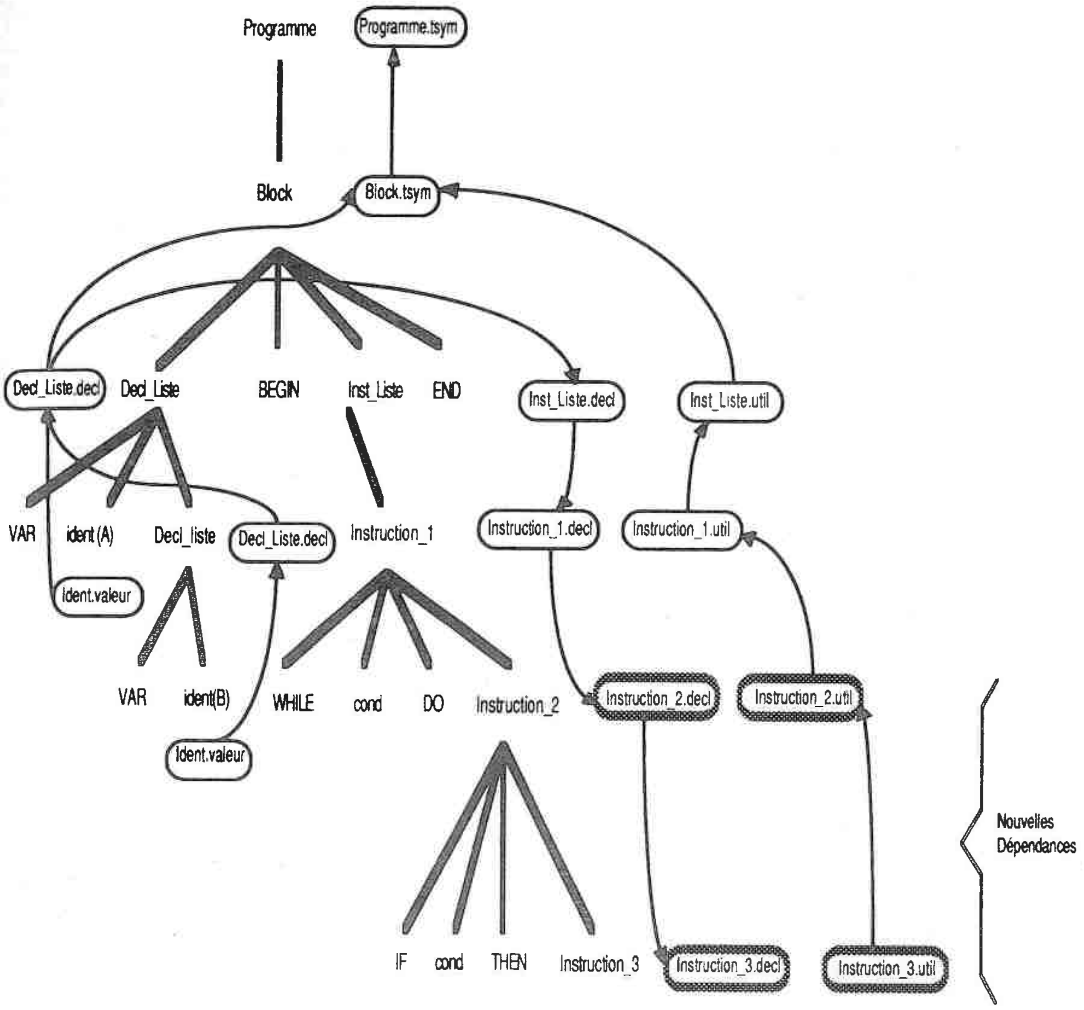


Figure 4.4 "Le graphe de dépendances DG_T après le remplacement de sous-arbre".

Les figures 4.3 et 4.4 montrent donc le graphe DG_T avant et après le remplacement de sous-arbre. Il est à remarquer que la nouvelle dépendance "Instruction_3.util" → "Instruction_2.util" apparaît lorsque nous lançons l'évaluation globale sur le sous-arbre ayant été greffé, car la DGp de la production associée au sous-arbre est :

DGp = ({Instruction_2.decl, Instruction_3.decl, Instruction_2.util, Instruction_3.util} ,
{ (Instruction_2.decl → Instruction_3.decl),
(Instruction3_util → Instruction_2.util) })

"Instruction_3.decl" faisant partie de l'ensemble de dépendances $DS_{pInstruction\_2.decl}$, et "Instruction_2.util" faisant partie de l'ensemble de dépendances $DS_{pInstruction\_3.util}$.

Quant à la dépendance " Instruction_2.decl" → "Instruction_3.decl", elle apparaît lors de la greffe du sous-arbre.

Une fois l'exécution de l'évaluation globale sur le sous-arbre terminée, nous lançons la deuxième étape de l'évaluation incrémentale. Cette deuxième étape consiste à propager l'information de sortie du sous-arbre (i.e. l'ensemble d'instances d'attribut synthétisées de la racine du sous-arbre), en suivant le graphe de dépendances construit durant la première étape. On utilise pour ceci l'algorithme de propagation qui sera présente dans la section suivante. Remarquons que, si au cours de la propagation de l'information, la nouvelle valeur d'une instance d'attribut est égale à son ancienne valeur, l'information n'a plus besoin d'être propagée et l'évaluation d'attributs est donc terminée. Si nous continuons avec notre exemple, nous constaterons que l'information est propagée comme nous le montre la figure 4.5.

Le problème maintenant, est de recalculer les instances d'attributs qui sont successeurs des instances d'attributs synthétisées de la racine du sous-arbre greffé. La solution est de parcourir le graphe de dépendances en faisant appel, chaque fois que l'on visite un nœud, à la fonction qui lui est associée. Il est à remarquer que lorsqu'on appelle la fonction associée à un nœud, on ne recalcule pas la valeur des successeurs; toutes les fonctions que celle-ci appelle ne feront que récupérer la valeur de l'instance d'attribut ayant été calculée lors de l'évaluation globale.

Ainsi, lorsque nous propageons l'information sur l'arbre de dérivation de la figure 4.5, nous faisons les appels suivants :

- On appelle la fonction Instruction_1.util(nœud), où nœud représente le symbole "Instruction_1". Cette fonction retourne la valeur {∅}. Cette valeur est calculée en faisant l'appel suivant :

1) Instruction_1.util = Instruction_2.util (fils(4, nœud));

- remarquons que l'appel Instruction_2.util (fils(4, nœud)) ne recalcule pas la valeur des instances d'attributs qu'il représente, il récupère, simplement, la valeur ayant été calculée lors de l'évaluation globale.

La propagation des valeurs des attributs continue de la même manière jusqu'au calcul de l'instance "Programme.tsym".

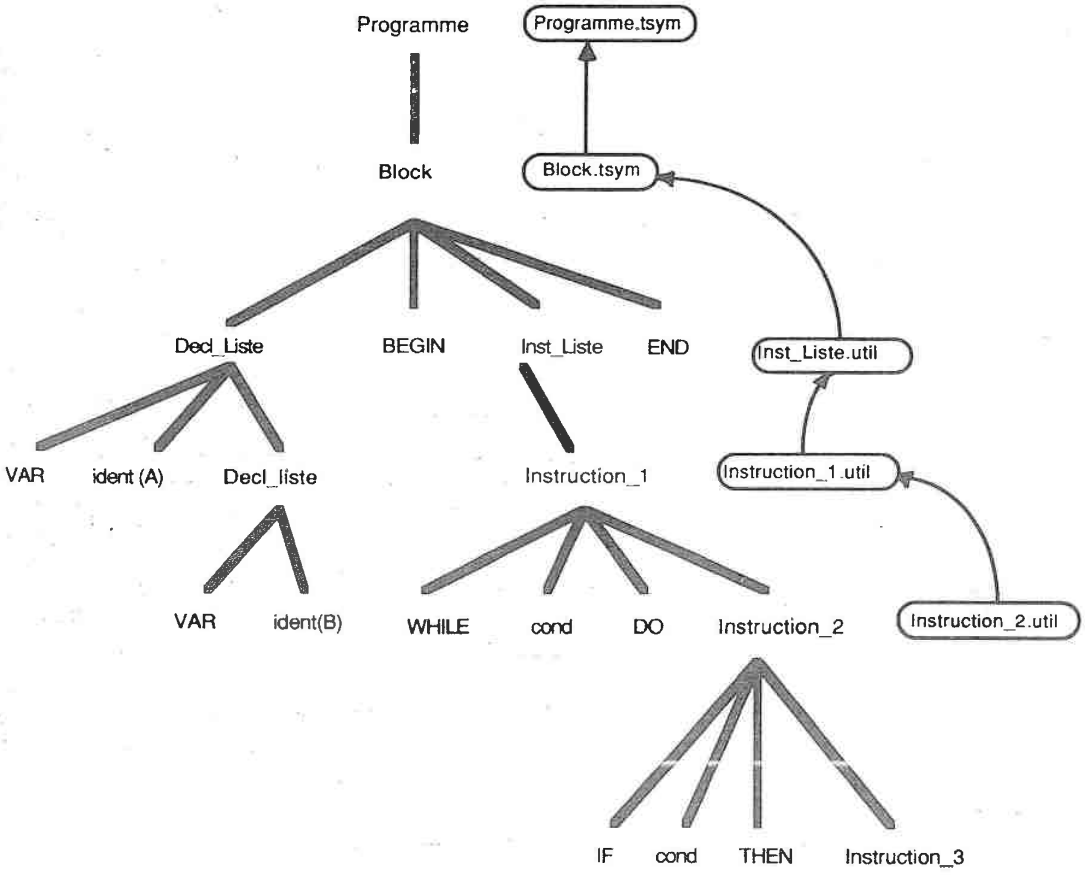


Figure 4.5 "La propagation de l'information lors de la deuxième étape de l'évaluation incrémentale".

2.2 L'ALGORITHME DE PROPAGATION.

Nous présentons ici l'algorithme de propagation qui est chargé de parcourir le graphe de dépendances de l'arbre de dérivation, de manière à ce que toutes les instances d'attributs affectées par une modification soient recalculées.

Un premier algorithme de propagation est dit l'algorithme de propagation naïve.

Algorithme 1. "La propagation naïve". [Reps 1983]

```
PROPAGATION (T, r);
Soit
   T =  un arbre de dérivation attribué ;
   r = un nœud de T contenant toutes les instances d'attributs inconsistantes de T ;
   S = un ensemble d'instances d'attributs ;
   β = une instance d'attribut ;
   AncValeur, NouValeur = des valeurs d'attributs ;
Début
   S ← l'ensemble d'instances d'attributs inconsistantes de r ;
   Tant que S ≠ {∅} faire
        sélectionner et enlever une instance d'attributs β de S ;
        AncValeur ← β ;
        évaluer β ;
        NouValeur ← β ;
        Si AncValeur ≠ NouValeur alors
             S ← S ∪ {successeurs de β} ;
        fin_Si ;
   fin_Tant que ;
Fin ;
```

Cet algorithme peut effectivement être utilisé pour propager des changements des instances d'attributs après un remplacement de sous-arbre. Malheureusement, le comportement de l'algorithme dépend de la manière dont les instances d'attributs sont

réévaluées. Dans certains cas l'algorithme possède un comportement non-linéaire par rapport aux nombre d'instances d'attributs recalculées, comme le montre l'exemple suivant :

Grammaire 4.0 † :

1) S → X S
   $X.c = S_2.a$
   $X.d = S_2.a + 1$
   $X.e = S_2.a + 2$
   $S_1.a = X.b + S_2.a$

2) S → s
   S.a = 1

3) S → t
   S.a = 2

4) X → x
   X.b = X.c + X.d + X.e

La figure 4.6 montre la forme générale des graphes de dépendances des arbres de dérivation issus de la grammaire 4.0.

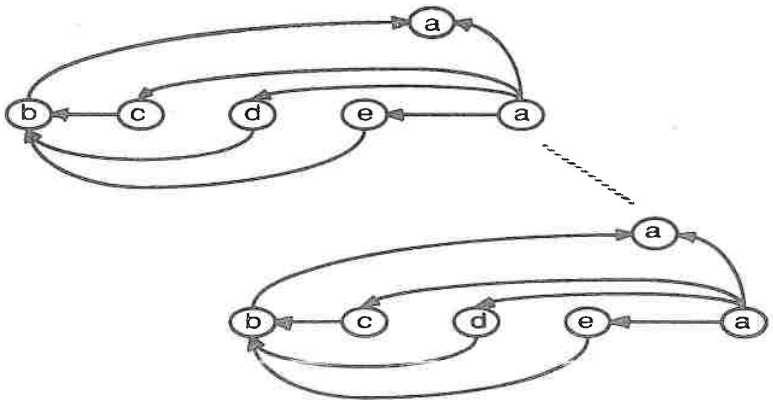


Figure 4.6 "La forme générale des graphes de dépendance des arbres de dérivation de la grammaire 4.0".

† Cette grammaire est une extension de celle utilisée par [Reps 1983b].

Pour analyser le comportement de l'algorithme 1, nous utiliserons l'arbre de dérivation suivant : (voir figure 4.7)

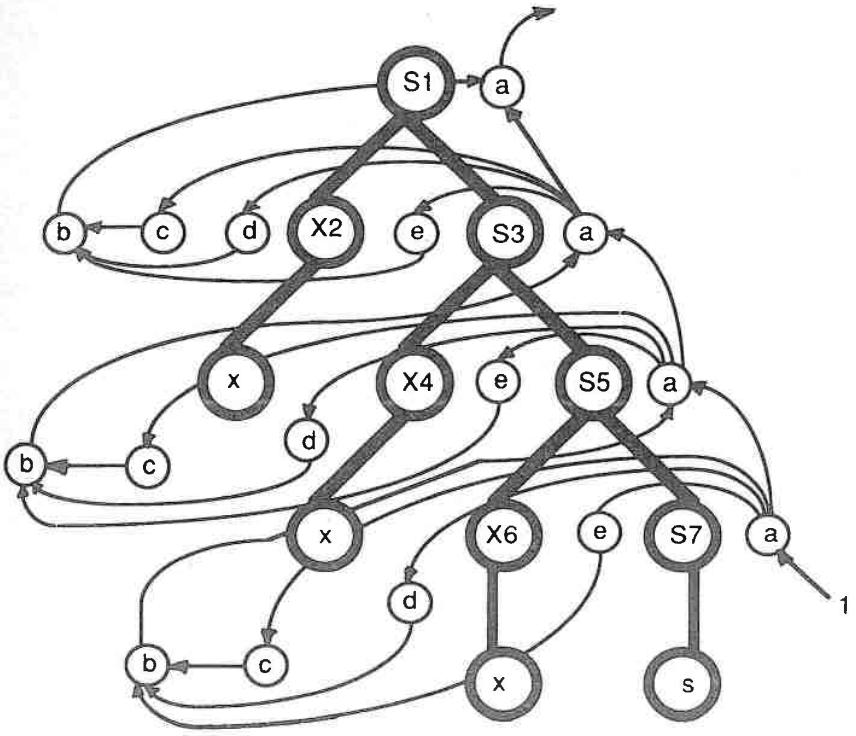


Figure 4.7 "Un arbre de dérivation pour la grammaire 4.0 et le graphe de dépendances associé".

NOTA : Les symboles syntaxiques de la figure 4.7 sont suivis d'un chiffre qui nous aidera par la suite à distinguer les différentes instances d'attributs de l'arbre. Ainsi "S1.b" notera l'instance d'attribut b associée au symbole non-terminal S1. Dans la figure 4.7 S1 représente la racine de l'arbre de dérivation.

Supposons maintenant que l'on effectue un remplacement de sous-arbre au niveau du symbole non-terminal S7. Le remplacement de sous-arbre consisterait à élaguer le sous-arbre correspondant à la production S $\rightarrow$ s et ensuite à greffer le sous-arbre correspondant à la production S $\rightarrow$ t. Ainsi, lorsque l'algorithme de propagation est exécuté, l'ensemble S contient l'instance d'attribut S7.a.

Lors du déroulement de l'algorithme de propagation, l'ensemble S va recevoir les instances d'attributs de S7.a. S contiendra donc S5.a, X6.e, X6.d et X6.c. Malheureusement, le comportement de l'algorithme 1 dépend de la manière dont l'ensemble S est manipulé.

Supposons d'abord que l'on privilégie les insertions des instances d'attributs de la forme S.a dans S. Ensuite, si l'on manipule l'ensemble S avec une politique FIFO (i.e. premier entré premier sorti), l'algorithme 1 parcourra le graphe de dépendances en largeur d'abord. Calculons, à partir de ceci la complexité de l'algorithme.

Si l'on considère que T(DGp) est le temps nécessaire pour réévaluer les instances d'attributs associées à une production p, nous obtiendrons que le temps de réévaluation dépend de la hauteur de l'arbre H(T), comme le montre la relation suivante :

T(réévaluation) = Si H(T) = 1 alors T(DGp)
Si H(T) = 2 alors T(DGp) + 2T(DGp)
Si H(T) = 3 alors T(DGp) + 2T(DGp) + 3T(DGp)

La figure 4.8 montre la réévaluation des instances d'attribut de l'arbre de la figure 4.7. Le nombre associé à chaque nœud du graphe de dépendances représente le nombre de fois que l'instance d'attribut correspondante a été réévaluée.

La figure 4.8 montre qu'il y a des instances d'attributs qui sont réévaluées plusieurs fois. Prenons par exemple l'instance d'attribut S5.a. Selon cette figure 4.8 l'instance est réévaluée 2 fois. En effet, elle est réévaluée une première fois au moment de réévaluer l'instance S7.a et puis une deuxième fois au moment de la réévaluation de l'instance X6.b.

$$T(\text{réévaluation}) = \sum_{i=1}^{H(T)} T(DG_p) * i = T(DG_p) * \sum_{i=1}^{H(T)} i = K * [\, H(T) * (H(T) + 1) \,] / 2 = O(H(T)^2)$$

Considérons maintenant le cas où l'ensemble S est manipulé selon une politique LIFO. Dans ce cas, le parcours du graphe de dépendances est fait en profondeur d'abord. Le

temps nécessaire pour réévaluer les instances d'attributs de l'arbre suit la relation suivante :

$$T(\text{réévaluation}) = \begin{cases} \text{Si } H(T) = 1 \text{ alors} \cong 2T(DGp) \\ \text{Si } H(T) = 2 \text{ alors} \cong 2T(DGp) + 8T(DGp) \\ \text{Si } H(T) = 3 \text{ alors} \cong 2T(DGp) + 8T(DGp) + 32T(DGp) \end{cases}$$

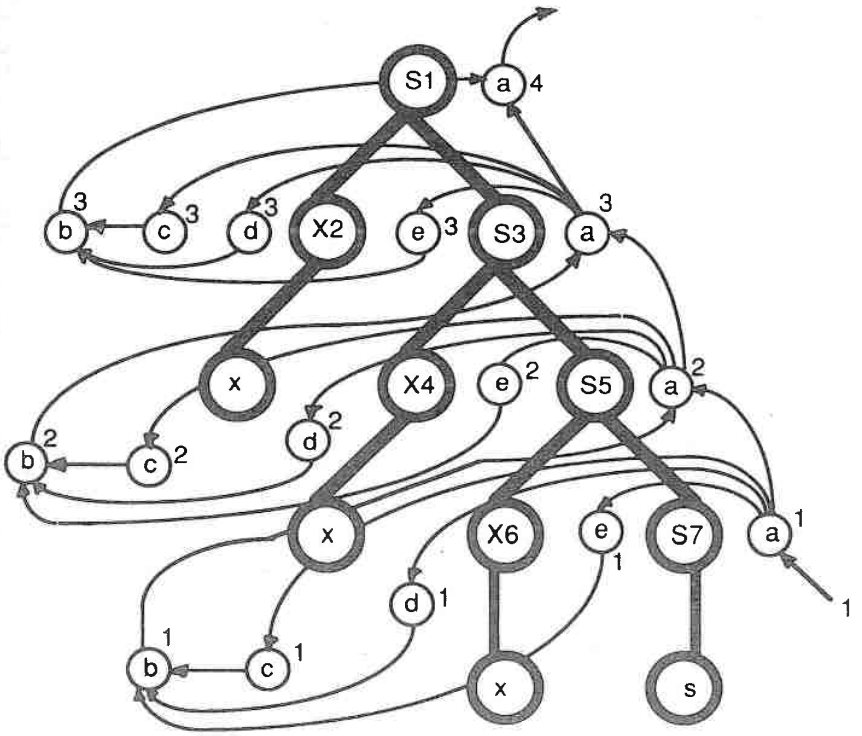


Figure 4.8 "La réévaluation des instances d'attributs d'après une manipulation FIFO de l'ensemble S de l'algorithme de propagation naïve".

La figure 4.9 montre la réévaluation des instances d'attribut de l'arbre de la figure 4.7.

Comme précédemment, il y a des instances d'attributs qui sont réévaluées plusieurs fois. Prenons par exemple l'instance d'attribut X4.b. D'après la figure 4.9 cette instance est réévaluée 12 fois. En effet, cette instance est réévaluée chaque fois que l'on réévalue un de ces ancêtres, à savoir X4.c, X4.d et X4.e. Etant donné que ces instances sont réévaluées 4

fois chacune, l'instance X4.b est donc réévaluée 12 fois. Un raisonnement similaire permet de vérifier les données de la figure 4.9.

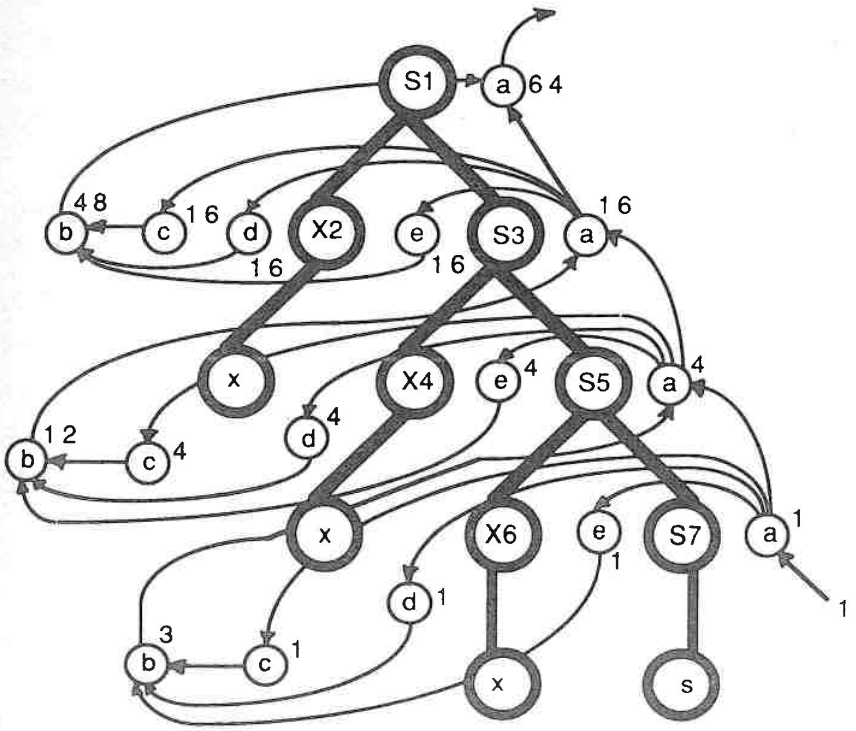


Figure 4.9 "La réévaluation des instances d'attributs d'après une manipulation LIFO de l'ensemble S de l'algorithme de propagation naïve".

$$T(\text{réévaluation}) = \sum_{i=0}^{H(T)-1} 2^{2i+1} * T(DG_p) = T(DG_p) * \sum_{i=0}^{H(T)-1} 2^{2i+1} = T(DG_p) * (2^{2H(T)} - 2) / 3$$

$$T(\text{réévaluation}) = T(DG_p) * 2^{2H(T)} / 3 - 2/3 = K1 * 2^{2H(T)} - K2 = O(2^{2H(T)})$$

Dans GEODE, nous désirions avoir un algorithme de propagation optimum, c'est-à-dire que une instance d'attribut n'est réévaluée que lorsque tous ses ancêtres dans le graphe de dépendances ont été réévalués. Autrement dit, l'algorithme de propagation doit

avoir un comportement linéaire par rapport au nombre d'instances d'attribut réévaluées. L'algorithme 1 n'as pas un comportement linéaire ce qui implique qu'il peut réévaluer plusieurs fois une même instance d'attribut, avant d'arriver à sa valeur définitive. Ceci explique le comportement quadratique de l'algorithme 1 lorsque S est géré d'après une politique FIFO, et sont comportement exponentiel lorsque S est géré selon une politique LIFO.

Avant d'analyser l'algorithme de propagation utilisé par GEODE, il est nécessaire de reconsidérer l'évaluation globale d'attributs. Comme nous l'avons mentionné précédemment, les attributs associés à un arbre T, doivent être réévalués dans un ordre respectant l'ordre partiel donné par le graphe de dépendances DG_T. L'évaluation d'attributs est donc un problème lié au tri topologique [Knuth 1968a]. Pour réaliser l'évaluation d'attributs on peut donc effectuer le tri topologique, pour ensuite évaluer les attributs dans l'ordre résultant [Lewis 1974], ou bien, on peut effectuer simultanément le tri et l'évaluation des attributs [Reps 1983], ce que l'on appellera *évaluation topologique*.

Voici l'algorithme d'évaluation topologique [Reps 1983]:

Algorithme 2. "L'évaluation topologique".

```
EVALUATION_TOPOLOGIQUE (T) ;
    Soit
        T = un arbre de dérivation attribué non-évalué ;
        G = un graphe dirigé ;
        S = un ensemble d'instances d'attributs ;
        β et χ = des instances d'attributs ;
    Début
        G ← DG_T ;
        S ← l'ensemble de nœuds de G ayant un nombre d'arcs d'entrée égal à zéro ;
        Tant que S ≠ ∅ faire
            sélectionner et enlever un nœud β de S ;
        (*) évaluer β ;
            Pour chaque χ étant successeur de β dans G faire
```

```
                enlever (β, χ) de G ;
                Si nombre-d'arcs-d'entrée (χ) = 0 alors
                    S ← S ∪ {χ} ;
                fin_Si ;
            fin_Pour ;
        fin_Tant que ;
Fin .
```

Par rapport à l'algorithme du tri topologique de Knuth, dans la ligne (*) au lieu d'assigner un numéro topologique, les attributs sont évalués. Bien entendu, cet algorithme suppose que G et donc DG_T est sans circuit. (i.e. la grammaire attribuée est non-circulaire). Si G possède un arc il existe au moins une instance d'attribut non-évaluée dont le nombre d'entrées est égale à zéro, donc S est non-vide. L'algorithme termine car G est un graphe connexe sans circuit, ce qui implique qu'à l'intérieur de la boucle, un arc est supprimé à chaque tour. A la fin donc de cette boucle G n'a plus d'arcs, S est vide et toutes les instances d'attributs ont été recalculées (i.e. chaque nœud du graphe a été visité une fois).

L'algorithme de propagation implanté dans GEODE doit être vu comme une généralisation de l'algorithme d'évaluation topologique.

Voici l'algorithme de propagation que nous implantons actuellement dans GEODE : (les commentaires sont précédés par "--")

```
PROPAGATION (X);
Soit
    Z = l'ensemble d'attributs synthétisés de X ;
-- Z est égale à S(X)
    S = {∅} ;
    G = un graphe dirigé ;
Début
    G = DG_T ;
-- Recherche des successeurs de S(X), ayant S(X) comme seuls ancêtres.
Tant que Z ≠ {∅} faire
```

```
            sélectionner et enlever une instance d'attribut α de Z ;
            Pour chaque β étant successeur de α dans G faire
                enlever (α, β) de G ;
                Si nombre-d'arcs-d'entrée (β) = 0 alors
                    S ← S ∪ {χ} ;
                fin_Si ;
            fin_Pour ;
        fin_Tant que ;
    -- S est égale à l'ensemble de successeurs de S(X), dont les nombre d'arcs d'entrée
    -- est égale à zéro.
    -- Evaluation Incrémentale. (Propagation)
        Tant que S ≠ {∅} faire
            sélectionner et enlever une instance d'attribut δ de S ;
            évaluer δ ;
            Si nouvelle_valeur (δ) ≠ ancienne_valeur (δ) alors
                Pour chaque ε étant successeur de δ dans G faire
                    enlever (δ, ε) de G ;
                    Si nombre-d'arcs-d'entrée (ε) = 0 alors
                        S ← S ∪ {ε} ;
                    fin_Si ;
                fin_Pour ;
            fin_Si ;
        fin_Tant que ;
    Fin .
```

Si nous analysons cet algorithme, nous constaterons qu'il constitué de deux parties :

Une première partie est consacrée à la recherche des instances d'attributs successeurs de S(X), ayant S(X) comme seuls ancêtres. Ces instances seront notées *succ S(X)*. Soit $\alpha \in$ S(X) et $\beta_i \in$ succ S(X), $\forall i$. Si nous enlevons tous les arcs de la forme (α, β_i) de G, alors le nombre d'arcs d'entrée de tous les β_i sera égal à zéro. A la fin de la première partie S contiendra donc toutes les instances d'attributs appartenant à succ S(X), ayant un nombre d'arcs d'entrée égal à zéro. Remarquons qu'en général, S n'est pas vide, car DG_T est censé

être un graphe connexe(1). Cette première partie termine, car le nombre d'instances d'attributs appartenant à succ S(X) est fini.

La deuxième partie de l'algorithme est chargée de propager les modifications. Cette propagation s'arrête de deux manières. Soit, lorsque la nouvelle valeur d'une instance d'attribut (i.e. celle qui vient d'être calculée) est égale à son ancienne valeur (i.e. celle stockée dans l'arbre), soit lorsque la valeur sémantique de T a été recalculée. Un raisonnement analogue à celui utilisé lors de l'analyse de l'algorithme d'évaluation topologique, permet de constater que cette deuxième partie termine et que toutes les instances d'attributs concernées par des modifications (i.e. celles où l'ancienne valeur est différente à la nouvelle) ont été réévaluées. Remarquer aussi que seules ces instances ont été réévaluées.

Cet algorithme de propagation étant une généralisation de l'algorithme d'évaluation topologique, il n'est pas difficile de constater que l'algorithme possède une complexité proportionnelle à O(H(T)).

Si nos appliquons l'algorithme de propagation à l'arbre de la figure 4.7, nous constaterons que :

T(réévaluation) = Si H(T) = 1 alors T(DGp)
Si H(T) = 2 alors 2T(DGp)
Si H(T) = 3 alors 3T(DGp)

La figure 4.10 montre la réévaluation des instances d'attribut de l'arbre de la figure 4.7.

(1) : L'ensemble S peut être vide si DG_T n'est pas un graphe connexe (i.e. les instances d'attributs synthétisées de X peuvent éventuellement ne pas avoir de successeurs). S est vide si X s'avère être la racine de l'arbre de dérivation complet. Remarquer que si S est vide, la deuxième partie de l'algorithme (i.e. la propagation) n'est pas exécutée.

Avec ce nouvel algorithme nous assurons qu'une instance d'attribut n'est évaluée que lorsque toutes les instances desquelles elle dépend ont été évalués. Remarquer que la manipulation de l'ensemble S est faite en tenant compte du nombre d'ancêtres de chaque nœud du graphe de dépendances. Aussi, il est important de noter que s'il y a une dépendance de la forme X.a → Y.b, lors de l' évaluation de l'instance X.a, Y.b est insérée dans S et l'arête correspondant à X.a → Y.b est "supprimée" de sorte que si Y.b n'a plus aucun ancêtre, elle sera mise en tête de S et donc évaluée avant les instances d'attributs où les ancêtres n'ont pas encore été évaluées. Une instance d'attribut Y.b n'est donc évaluée que si tous ses ancêtres ont déjà été évaluées.

Le temps de réévaluation T(réévaluation) est donc égal à T(DGp) * H(T).

T(réévaluation) = T(DGp) * H(T) = K * H(T) = O(H(T))

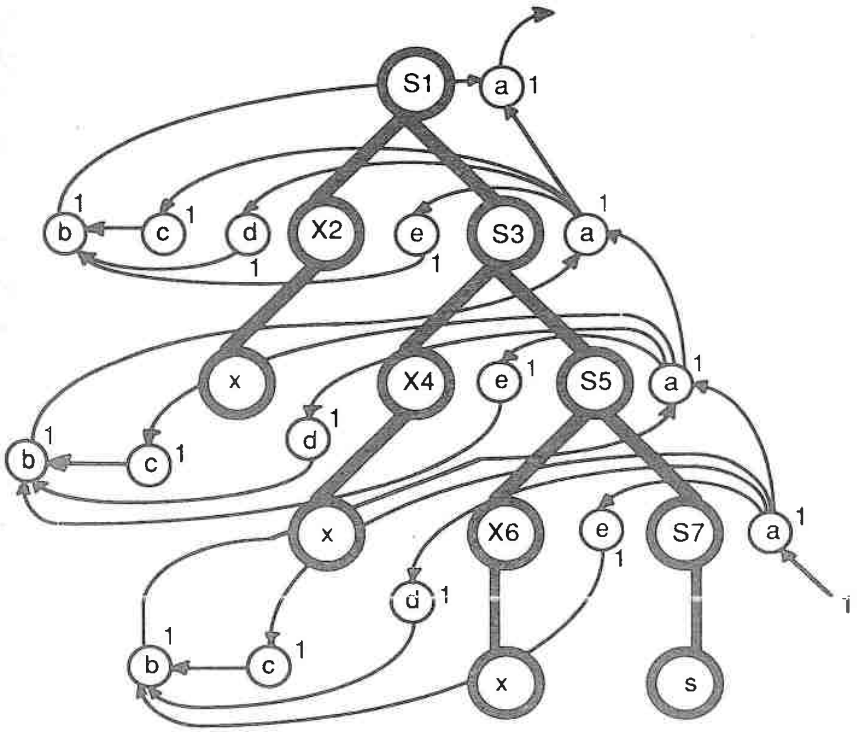


Figure 4.10 "La réévaluation des instances d'attributs avec l'algorithme de propagation implanté dans GEODE".

3. LA CONSTRUCTION D'OUTILS DANS LE SYSTEME GEODE.

3.1 INTRODUCTION.

Nous avons vu dans le chapitre II, quels sont les composants du système GEODE et quelles sont leurs fonctionnalités. Dans cette section nous allons voir, de façon plus ou moins exhaustive, la manière dont les grammaires attribuées sont utilisées par GEODE. Ainsi, nous présenterons deux outils : un décompilateur d'arbres de dérivation et un gestionnaire arbre-écran. L'union de ces deux outils constitue l'éditeur structuré du système GEODE.

Il est clair que GEODE est un système paramétré par la grammaire du langage de programmation choisi. Il peut donc générer des environnements de programmation pour n'importe quel langage. Pour des questions de facilité, les exemples qui seront donnés sont basés sur la grammaire du langage PL0 [Wirth 1976] qui est entièrement donnée en annexe. Nous avons aussi travaillé sur la grammaire du langage PASCAL [Wirth 1971].

3.2 L'EDITION STRUCTUREE.

Nous allons présenter dans cette section l'éditeur structuré du système GEODE et nous montrerons comment, à l'aide des grammaires attribuées, il a été conçu. Tous les exemples qui seront donnés n'analysent qu'une partie de la définition syntaxico-sémantique d'une grammaire. Les définitions complètes se trouvent dans les annexes.

Un éditeur structuré est un outil qui construit un programme en développant progressivement les symboles non-terminaux d'une grammaire. Le développement d'un symbole non-terminal consiste à remplacer ce symbole par une des parties droites des règles dans lesquelles il apparaît en partie gauche.

Ainsi, si nous avons les productions suivantes : (voir Grammaire 3.4 Chapitre III)

3) Decl_liste → "VAR" ident

4) Decl_liste → "VAR" ident Decl_liste

le symbole Decl_liste peut être développé de deux manières : soit comme la suite "VAR" ident, soit comme la suite "VAR" ident Decl_liste. Le choix d'utiliser la production 3 ou la production 4 pour le développement de Decl_liste est fait par l'utilisateur.

Au fur et à mesure que l'utilisateur développe les non-terminaux de la grammaire il construit son programme, en faisant grandir l'arbre de dérivation qui le représente. Or, il est clair que l'utilisateur n'a pas à se soucier de cet arbre et que pour lui le programme n'est que la représentation textuelle qu'il reçoit sur l'écran. Il doit donc exister un mécanisme permettant de transformer un arbre de dérivation (i.e. représentation du programme du point de vue du système) en un texte affichable à l'écran (i.e. représentation du programme du point de vue de l'utilisateur). Ce mécanisme reçoit le nom de *décompilateur* [Donzeau-Gouge 1975].

Il reste cependant un problème à résoudre : lorsque l'utilisateur utilise le curseur sur l'écran, il peut choisir n'importe quel symbole non-terminal pour le développer. Il est donc nécessaire d'avoir un mécanisme nous permettant, à tout moment, de faire le lien entre la représentation textuelle du programme et sa représentation arborescente. Ce mécanisme est appelé *gestionnaire arbre-écran.*

Il est aussi à remarquer que le curseur n'est plus déplacé caractère par caractère comme sous un éditeur de textes classique; il se déplace d'unité syntaxique en unité syntaxique. Le curseur signale donc l'unité syntaxique courante, c'est-à-dire, celle choisie par l'utilisateur.

La figure 4.11 illustre le fonctionnement du décompilateur et du gestionnaire arbre-écran, lorsque ceux-ci travaillent sur un programme du langage PL0.

Dans les sections qui suivent, nous étudions en détail comment définir ces deux outils avec une grammaire attribuée. Il est à remarquer que chaque outil est conçu indépendamment de l'autre, bien qu'il puissent communiquer entre eux comme nous allons le montrer.

3.21 Le décompilateur.

Le décompilateur du système GEODE a été construit en utilisant une fonction d'affichage : "display" qui réalise l'affichage d'un sous-arbre; éventuellement la fonction reçoit comme premier paramètre une variable entière représentant l'indentation du texte. Cette fonction réalise l'affichage qui lui est demandé et retourne en résultat le nombre de caractères ayant été affichés.

La décompilation d'un arbre de dérivation complet est donc une combinaison d'appels à la fonction "display".

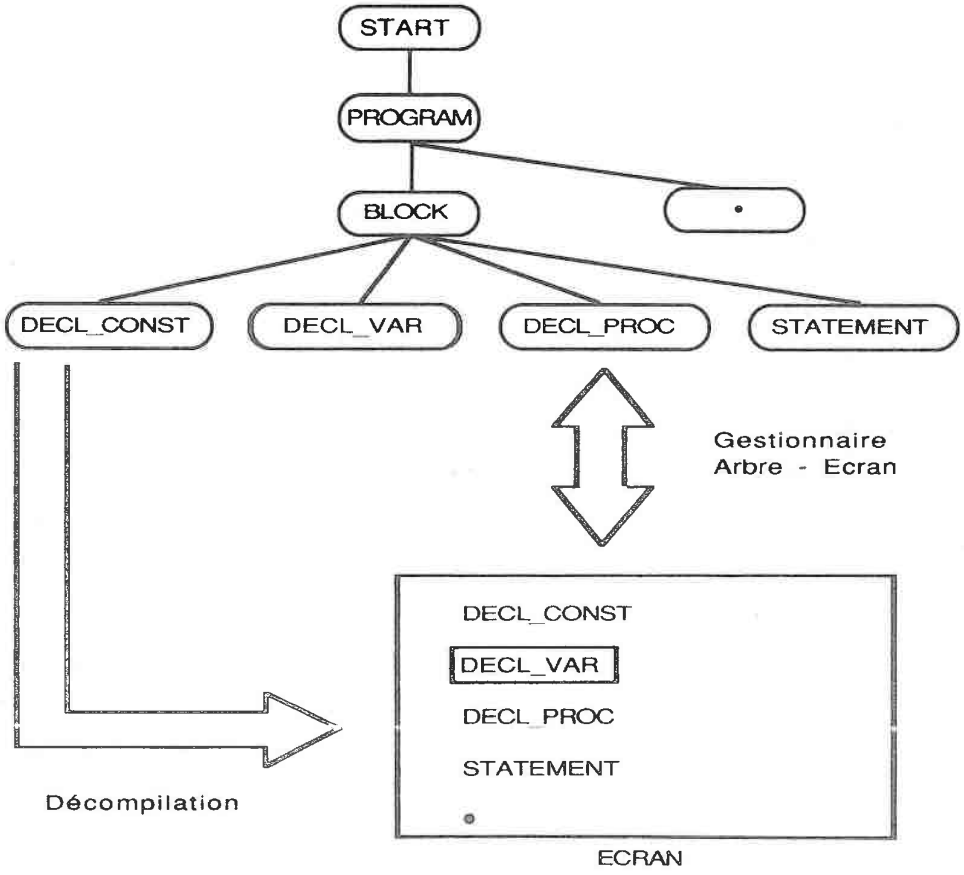


Figure 4.11 "La décompilation d'un arbre représentant un programme du langage PL0".

"Display" reçoit comme paramètres les valeurs des instances d'attributs de l'arbre de dérivation et éventuellement les caractères de "retour-chariot" symbolisé par "cr" et d'espacement symbolisé par " ". Nous utilisons trois attributs :

- L'attribut "rep" qui contient le nombre de caractères de la représentation textuelle d'un nœud. Cet attribut est synthétisé pour les symboles non-terminaux et hérité pour les symboles terminaux;
- L'attribut "tab" qui est chargé de contrôler l'indentation. Cet attribut est hérité et il n'est associé qu'à certains symboles non-terminaux. Il est à remarquer que cet attribut n'intervient pas dans le calcul de la valeur retournée par la fonction "display";
- Les symboles terminaux génériques utilisent un attribut hérité "str" qui représente la chaîne de caractères (i.e. représentation lexicographique) qui leur est associée;

Analysons maintenant un extrait de la grammaire attribuée décrivant le décompilateur. (La grammaire complète est donnée en annexe).

Grammaire 4.1. "Un extrait de la définition du décompilateur du langage PL0".

1) start → ¥
   start.rep = display ("START");

2) start → program
   start.rep = program.rep;
   program.tab = 0;

3) program → ¥
   program.rep = display (program.tab, "PROGRAM");

4) program → block "."
   program.rep = block.rep + display (cr) + point.rep;
   point.rep = display (".");
   block.tab = program.tab;

5) block → ¥
   block.rep = display (block.tab, "BLOCK");

6) block → decl_const decl_var decl_proc statement

block.rep = decl_const.rep + display (cr) + decl_var.rep + display (cr) + decl_proc.rep + display (cr) + statement.rep + display (cr);

decl_const.tab = block.tab;

decl_var.tab = block.tab;

decl_proc.tab = block.tab;

statement.tab = block.tab;

7) decl_const → ¥

decl_const.rep = display (decl_const.tab, "DECL_CONST");

8) decl_const → "CONST" decl_const_list ";"

decl_const.rep = CONST.rep + display (cr) + decl_const_list.tab + decl_const_list.rep + display (" ") + semicolon.rep;

CONST.rep = display (decl_const.tab, "CONST");

semicolon.rep = display (";");

decl_const_list.tab = 1 + decl_const.tab;

9) decl_const → "empty"

decl_const.rep = 0;

10) decl_const_list → ¥

decl_const_list.rep = display (decl_const_list.tab, "DECL_CONST_LIST");

11) decl_const_list → "ident" "=" "number"

decl_const_list.rep = decl_const_list.tab + ident.rep + display (" ") + equal.rep + display (" ") + number.rep;

ident.rep = display (decl_const_list.tab, ident.str);

ident.str = fonction_id;

equal.rep = display ("=");

number.rep = display (number.str);

number.str = fonction_num;

12) decl_const_list → "ident" "=" "number" "," decl_const_list
decl_const_list@1.rep = decl_const_list@1.tab + ident.rep + display (" ") +
equal.rep + display (" ") + number.rep + comma.rep +
display (cr) + decl_const_list@2.rep;
decl_const_list@2.tab = decl_const_list@1.tab;
ident.rep = display (decl_const_list@1.tab, ident.str);
ident.str = fonction_id;
equal.rep = display ("=");
number.rep = display (number.str);
number.str = fonction_num;
comma.rep = display (",");

Plusieurs remarques concernant la Grammaire 4.1 doivent être faites. Analysons les équations des productions les plus représentatives de la grammaire :

- Production 1 : L'instance d'attribut "start.rep" reçoit la valeur display ("START"), c'est-à-dire, le nombre de caractères de la représentation textuelle du symbole "start";

- Production 2 : L'instance d'attribut "start.rep" reçoit la valeur de l'instance "program.rep". On initialise l'indentation du programme, en faisant "program.tab = 0";

- Production 3 : Cette production indique que le symbole "program" n'a pas encore été développé et que sa représentation textuelle, "program.rep" est égale à display (program.tab, "PROGRAM"). Il est à remarquer que ce dernier appel à la fonction "display" ne retourne que la longueur de la chaîne "PROGRAM"; l'instance d'attribut "program.tab" n'intervient pas dans le calcul de la valeur retournée par la fonction;

- Production 4 : L'instance d'attribut "program.rep" est calculée en additionnant l'instance d'attribut "block.rep", l'appel display (cr) et l'instance

d'attribut "point.rep". Cette dernière instance est calculée à partir de l'appel display ("."); l'indentation associée au symbole "block" est celle associée au symbole "program";

- Production 6 : L'instance d'attribut "block.rep" sera calculée en fonction des instances d'attributs "decl_const.rep", "decl_var.rep", "decl_proc.rep", "statement.rep" et de quelques appels à display (cr). En outre, les instances d'attributs "decl_var.tab", "decl_const.tab", "decl_proc.tab" et "statement.tab" reçoivent la valeur de l'instance "block.tab";

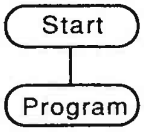
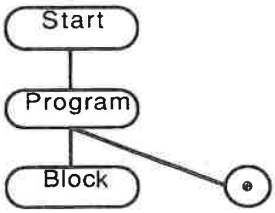
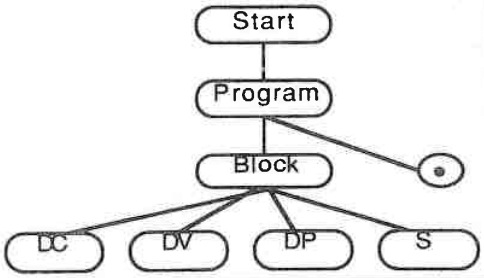
- Production 9 : Cette production signifie que l'utilisateur a décidé de développer le symbole "decl_const" en tant que la chaîne vide. Autrement dit, il n'y aura pas de déclarations de constantes dans le programme. Le mot clé "empty" représente en GEODE la chaîne vide et la constante zéro indique au décompilateur que le non-terminal en question ne devra pas être affiché à l'écran;

- Production 11 : La sémantique indique que la représentation textuelle du symbole "decl_const_list" sera donnée en appliquant la fonction "display" aux instances d'attributs "ident.rep", "equal.rep", "number.rep" et "decl_const_list.tab". Les équations "ident.str = fonction_id" et "number.str = fonction_num" indiquent que les représentations lexicographiques des symboles "ident" et "number" seront calculées en appelant respectivement "fonction_id" et "fonction_num", respectivement. En effet, "ident" et "number" sont des symboles terminaux génériques, ils ont donc besoin d'une analyse lexicographique. Ceci est le rôle des fonctions "fonction_id" et "fonction_num". Il est à remarquer que ces deux fonctions restent à la charge du concepteur de l'environnement;

- Production 12 : Cette production a le même fonctionnement que la production 11. Il est à noter que, du fait que le symbole "decl_const_list" apparaît

deux fois dans la production, il faut, lors de la définition des équations sémantiques, distinguer les deux occurrences. Ceci est le rôle du méta-caractère @.

Une fois que nous avons expliqué le fonctionnement du décompilateur, il nous paraît pertinent de montrer le déroulement de la construction d'un programme PL0 sous GEODE, afin de montrer la relation existant entre les représentations arborescente et textuelle du programme. (Les noms de certains symboles on été abrégés).

Description	Représentation Arborescente	Représentation Textuelle
Début de la session		START
Développement du non-terminal "Start"		PROGRAM
Développement du non-terminal "Program"		BLOCK •
Développement du non-terminal "Block"		DECL_CONST DECL_VAR DECL_PROC STATEMENT •

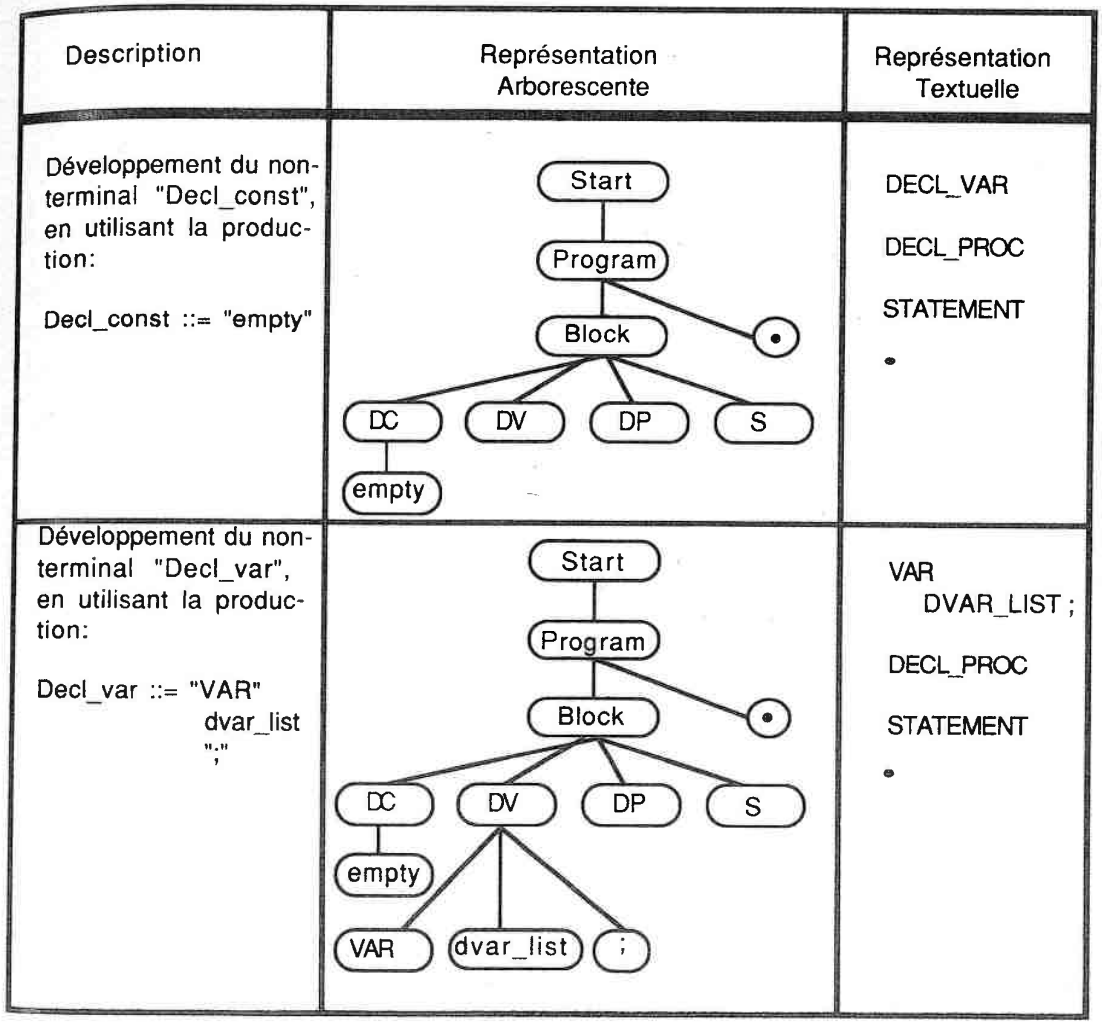


Figure 4.12 "La construction d'un programme sous GEODE".

3.22 Le gestionnaire arbre-écran.

Le fait que l'utilisateur puisse travailler sur la représentation textuelle du programme est un énorme avantage, car il a l'impression de travailler sur un éditeur de textes classique et il n'a donc pas à changer ni ses habitudes de travail ni son mode de raisonnement (i.e. l'utilisateur n'a pas à s'occuper de manipulations sur l'arbre de dérivation; ses intentions ne

s'expriment qu'en termes du texte).

Cependant, du point de vue du système ceci pose un problème : comment faire le lien, à tout moment, entre les deux représentations du programme? Ceci est le rôle du gestionnaire arbre-écran.

Le gestionnaire arbre-écran se charge donc d'identifier quelle est la partie affectée de l'arbre, lorsque l'utilisateur fait des manipulations sur la représentation textuelle. Le lien entre les deux représentations du programme doit être fait à tout moment, car s'il est vrai que l'utilisateur ne manipule que la représentation textuelle, le système lui ne connaît que la représentation arborescente.

Le lien texte-arbre est fait à travers des équations sémantiques nous permettant de lier la position du curseur sur l'écran à un nœud (ou à un sous-arbre) particulier de la représentation arborescente. Bien évidemment, lorsque nous développons ou modifions l'arbre de dérivation, les liens entre le curseur et l'arbre sont mis à jour à travers l'évaluation des attributs du gestionnaire arbre-écran.

Il est aussi à remarquer que le gestionnaire arbre-écran est un outil dépendant du type de terminal utilisé. Ainsi, cet outil n'est pas représenté par le même type d'équations s'il s'agit d'un terminal "bit-map" ou s'il s'agit d'un terminal classique dont les désignations sur l'écran se font à l'aide de coordonnées x, y. Le gestionnaire arbre-écran que nous allons décrire est celui que nous avons réalisé pour un Apple Macintosh Plus.

Le gestionnaire arbre-écran utilise les attributs suivants :

- L'attribut hérité "pos" représente la position du caractère de début de la représentation arborescente d'un nœud de l'arbre. Cet attribut est synthétisé pour le symbole "start" et hérité pour tous les autres symboles y compris les terminaux;
- Certaines productions utilisent les attributs "rep" et "tab" qui ont été décrits lors de la définition du décompilateur. Il est à remarquer que ces productions utilisent la notation <decomp>X.rep ou <decomp>X.tab, pour faire allusion à l'attribut rep ou à l'attribut "tab" du symbole X, respectivement.

Voici une partie de la définition du gestionnaire arbre-écran pour la grammaire de PL0. (La définition complète du gestionnaire arbre-écran est donnée en annexe).

Grammaire 4.2. "Un extrait de la définition du gestionnaire arbre-écran du langage PL0".

```
1) start        →   ¥
        start.pos = 0;
2) start        →   program
        start.pos = 0;
        program.pos = start.pos;
3) program      →   ¥
4) program      →   block "."
        block.pos = program.pos;
        point.pos = block.pos + <decomp>block.rep + 1;
5) block  →     ¥
6) block  →     decl_const decl_var decl_proc statement
        decl_const.pos = block.pos;
        decl_var.pos = decl_const.pos + <decomp>decl_const.rep + 1;
        decl_proc.pos = decl_var.pos + <decomp>decl_var.rep + 1;
        statement.pos = decl_proc.pos + <decomp>decl_proc.rep + 1;
7) decl_const   →   ¥
8) decl_const   →   "CONST" decl_const_list ";"
        CONST.pos = decl_const.pos;
        decl_const_list.pos = CONST.pos + <decomp>CONST.rep +
                              <decomp>decl_const_list.tab + 1;
        semicolon.pos = decl_const_list.pos + <decomp>decl_const_list.rep + 1;
9) decl_const   →   "empty"
```

10) decl_const_list → ¥

11) decl_const_list → "ident" "=" "number"
ident.pos = decl_const_list.pos;
equal.pos = ident.pos + <decomp>ident.rep + 1;
number.pos = equal.pos + <decomp>equal.rep + 1;

12) decl_const_list → "ident" "=" "number" "," decl_const_list
ident.pos = decl_const_list@1.pos;
equal.pos = ident.pos + <decomp>ident.rep + 1;
number.pos = equal.pos + <decomp>equal.rep + 1;
comma.pos = number.pos + <decomp>number.rep + 1;
decl_const_list@2.pos = comma.pos + <decomp>comma.rep +
<decomp>decl_const_list@2.tab + 1;

De même que pour le décompilateur, plusieurs explications concernant cette nouvelle grammaire sont nécessaires. Lorsque dans ces explications nous parlerons de représentation, nous ferons référence à la représentation textuelle de l'arbre ou d'un de ses sous-arbres. Aussi, lorsque nous parlerons du sous-arbre X, nous entendrons qu'il s'agit d'un sous-arbre dont la racine est étiquetée par le symbole X.

- Production 2 : Ces équations indiquent que la position de la représentation du sous-arbre program débute à la position zéro;

- Production 4 : La position de la représentation de block est celle de program. La position de la représentation de program est calculée à partir de la position et de la longueur de la représentation de block;

- Production 6 : On calcule la longueur du sous-arbre block à partir de ses fils. La position d'un fils est calculée en fonction de la position du père et de celle de ses frères;

- Production 8 : La longueur du sous-arbre decl_const est calculée à partir de ses

fils. La position d'un fils est calculée à partir de la position du père et de celle de ses frères. Remarquer que pour calculer la position de la représentation de decl_const_list, on fait appel à l'attribut decl_const_list.tab du décompilateur;

- Production 11 : La longueur du sous-arbre decl_const_list est calculée à partir de ses fils. La position d'un fils est calculée à partir de la position du père et de celle de ses frères;

Il est important de noter que le gestionnaire arbre-écran n'est pas indépendant du décompilateur car il utilise des attributs de ce dernier. Remarquons, toute fois, que les deux outils ont pu être définis séparément : On a d'abord construit le décompilateur et ensuite, **en utilisant le même formalisme** on a construit le gestionnaire arbre-écran; finalement afin de répercuter tout changement de la représentation textuelle de l'arbre, on a fait communiquer les deux outils à travers l'utilisation des attributs du décompilateur par le gestionnaire arbre-écran.

Le fait de définir des outils de manière indépendante et puis de les faire collaborer au moment de la construction du programme est un des principes de base de la conception même du système GEODE. Malheureusement, il arrive que des contraintes de temps d'exécution et d'espace mémoire nous obligent à fusionner deux outils. Ce problème sera abordé dans la section 3.3.

3.23 La vérification de la sémantique statique.

Un éditeur structurel doit être capable, non seulement d'assurer la validité syntaxique du programme, mais aussi de contrôler la sémantique statique. Rappelons que la sémantique statique d'un programme concerne les contrôles pouvant être effectués avant même d'exécuter le programme. Un exemple typique est celui de la vérification de type des variables utilisées dans le programme, c'est-à-dire que le contexte dans lequel est utilisée une variable doit être cohérent par rapport à la déclaration de celle-ci.

Le vérificateur sémantique que nous allons présenter est basé sur la gestion d'une table

des symboles et son fonctionnement est similaire à celui que nous avons analysé dans le chapitre III. Voici donc un extrait de la grammaire attribuée faisant les vérifications de la sémantique statique des programmes écrits en PL0. La grammaire complète est donnée dans les annexes.

Le vérificateur sémantique utilise les attributs suivants :

- L'attribut synthétisé "tsym" représente l'ensemble d'identificateurs ayant été déclarés et/ou pouvant être référencés à l'intérieur d'un module (i.e. un module est soit le programme principale, soit un sous-programme). Ces identificateurs représentent des constantes, des variables ou des procédures;
- Les attributs hérités "iconst", "ivar" et "iproc" représentent l'ensemble de variables, de constantes et de procédures pouvant être référencées à l'intérieur d'un module, mais non déclarées dans celui-ci;
- Les attributs synthétisés "sconst", "svar" et "sproc" représentent les ensembles de variables, de constantes et de procédures ayant été déclarés à l'intérieur d'un module.

Grammaire 4.3. "Un extrait de la définition du vérificateur de la sémantique statique du langage PL0".

1) start → ¥
   start.tsym = {∅};

2) start → program
   start.tsym = program.tsym;

3) program → ¥
   program.tsym = {∅};

4) program → block "."
   program.tsym = block.tsym;
   block.iconst = {∅};
   block.ivar = {∅};

5) block → ¥
block.tsym = block.iconst ∪ block.ivar;

6) block → decl_const decl_var decl_proc statement
block.tsym = (decl_const.sconst ∪ decl_var.svar ∪ decl_proc.sproc) ⊕
(block.iconst ∪ block.ivar);
decl_proc.iconst = decl_const.sconst ⊕ (block.iconst ∪ block.ivar);
decl_proc.ivar = decl_var.svar ⊕ (block.iconst ∪ block.ivar);
statement.iconst = decl_const.sconst ⊕ (block.iconst ∪ block.ivar);
statement.ivar = decl_var.svar ⊕ (block.iconst ∪ block.ivar);
statement.iproc = decl_proc.sproc;
ASSERT: decl_const.sconst ∩ decl_var.svar = {∅};
ASSERT: decl_const.sconst ∩ decl_proc.sproc = {∅};
ASSERT: decl_var.svar ∩ decl_proc.sproc = {∅};

7) decl_const → ¥
decl_const.sconst = {∅};

8) decl_const → "CONST" decl_const_list ";"
decl_const.sconst = decl_const_list.sconst;

9) decl_const → "empty"
decl_const.sconst = {∅};

10) decl_const_list → ¥
decl_const_list.sconst = {∅};

11) decl_const_list → "ident" "=" "number"
decl_const_list.sconst = {<decomp>ident.str};

12) decl_const_list → "ident" "=" "number" "," decl_const_list
ASSERT: {<decomp>ident.str} ∉ decl_const_list@2.sconst;
decl_const_list@1.sconst = {<decomp>ident.str} ∪ decl_const_list@2.sconst;

.
.
.

25) decl_proc_list → "ident" ";" block
decl_proc_list.sproc = {<decomp>ident.str};
block.iconst = decl_proc_list.iconst;
block.ivar = decl_proc_list.ivar;

26) statement → ¥

27) statement → "ident" " :=" expression
ASSERT: {<decomp>ident.str} ∈ statement.ivar;
ASSERT: {<decomp>ident.str} ∉ statement.iconst;
ASSERT: {<decomp>ident.str} ∉ statement.iproc;
expression.iconst = statement.iconst;
expression.ivar = statement.ivar;

.
.
.

Le vérificateur sémantique du langage PL0 réalise les vérifications suivantes :

- Un identificateur doit être associé soit à une déclaration de constante, à une déclaration de procédure ou à une déclaration de variable;
- Un identificateur associé à une constante ou à une procédure ne peut jamais apparaître en partie gauche d'une affectation;
- L'identificateur suivant le mot clé "CALL" doit nécessairement être associé à une déclaration de procédure;
- Les identificateurs apparaissant dans une expression ne peuvent être associés à des procédures;
- Un identificateur ne peut être déclaré plusieurs fois.

Nous analyserons maintenant quelques équations sémantiques de la grammaire attribuée du vérificateur sémantique :

- Production 6 : La première équation sémantique associée à cette production indique que l'ensemble des constantes, des variables et des

procédures associé au module est constitué de deux types d'identificateurs : Le premier est l'ensemble d'identificateurs ayant été déclarés à l'intérieur du module; le deuxième est l'ensemble d'identificateurs ayant été déclarés à l'extérieur du module, mais pouvant être référencés par celui-ci. Le symbole $\oplus$ signifie que lors de l'union de ces deux ensembles, si un identificateur appartient aux deux ensembles, on ne gardera que les caractéristiques (i.e. s'il s'agit d'une constante, d'une variable ou d'une procédure) de l'identificateur appartenant au premier ensemble, c'est-à-dire l'ensemble de déclarations propres au module. Les autres équations sémantiques associées à la production fonctionnent de la même manière, à l'exception des trois dernières, qui sont des assertions vérifiant qu'un identificateur déclaré à l'intérieur du module n'est pas associé à plusieurs types.

- Production 9 : Le non-terminal "decl_const" dérivant sur le vide, l'ensemble de déclarations de constantes est égale à l'ensemble vide;

- Production 11 : L'ensemble des identificateurs déclarés comme constantes contient l'identificateur venant d'être déclaré;

- Production 12 : On rajoute un nouvel identificateur à l'ensemble de constantes; si l'identificateur y figuré déjà, une erreur sémantique est produite;

- Production 25 : L'identificateur apparaissant dans cette production est rajouté à l'ensemble de procédures du module concerné. Le nouveau sous-programme hérite de l'ensemble de constantes et de variables ayant été déclarés dans les modules précédentes;

- Production 27 : L'identificateur apparaissant dans la partie gauche d'une affectation ne peut être ni une constante ni une procédure, il a du être déclaré en tant que variable. Le symbole "expression" hérite de l'ensemble de constantes et de procédures pouvant être utilisées par les

instructions du programme.

3.3 LA DOCUMENTATION AUTOMATIQUE DES PROGRAMMES.

Il est clair que le principal composant d'un environnement de programmation, tel que nous le concevons, est l'éditeur structuré, car il constitue l'interface entre l'utilisateur et l'environnement. Cependant, dans un environnement vraiment général il doit y avoir d'autres outils qui fournissent une aide supplémentaire à l'utilisateur tout au long de la construction de son programme.

Un exemple de ce type d'outils est celui de la documentation automatique des programmes [Mills 1970]. Cet outil sera appelé documentaliste.

Un documentaliste est un outil dont la tâche est de gérer et de stocker un ensemble d'informations concernant le programme qui est en train d'être construit. Ces informations sont typiquement constituées d'un dictionnaire de variables et par une liste de références croisées, et leur gestion est réalisé par une table des symboles.

Lorsque l'on développe les symboles non-terminaux de la grammaire, le documentaliste fournit à l'utilisateur un certain nombre d'informations. Ces informations peuvent concerner l'utilisation d'une variable, mais aussi les conditions de sortie d'une instruction d'itération ou simplement des remarques faites au sujet de la méthode ou de l'algorithme utilisé. Il est évident que dans le cadre de l'utilisation d'un langage de programmation comme PASCAL ou ALGOL, un documentaliste doit prendre en compte le niveau d'imbrication des sous-programmes et donc la portée des différentes variables.

Il est à remarquer que la documentation de programmes peut aller beaucoup plus loin, dans le sens où on pourrait construire un véritable outil de vérification de programmes [Basili 1982]. Cependant, dans l'environnement de programmation du langage PL0 nous n'avons pas pris en compte les aspects concernant la vérification de programmes, car nous considérons qu'un outil de vérification de programmes doit être conçu comme un outil à part entière et non pas comme un "simple" composant d'un outil de documentation.

Le documentaliste du langage PL0 fournit des informations que l'on rencontre

typiquement dans une liste de références croisées. (i.e. une liste de références croisées constitue un moyen nous permettant d'avoir des informations concernant l'utilisation des identificateurs d'un programme). Dans le cas du langage PL0, les identificateurs peuvent être des constantes, des variables ou bien des procédures.

Une liste de références croisées pour un programme de PL0 contient l'information suivante :

- Pour les constantes : On donne la valeur de la constante, le nom de la procédure où elle a été déclarée et les noms des procédures et les numéros de ligne des instructions où celle-ci est utilisée;
- Pour les variables : On donne le nom de la procédure où la variable a été déclarée, ainsi que les noms des procédures et les numéros des lignes où elle est utilisée;
- Pour les procédures : On donne le nom de la procédure, éventuellement le programme principal, où une procédure a été déclarée; on indiquera quelles sont les constantes, les variables et les procédures lui étant associés et finalement, on donne les noms des procédures et les numéros des lignes où la procédure en question est appelée.

L'exemple suivant nous montre une liste de références croisées :

Exemple : "Une liste de références croisées".

CONSTANTES

Identificateur	Valeur	Déclarée dans :	Numéro de Ligne
Pi	3.1416	Programme_Principal	4

Utilisée par :	Numéro de Ligne
Programme_Principal	128
Programme_Principal	135
Surface_Cercle	256

Identificateur	Valeur	Déclarée dans :	Numéro de Ligne
Vitesse_Lumière	3.0E+05	Calculer_Energie	6

Utilisée par :	Numéro de Ligne
Calculer_Energie	12
Calculer_Energie	35

VARIABLES

Identificateur	Déclarée dans :	Numéro de Ligne
Salaire	Programme_Principal	6

Utilisée par :	Numéro de Ligne
Sécurité_Sociale	25
Régime_Complémentaire	37
Mutuelle	54
Programme_Principal	75

Identificateur	Déclarée dans :	Numéro de Ligne
I	Programme_Principal	4

Utilisée par :	Numéro de Ligne
Programme_Principal	10
Programme_Principal	25
Calculer_Moyenne	42
Comptabiliser_Totaux	80
Comptabiliser_Totaux	100

PROCEDURES

Identificateur	Déclarée dans :	Numéro de Ligne
Impression_des_Résultats	Programme_Principal	10

Appelée par :	Numéro de Ligne
Surface	20
Périmètre	35
Distance	50
Longueur	65

La grammaire attribuée suivante constitue donc un extrait de l'outil de documentation de programmes où seules les références croisées ont été prises en considération. (La grammaire complète de cet outil est donnée en annexe). Il est à remarquer que le documentaliste et l'outil de vérification sémantique (c.f. section 3.13 chapitre IV) utilisent tous les deux une table des symboles. La seule différence est que la table utilisée par le documentaliste contient plus d'informations que celle utilisée par l'outil de vérification sémantique. Il serait cependant regrettable d'avoir à gérer deux tables de symboles, d'autant plus que la table utilisée par l'outil de vérification sémantique n'est qu'un sous-ensemble de celle employée par le documentaliste. C'est la raison pour laquelle nous avons décidé de construire un nouvel outil qui sera chargé à la fois des vérifications sémantiques de l'éditeur et de la gestion des informations de la liste des références croisées.

Ce nouvel outil utilise les attributs suivants :

- L'attribut synthétisé "tsym" représente la table des symboles où seront stockées les informations utilisées pour la vérification sémantique et pour la documentation automatique des programmes. Cet attribut va représenter l'ensemble de variables, de constantes et de procédures déclarées et utilisées par une procédure ou éventuellement, le programme principal;
- L'attribut "acproc" représente le nom de la procédure à laquelle appartiennent les déclarations et/ou les instructions à un instant donné. Cet attribut est synthétisé pour l'axiome et hérité pour les autres symboles de la grammaire;
- L'attribut hérité "iconst" représente l'ensemble de constantes pouvant être utilisées par une procédure;
- L'attribut hérité "ivar" représente l'ensemble de variables pouvant être utilisées par une procédure;
- L'attribut hérité "iproc" représente l'ensemble de procédures pouvant être appelées par une procédure;
- L'attribut synthétisé "sconst" représente l'ensemble de constantes ayant été déclarées à l'intérieur d'une procédure;
- L'attribut synthétisé "svar" représente l'ensemble de variables ayant été déclarées à l'intérieur d'une procédure;

- L'attribut synthétisé "decl" représente, pour les symboles non-terminaux, l'ensemble de variables, de constantes et de procédures ayant été déclarées à l'intérieur d'une procédure;
- L'attribut synthétisé "util" représente, pour les symboles non-terminaux, l'ensemble de constantes, de variables et de procédures ayant été utilisées par une procédure;
- L'attribut hérité "valeur" contient la valeur numérique d'une constante;
- L'attribut hérité "decl" représente, pour le symbole terminal "ident", le nom de la procédure où l'identificateur a été déclaré;
- L'attribut hérité "dligne" contient le numéro de la ligne où un identificateur a été déclaré;
- L'attribut hérité "util" représente, pour le symbole terminal "ident", le nom des procédures où l'identificateur a été utilisé;
- L'attribut hérité "uligne" représente les numéro des lignes où l'identificateur a été utilisé.

Voici donc un extrait de la grammaire attribuée réalisant la vérification sémantique et la documentation des programmes. La grammaire complète est donnée en annexe.

Grammaire 4.4. "Un outil de vérification sémantique et de documentation automatique des programmes".

1) start → ¥
   start.tsym = {∅};
   start.acproc = "Programme_Principal";

2) start → program
   start.tsym = program.tsym;
   program.acproc = start.acproc;

3) program → ¥
   program.tsym = {∅};

4) program → block "."

program.tsym = block.decl ∪ block.util;
block.acproc = program.acproc;
block.iconst = {∅};
block.ivar = {∅};

5) block → ¥

block.decl = block.iconst ∪ block.ivar;
block.util = {∅};

6) block → decl_const decl_var decl_proc statement

block.decl = (decl_const.sconst ∪ decl_var.svar ∪ decl_proc.sproc) ⊕
(block.iconst ∪ block.ivar);
block.util = statement.util;
decl_proc.iconst = decl_const.sconst ⊕ (block.iconst ∪ block.ivar);
decl_proc.ivar = decl_var.svar ⊕ (block.iconst ∪ block.ivar);
statement.iconst = decl_const.sconst ⊕ (block.iconst ∪ block.ivar);
statement.ivar = decl_var.svar ⊕ (block.iconst ∪ block.ivar);
statement.iproc = decl_proc.sproc;
decl_const.acproc = block.acproc;
decl_var.acproc = block.acproc;
decl_proc.acproc = block.acproc;
statement.acproc = block.acproc;
decl_const.util = statement.util;
decl_var.util = statement.util;
ASSERT: decl_const.sconst ∩ decl_var.svar = {∅};
ASSERT: decl_const.sconst ∩ decl_proc.sproc = {∅};
ASSERT: decl_var.svar ∩ decl_proc.sproc = {∅};

7) decl_const → ¥

decl_const.sconst = {∅};

8) decl_const → "CONST" decl_const_list ";"

decl_const.sconst = decl_const_list.sconst;
decl_const_list.acproc = decl_const.acproc;
decl_const_list.util = decl_const.util;

9) decl_const → "empty"

decl_const.sconst = {∅};

10) decl_const_list → ¥

decl_const_list.sconst = {∅};

11) decl_const_list → "ident" "=" "number"

ident.decl = decl_const_list.acproc;
ident.valeur = <decomp>number.str;
ident.dligne = ligne (<scrmgr>ident.pos);
decl_const_list.sconst = {<decomp>ident.str, ident.decl, ident.valeur,
ident.dligne};
ident.util = prendre_util (<decomp>ident.str, decl_const_list.util);
ident.uligne = prendre_uligne (<decomp>ident.str, decl_const_list.util);

12) decl_const_list → "ident" "=" "number" "," decl_const_list

ASSERT: {<decomp>ident.str} ∉ decl_const_list@2.sconst;
ident.decl = decl_const_list@1.acproc;
ident.valeur = <decomp>number.str;
ident.dligne = ligne(<scrmgr>ident.pos);
decl_const_list@1.sconst = {<decomp>ident.str, ident.decl, ident.valeur,
ident.dligne} ∪ decl_const_list@2.sconst;
ident.util = prendre_util (<decomp>ident.str, decl_const_list@1.util);
ident.uligne = prendre_uligne (<decomp>ident.str, decl_const_list@1.util);
decl_const_list@2.acproc = decl_const_list@1.acproc;
decl_const_list@2.util = decl_const_list@1.util;

.
.
.

25) decl_proc_list → "ident" ";" block

block.iconst = decl_proc_list.iconst;
block.ivar = decl_proc_list.ivar;
block.acproc = <decomp>ident.str;
ident.decl = decl_proc_list.acproc;
ident.dligne = ligne (<scrmgr>ident.pos);
decl_proc_list.sproc = {<decomp>ident.str, ident.decl, ident.ligne};

26) statement → ¥

27) statement → "ident" ":=" expression

ASSERT: {<decomp>ident.str} ∈ statement.ivar;
ASSERT: {<decomp>ident.str} ∉ statement.iconst;
ASSERT: {<decomp>ident.str} ∉ statement.iproc;
expression.iconst = statement.iconst;
expression.ivar = statement.ivar;
expression.acproc = statement.acproc;
ident.decl = prendre_decl (<decomp>ident.str, statement.iconst, statement.ivar);
ident.dligne = prendre_dligne (<decomp>ident.str, statement.iconst, statement.ivar);
ident.util = statement.acproc;
ident.uligne = ligne (<scrmgr>ident.pos);
statement.util = {<decomp>ident.str, ident.util, ident.uligne} ∪ expression.util;

Analysons maintenant quelques productions de cette grammaire :

- Production 1 : Les équations sémantiques associées à cette production indiquent que la table des symboles est vide et que le procédure active est le programme principal;

- Production 4 : Les équations sémantiques indiquent que la table des symboles est constituée des ensembles de variables, de constantes et de procédures déclarées et utilisées dans le programme;

- Production 6 : Les équations sémantiques associées à cette production possèdent la signification suivante :

  - Les ensembles de variables, de constantes et de procédures déclarés au niveau du non-terminal "block" (i.e. la structure de programme représente au niveau de "block" sera appelée "module") sont constitués des déclarations faites à l'intérieur du module, mais aussi par les déclarations faites dans les sous-programmes à l'intérieur desquels le module a été déclaré. Ces règles de portée des identificateurs sont implantées, au niveau des symboles non-terminaux, par l'attribut "decl";

  - Les constantes, les variables et les procédures utilisées à l'intérieur d'un module, sont déterminées dans la partie "instructions" de celui-ci;

- Production 11 : Les equations sémantiques de cette production, ainsi que celles de la production 12, représentent la mise à jour de la table des symboles. Les informations associées à un identificateur dans cette table des symboles sont les suivantes :

  - Le nom du module où l'identificateur a été déclaré. Ceci est représenté par l'attribut "decl" associé au symbole terminal "ident";

  - Si l'identificateur est une constante, on stocke sa valeur. Ceci est représenté par l'attribut "valeur";

  - L'attribut "dligne" représente le numéro de ligne où un identificateur a été déclaré. Cet attribut est calculé à partir de l'attribut "pos" du gestionnaire arbre-écran;

  - L'attribut "util", associé au symbole terminal "ident", représente

l'ensemble des modules utilisant l'identificateur en question. Bien entendu, cet attribut prend en compte les règles de portée des identificateurs;

- L'attribut "uligne" représente un ensemble contenant les numéros des lignes où un identificateur a été utilisé. Cet attribut prend aussi en compte, les règles de portée des identificateurs.

L'outil de documentation automatique nous permet donc, à tout moment, de savoir quel est le contexte dans lequel un identificateur a été déclaré et les contextes dans lesquels celui-ci est utilisé. Aussi, il est important de noter que cet outil est complètement indépendant de la manière dont la table des symboles sera implantée.

3.4 LA CONSTRUCTION D'AUTRES OUTILS.

Dans les sections précédentes nous avons vu de manière générale comment construire un éditeur structurel et un outil de vérification sémantique et de documentation. Or, dans un véritable environnement de programmation nous voudrions disposer d'autres outils qui aident l'utilisateur à mieux construire ses programmes. Mais, quels sont les autres outils que nous pourrions trouver dans un environnement de programmation basé sur le langage?

Pour répondre à cette question nous pouvons regarder dans la direction des environnements de programmation classiques. Nous pouvons donc constater que les outils qui manquent dans nos environnements basés sur le langage sont des outils d'exécution et d'aide à la mise au point des programmes, c'est-à-dire un interprète ou un compilateur et un débogueur. Remarquons aussi qu'il serait intéressant d'avoir un seul outil réalisant l'exécution et la mise au point des programmes, comme il a été suggéré par [Teitelbaum 1981]. Cet outil sera appelé *interprète incrémental.*

Dans un environnement basé sur le langage, un interprète incrémental doit pouvoir exécuter et éventuellement déboguer un programme, au fur et à mesure de sa construction. Ceci implique qu'un programme pourra être exécuté et/ou débogué même s'il n'est pas complètement terminé. Plus encore, on devra pouvoir modifier le programme et puis

reprendre l'exécution à l'endroit où on l'avait arrêtée, moyennant, bien entendu, certaines restrictions. On pourra, par exemple, modifier la valeur d'une variable, mais on ne pourra pas modifier le corps d'une procédure.

L'interprète incrémental d'un environnement basé sur le langage devra aussi être défini en utilisant les grammaires attribuées.

A l'heure actuelle, dans GEODE, nous n'avons pas encore de propositions dans ce sens. L'interprète incrémental reste donc une voie de recherche future.

Chapitre V.

L'IMPLANTATION DU SYSTEME GEODE.

1. INTRODUCTION.

Dans le chapitre II nous avons présenté un panorama général de l'architecture du système GEODE. Nous avons ainsi introduit la notion de représentation interne pour la syntaxe d'un langage de programmation et pour les descriptions des outils de l'environnement généré. Ce chapitre présente la philosophie de la programmation par objets, les représentations internes GEODE d'un langage de programmation et d'un programme, et enfin, l'implantation de l'algorithme d'évaluation d'attributs qui a été introduit dans le chapitre IV.

2. L'IMPLANTATION DU SYSTEME GEODE.

2.1 A PROPOS DE LA PHILOSOPHIE DE LA PROGRAMMATION ORIENTEE OBJET.

Cette section doit être considérée comme une brève introduction à la philosophie de la programmation par objets. Les ouvrages traitant le sujet sont nombreux, citons par exemple : [Althoff 1981], [Colnet 1987], [Cox 1984], [Meyer 1979].

La programmation par objets est un concept en programmation introduit par le langage SIMULA [Dahl 1966] qui est en train de jouer un rôle de plus en plus important dans la conception et l'implantation de systèmes informatiques actuels.

Les principaux buts de la programmation par objets sont :

- Simplifier la construction de systèmes informatiques grands et complexes;

- Encourager la production de systèmes informatiques modulaires, faciles à comprendre, réutilisables et évolutifs.

Le principe de base de la programmation par objets est de considérer que les données et les procédures qui manipulent ces données doivent être considérées comme une entité indivisible : l'*objet*. Les données ou *variables* inclues dans un objet ne peuvent être modifiées que par les procédures appartenant à cet objet. Dans la programmation par objets ces procédures sont appelées des *méthodes*. Dans les objets les variables représentent la composante statique et les méthodes représentent la composante dynamique.

Les ensembles d'objets ayant des comportements communs sont regroupés en *classes*. Une classe peut être considérée comme un modèle conceptuel décrivant le comportement des objets qui la représentent. Les représentants physiques d'une classe sont appelés *instances*. Une instance est donc un objet particulier qui a été créé en respectant les plans de construction de sa classe.

Les calculs effectués par un programme sont réalisés par des *envois de messages* aux différents objets de ce programme. L'envoi d'un message à un objet équivaut à adresser une requête à cet objet. Les envois de messages permettent d'activer une méthode et ils représentent donc le moyen de communiquer avec les objets. Le comportement d'un objet est défini uniquement selon la manière dont cet objet répond à l'envoi d'un message.

Les objets d'un programme peuvent être aussi rangés selon une structure hiérarchique définie par les classes. En haut de cette hiérarchie l'on retrouve la *"classe-objet"*, et attachées à la classe-objet on retrouve des *sous-classes*. Les sous-classes constituent des raffinements de la classe à laquelle elles sont rattachées. Les sous-classes permettent d'introduire des nouvelles variables et/ou de nouvelles méthodes qui représentent les caractéristiques propres aux sous-ensembles d'objets ainsi décrits.

Les caractéristiques des classes supérieures sont *héritées* par les classes inférieures. On peut donc considérer que les informations des classes supérieures sont recopiées dans les sous-classes inférieures. La structure hiérarchique des classes et des sous-classes constitue l'*arbre d'héritage*. Il existe aussi la notion d'*héritage multiple* où une classe peut hériter de

plusieurs autres classes sans que ces dernières ne soient liées par des relations hiérarchiques. L'ensemble de classes n'est plus structuré sous forme arborescente.

Exemple : La programmation par objets.

Le but de cet exemple est de montrer, brièvement, les principes fondamentaux de la programmation par objets. Il s'agit d'introduire un certain nombre de classes qui utilise GEODE pour la représentation des grammaires attribuées.

```
CLASSE†        Grammar_symbols

VARIABLES      symbol_type : string;
               symbol_representation : string;

METHODES       print_symbol_type (Grammar_symbols) → string
               ; impression du type d'un symbole de grammaire : terminal, non-terminal,
               ; axiome, ...
               print_symbol_representation (Grammar_symbols) → string
               ; impression de le représentation affichable à l'écran d'un symbole de
               ; grammaire
```

Nous avons ainsi construit la classe "Grammar_symbols". Cette classe possède une partie statique, les variable "symbol_type" et "symbol_representation", et une partie dynamique, les méthodes "print_symbol_type" et "print_symbol_representation".

A partir de la classe "grammar_symbols" nous pouvons définir des sous-classes qui héritent donc des variables et des méthodes de cette classe. Ainsi, par exemple, nous pouvons définir les sous-classes "Semantic_symbols", "Special_symbol" et "Syntax_symbol". Ces sous-classes sont construites de la manière suivante :

```
CLASSE         Semantic_symbols
               SOUS_CLASSE de Grammar_symbols

VARIABLES

METHODES
```

† La syntaxe utilisée dans cet exemple est purement illustrative.

pour la sous-classe "Special_symbol",

CLASSE Special_symbol
SOUS_CLASSE de Grammar_symbols

VARIABLES

METHODES

et pour la sous-classe "Syntax_symbols",

CLASSE Syntax_symbols
SOUS_CLASSE DE Grammar_symbols

VARIABLES

METHODES

Ces trois nouvelles classes ne définissent pas de nouvelles variables ni des nouvelles méthodes, elles héritent des variables et des méthodes de la classe "Grammar_symbols". Remarquons que le fait d'insérer des classes n'ayant ni des nouvelles variables ni des nouvelles méthodes n'est nullement une redondance ou un découpage inutile. En effet, il faut se rappeler qu'il s'agit de représenter les objets avec lesquels on travail le plus fidèlement possible.

Bien entendu, on peut aussi définir des nouvelles sous-classes à partir des classes "Semantic_symbols", "Special_symbol" et "Syntax_symbols". Par exemple, à partir de la classe "Syntax_symbols" on peut définir deux nouvelles sous-classes :

Une classe représentant les symboles non-terminaux,

CLASSE Nonterminal_symbols
SOUS_CLASSE DE Syntax_symbols

VARIABLES symbol_development
synthesized_attributes

METHODES develop_symbol (Nonterminal_symbols) → Productions_list
; cette méthode retourne les productions ayant comme partie gauche le
; symbole non-terminal en question
get_synthesized_attributes (Nonterminal_symbols) → Synthesized_attributes
; cette méthode récupère les attributs synthétisés associés au symbole
; non-terminal en question

et une classe représentant les symboles terminaux,

CLASSE Terminal_symbols
SOUS_CLASSE DE Syntax_symbols

VARIABLES inherited_attributes

METHODES get_inherited_attributes (Terminal_symbols) → Inherited_attributes
; cette méthode récupère les attributs hérités associés au symbole
; terminal en question

La sous-classe "Nonterminal_symbols" définit deux nouvelles variables : "symbol_development" et "synthesized_attributes", et deux nouvelles méthodes "develop_symbol" et "get_synthesized_attributes". La variable "symbol_development" contient toutes les parties droites des productions où le non-terminal en question apparaît en partie gauche. Ces productions sont recherchées par la méthode "develop_symbol". La variable "synthesized_attributes" contient une liste des attributs synthétisés associés à ce non-terminal. La méthode "get_synthesized_attributes" s'occupe de récupérer cette liste. La sous-classe "Terminal_symbols" définit une nouvelle variable : "inherited_attributes" et une nouvelle méthode "get_inherited_attributes". La variable "inherited_attributes" contient une liste des attributs hérités associés au symbole terminal en question. La méthode "get_inherited_attributes" récupère cette liste.

Finalement, on définit deux nouvelles classes à partir de la classe "Nonterminal_symbols" : "Start_symbol", pour représenter l'axiome de la grammaire, et "Other_nonterminals" pour représenter tous les autres symboles non-terminaux :

CLASSE Start_symbol
SOUS_CLASSE DE Nonterminal_symbols

VARIABLES

METHODES

et pour les autres symboles non-terminaux,

CLASSE Other_nonterminals
SOUS_CLASSE DE Nonterminal_symbols

VARIABLES inherited attributes

METHODES get_inherited_attributes (Other_nonterminals) → Inherited_attributes
; cette méthode récupère les attributs hérités associés au symbole
; non-terminal en question

Notons que les méthodes représentent fidèlement le monde réel. En effet, le symbol initial ou axiome d'une grammaire attribuée ne possède pas d'attributs hérités.

La figure 5.1 montre l'arbre d'héritage de cet exemple.

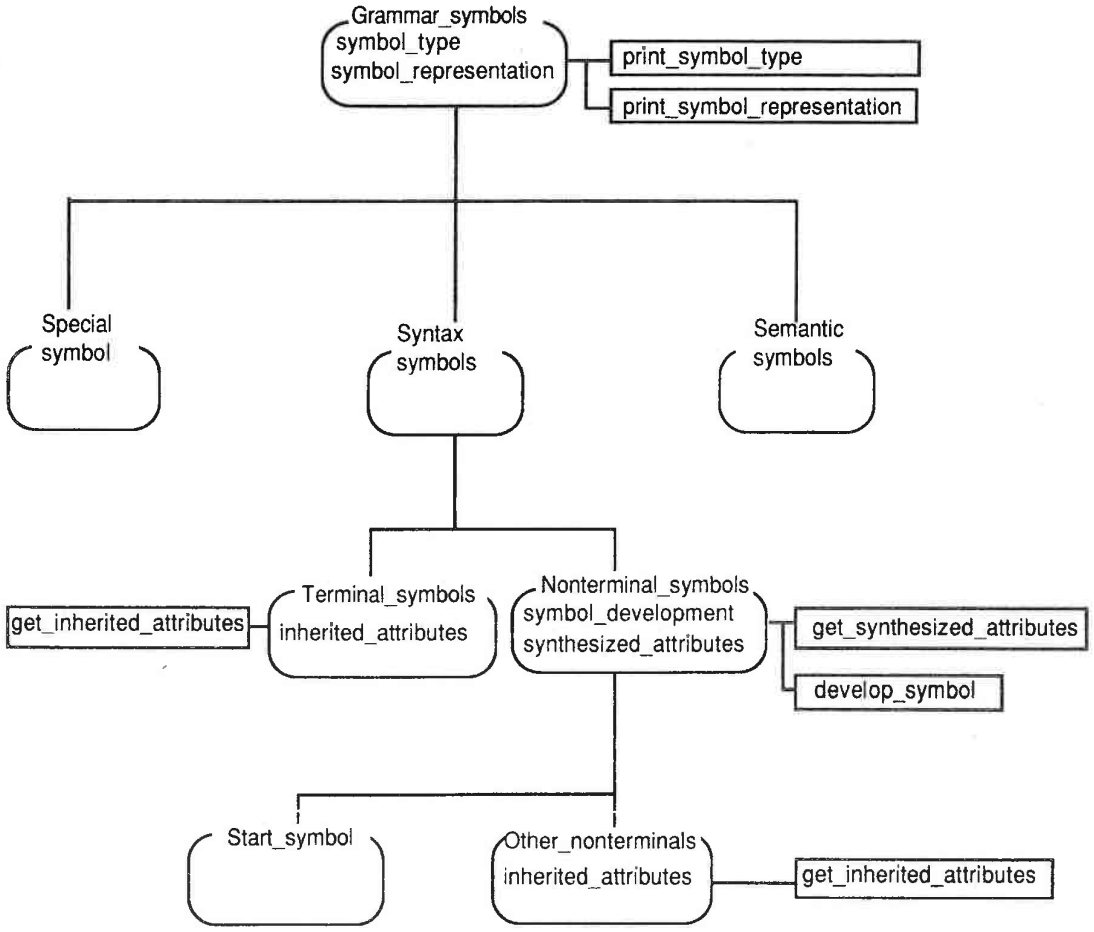


Figure 5.1 "L'arbre d'héritage de l'exemple".

Analysons maintenant la démarche qui doivent suivre les programmes susceptibles d'utiliser de symboles de grammaires.

- **Création des objets :** Le premier pas pour l'utilisation des objets est de les créer. La création des objets est effectuée à travers le mécanisme d'instanciation.

- **Manipulation des objets** : Une fois que nous avons créé des objets en instanciant les classes concernées, nous sommes en mesure d'appliquer à ces objets, les opérations prévues par les méthodes. Rappelons que le seul moyen de communiquer avec les objets est à travers les méthodes de leur classe d'appartenance.

La programmation par objets peut être résumée, brièvement, comme suit : [Bell 1987]

1. Identifier les objets jouant un rôle pour la résolution d'un problème et les opérations pouvant être appliquées à ces objets; [Vue externe des objets]

2. Déterminer les structures de données appropriés représentant la partie statique des objets (i.e. les variables) et les algorithmes réalisant les opérations applicables à ces objets (i.e. les méthodes). [Vue interne des objets]

Les objets constituent un mécanisme naturel pour l'obtention de programmes modulaires. L'interface d'un objet est décrite par l'objet lui-même à travers son protocole ou vue externe. Le protocole est constitué de deux parties : les messages auxquels l'objet répond et les réponses de l'objet lors de la réception de ces messages.

2.2 LES REPRESENTATIONS INTERNES DE LA SYNTAXE D'UN LANGAGE DE PROGRAMMATION ET DES OUTILS DE L'ENVIRONNEMENT GENERE.

Au chapitres III et IV nous avons détaillée la manière d'utiliser les grammaires attribuées dans GEODE pour représenter, d'un côté la syntaxe du langage de programmation, et d'un autre côté, les outils de l'environnement généré.

Dans cette section, nous étudierons la représentation interne GEODE des grammaires

attribuées qui a été introduite dans la section précédente. Une grammaire attribuée est représentée par un ensemble de classes Le_Lisp15.2 [Chailloux 1986] où la classe-objet est nommée"Grammar_symbols". Cette classe-objet a attachées deux sous-classes qui représentent, à leur tour, la grammaire à contexte libre générant la syntaxe du langage de programmation, et l'ensemble d'équations sémantiques définissant les outils de l'environnement à générer. Ces deux sous-classes ont aussi attachées d'autres sous-classes qui représentent des symboles syntaxiques pour la première, et des symboles sémantiques pour la deuxième. La figure 5.2 montre l'arbre de classes associées à la classe-objet "attribute-grammar". Remarquer que les classes "Attribute_grammar", "Productions" et "Semantic_equations" ne font pas partie de la hiérarchie des classes. Néanmoins, elles utilisent ou sont utilisées par les autres classes et sous-classes de la hiérarchie établie à partir de la classe-objet "Grammar_symbols". Ceci est représenté, dans la figure 5.2, par des lignes discontinues.

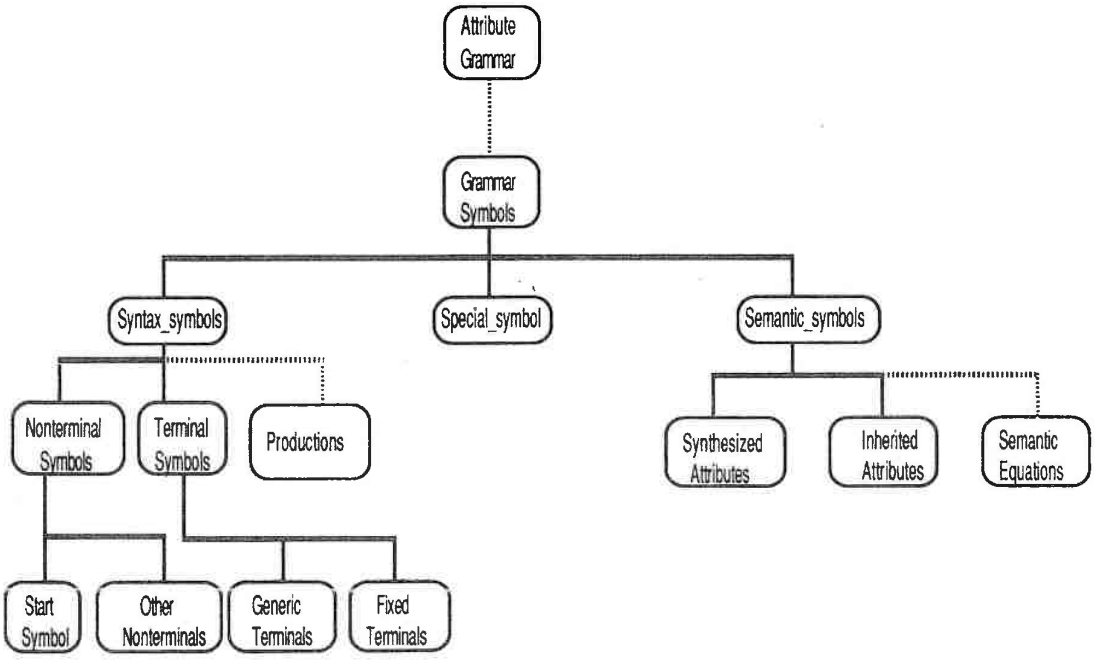


Figure 5.2 "La hiérarchie des classes utilisées dans GEODE pour la représentation syntaxico-sémantique d'un langage de programmation".

Quelques classes utilisées dans la figure 5.2 nécessitent une explication :

- La classe "Generic Terminals" représente l'ensemble de symboles terminaux ayant besoin d'une analyse lexicographique, comme les identificateurs ou les constantes;
- La classe "Fixed Terminals" représente l'ensemble de symboles terminaux n'ayant pas besoin d'une analyse lexicographique, comme les mots clés;
- La classe "Start Symbol" représente l'axiome de la grammaire; (c.f. section 2.1)
- La classe "Other Nonterminals" représente tous les symboles non-terminaux de la grammaire à l'exception de l'axiome; (c.f. section 2.1)
- La classe "Special Symbol" représente le symbole spécial ¥ dont le rôle a été expliqué dans le chapitre III.

La figure 5.3 montre les variables et les principales méthodes associées à la classe "attribute_grammar".

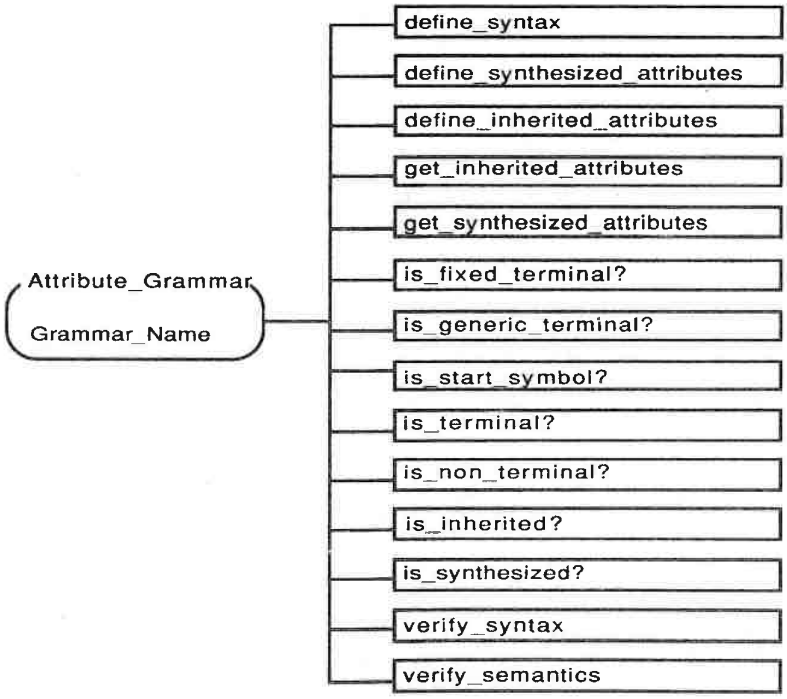


Figure 5.3 "Les variables et les principales méthodes associées à la classe 'attribute_grammar ".

La représentation interne d'une grammaire attribuée est obtenue en deux temps : on obtient d'abord la représentation de la grammaire à contexte libre sous-jacente, et ensuite les représentations des différents outils de l'environnement à générer.

La représentation interne de la grammaire à contexte libre est construite à partir d'un fichier Le_Lisp15.2 qui sera obtenu à partir de la définition LDG faite par l'utilisateur.

Exemple : Une partie de la définition de la grammaire de PL0 [Wirth 1976]. La définition complète Le_Lisp15.2 de cette grammaire est donnée dans les annexes.

```
({create_grammar}† pl0_grammar)
; création d'un objet de type "attribute_grammar", dont le nom est "pl0_grammar"

({create_start_symbol} pl0_grammar 'start)
; définition du symbole "start" en tant qu'axiome de la grammaire "pl0_grammar"

({create_generic_terminal} pl0_grammar "ident" 'fonction_id)
; déclaration du symbole "ident" en tant que terminal générique de la grammaire
; "pl0_grammar". La fonction "fonction_id" est la fonction réalisant l'analyse
; lexicographique de ce terminal générique.
({create_generic_terminal} pl0_grammar "number" 'fonction_nul)
; déclaration du symbole "number" en tant que terminal générique de la grammaire
; "pl0_grammar". La fonction "fonction_num" est la fonction réalisant l'analyse
; lexicographique de ce terminal générique.

({define_synonym} pl0_grammar "point" ".")
({define_synonym} pl0_grammar "semicolon" ";")
({define_synonym} pl0_grammar "equal" "=")
({define_synonym} pl0_grammar "point" ".")
({define_synonym} pl0_grammar "comma" ",")
```

† Les accolades délimitent le nom du message qui sera envoyé a l'objet de type "attribute_grammar". L'utilisation des accolades a été prise de Ceyx [Hullot 1983].

```
({define_synonym} pl0_grammar "assign" ":=")
({define_synonym} pl0_grammar "not_equal" "<>")
({define_synonym} pl0_grammar "less" "<")
({define_synonym} pl0_grammar "greater" ">")
({define_synonym} pl0_grammar "less_equal" "<=")
({define_synonym} pl0_grammar "greater_equal" ">=")
({define_synonym} pl0_grammar "plus" "+")
({define_synonym} pl0_grammar "minus" "-")
({define_synonym} pl0_grammar "pleft" "(")
({define_synonym} pl0_grammar "pright" ")")
({define_synonym} pl0_grammar "multiply" "*")
({define_synonym} pl0_grammar "divide" "/")

; Les synonymes sont utilisés pour pouvoir associer des attributs a des symboles terminaux
; de la grammaire tels que les opérateurs. Seuls les symboles apparaissant à droite seront
; utilisés lors de l'affichage et de l'impression de la grammaire.

({define_syntax} pl0_grammar
        '(
        (start
                        program)
        (program
                        block
                        "point")
        (block
                        decl_const
                        decl_var
                        decl_proc
                        statement)
        (decl_const
                        "CONST"
                        decl_const_list
                        "semicolon")
```

```
        (decl_const
                        "empty")
        (decl_const_list
                        "ident"
                        "equal"
                        "number")
        (decl_const_list
                        "ident"
                        "equal"
                        "number"
                        "comma"
                        decl_const_list)

.
.
.

; La forme générale pour définir une grammaire à contexte libre est la suivante :
;
; ({define_syntax} grammaire
;       '(
;        (production_1)
;        (production_2)
;        .
;        .
;        .
;        (production_n)
;        )
; )
; Où production_i est une liste de la forme :
; (Partie_gauche Partie_droite)
;
; Partie_gauche ∈ Symboles Non-terminaux
```

; Partie droite ∈ (Symboles Non-terminaux ∪ Symboles Terminaux)*
; Le symbole vide est représenté par la chaîne de caractères "empty"

A partir de cette définition le constructeur syntaxique (c.f. chapitre II) se charge d'obtenir la représentation interne de la grammaire à contexte libre. Cette représentation interne contient des informations qui faciliteront par la suite le développement des programmes des utilisateurs de l'environnement à générer, ainsi que la définition des outils de cet environnement.

En ce qui concerne la définition des outils de l'environnement, on procède en deux temps : on définit d'abord les attributs hérités et synthétisés qui seront associés aux symboles de la grammaire à contexte libre, et ensuite, on construit l'ensemble d'equations sémantiques qui seront associées aux productions de cette même grammaire. Un fichier Le_Lisp est obtenu à partir d'une définition LDS.

Exemple : Une partie de la définition des attributs hérités et synthétisés du décompilateur du langage PL0 [Wirth 1976]. La définition complète est donnée dans les annexes.

```
({define_synthesized_attributes} pl0_grammar 'unparser
        '(
        (start          rep)
        (program        rep)
        (block          rep)
        (decl_const     rep)
        .
        .
        .
        (number         rep)
        )
)

; La forme générale de la définition des attributs synthétisés d'un outil est la suivante :
;
; ({define_synthesized_attributes} grammaire 'outil_de_l'environnement
```

```
;       '(
;        (Symbole_syntaxique_1   Attribut_hérité_1)
;        (Symbole_syntaxique_2   Attribut_hérité_2)
;        .
;        .
;        .
;        (Symbol_syntaxique_n    Attribute_hérité_n)
;        )
;)
```

La déclaration des attributs hérités est faite de la même manière.

En ce qui concerne la déclaration des équations sémantiques, il faut faire comme suit :

```
({define_semantics} pl0_grammar 'unparser
        '(
        (1              (setatt     start.rep
                                    (display "START")))
        (2              (setatt     start.rep
                                    program.rep)
                        (setatt     program.tab 0))
        (3              (setatt     program.rep
                                    (display program.tab "PROGRAM")))
        (4              (setatt     program.rep
                                    (+ block.tab block.rep (display #\cr) point.rep))
                        (setatt     point.rep
                                    (display #\cr))
                        (setatt     block.tab
                                    program.tab))

        .
        .
        .
```

```
; La forme générale des équations sémantiques représentant un outil est la suivante :
;
; ({define_semantics} grammaire 'nom_de_l'outil
;         '(
;          (No_de_production        Equation_associée)
;          .
;          .
;          .
;          )
; )
```

A partir des définitions des attributs hérités et synthétisés, et des équations sémantiques d'un outil, le constructeur sémantique (c.f. chapitre II) produit une représentation interne GEODE, qui sera utilisée lors de la création des programmes des utilisateurs de l'environnement à générer.

Plus précisément, la représentation de la grammaire à contexte libre, ainsi que les représentations des différents ensembles d'équations des outils de l'environnement, seront utilisées pour le générateur d'évaluateurs sémantiques qui sera étudié dans la section 2.4 de ce chapitre.

Aussi, il est important de noter que les programmes Le_Lisp représentant la grammaire à contexte libre et les ensembles d'équations des outils de l'environnement, ne sont pas écrits directement par le concepteur de l'environnement. Ces programmes seront générés automatiquement par l'environnement de conception et définition de grammaires à contexte libre et par l'environnement de conception d'outils. Signalons toutefois qu'à l'heure actuelle, seul l'environnement de conception et définition de grammaires à contexte libre est entièrement opérationnel.

2.3 LA REPRESENTATION INTERNE DES PROGRAMMES DES UTILISATEURS DE L'ENVIRONNEMENT GENERE.

En ce qui concerne la représentation des programmes des utilisateurs des environnements de programmation générés par le système GEODE, nous avons choisi aussi une représentation orientée objet. La figure 5.4 montre la hiérarchie des classes utilisées pour la représentation des programmes.

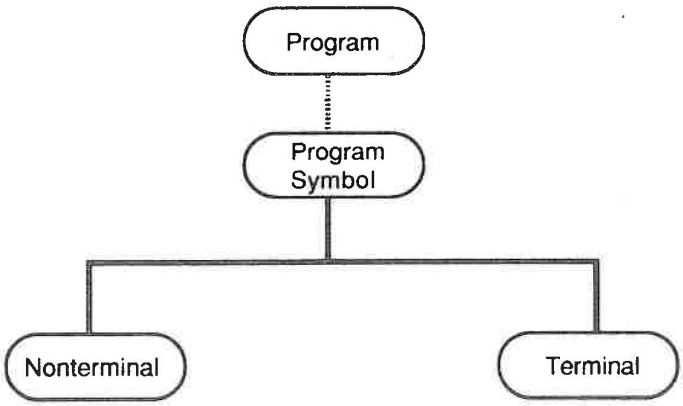


Figure 5.4 "La hiérarchie des classes GEODE pour la représentation des programmes".

Cette hiérarchie de classes GEODE pour la représentation des programmes, ainsi que la représentation interne de la grammaire à contexte libre, permettent au constructeur incrémental de programmes de produire un arbre de dérivation attribuée qui n'est en faite que la représentation GEODE de ces programmes. C'est donc sur cet arbre qui agiront les différents évaluateurs sémantiques qui représentent les outils de l'environnement de programmation généré.

Bien entendu, cette représentation de programmes inclue aussi une représentation du graphe de dépendances (i.e. chaque instance d'attribut associé à un nœud de l'arbre possède une liste indiquant quels sont ses successeurs). Rappelons que ce graphe est utilisé lors de l'évaluation incrémentale d'attributs.

2.4 L'IMPLANTATION DE L'EVALUATEUR D'ATTRIBUTS.

Comme nous avons vu dans le chapitre IV, l'algorithme d'évaluation incrémental d'attributs utilisé dans GEODE est divisé en deux parties : l'évaluation globale et la propagation des modifications. Cette dernière ayant été entièrement décrite dans le chapitre précédent, nous nous limiterons à expliquer l'implantation de l'évaluation globale.

L'évaluation globale est réalisée à l'aide de fonctions Le_Lisp mutuellement récursives. Ces fonctions représentent les différentes instances d'attributs utilisées et elles sont automatiquement générées par GEODE à partir des représentations internes de la syntaxe du langage utilisé et des descriptions des outils de l'environnement de programmation.

La méthode utilisée pour la génération automatique de ces fonctions, est celle proposée par M. Jourdan [Jourdan 1984] :

Algorithme5.1(1) "La génération des fonctions récursives pour l'évaluation incrémentale d'attributs".

```
PROCEDURE génère-fonction-attribut (attribut, production, instance, à-générer);
DEBUT
  ; Cette fonction génère la représentation fonctionnelle de l'attribut "attribut" associé au
  ; symbole X occupant la position "instance" dans la production "production". Si instance
  ; est égale à zéro, le symbole se trouve alors dans la partie gauche de la production,
  ; autrement, il se trouve dans la partie droite.
  ;
  ; "à-générer" est une variable booléenne indiquant si l'on doit générer une fonction ou
  ; bien, un simple appel de fonction. Ceci est particulièrement util si la grammaire attribuée
  ; en question n'est pas en forme normale; seuls les attributs synthétisés sont concernés.
  SI attribut ∈ S (X) ∧ ( (instance > 0) ∨ (¬ à-générer) ) ALORS
      Générer l'appel fonctionnel de l'attribut synthétisé "attribut" ;
  SI NON
      SI attribut ∈ I(X) ∧ instance = 0 ALORS
          Générer l'appel fonctionnel de l'attribut hérité "attribut" ;
      SI NON     ; la grammaire n'est pas en forme normale
```

```
                Générer la partie-droite de l'équation sémantique correspondant à "attribut,
                production, instance", en remplaçant chaque occurrence α.k (k = 0, ... , n),
                par l'appel récursif : génère-fonction-attribute (α, production, k_i, faux) ;
            FIN_SI
        FIN_SI
FIN génère-fonction-attribut ;
; programme_principal
POUR chaque α ∈ Ensemble_d'attributs_synthétisés FAIRE
    POUR chaque production de la forme p : X_0 → X_1 , ... , X_n FAIRE
        SI α ∈ S(X_0) ALORS génère-fonction-attribut (α, p, 0, vrai); FIN_SI
    FIN_POUR
FIN_POUR ; {Attributs Synthétisés }
POUR chaque α ∈ Ensemble_d'attributs_hérités FAIRE
    POUR chaque production de la forme p : X_0 → X_1 , ... , X_n FAIRE
        POUR k = 1, ... , n FAIRE
            SI α ∈ I(X_k) ALORS génère-fonction-attribut (α, p, k, vrai); FIN_SI
        FIN_POUR
    FIN_POUR
FIN_POUR . { Attributs Hérités }
```

Cet algorithme a été implanté en Le_Lisp 15.2. Bien entendu, toutes les informations relatives à cette génération de fonctions (i.e. les productions de la grammaire, les équations sémantiques, les attributs, ... etc.) sont obtenues par des envois de messages à l'objet issu de la classe "attribute_grammar". L'implantation de l'algorithme de propagation (c.f. chapitre IV) a aussi été réalisée de cette manière. Il est clair que le graphe de dépendances est aussi représenté par des classes et de sous-classes et que les informations nécessaires pour la propagation sont obtenues par des envois de messages aux classes pertinentes.

---

(1) On pourra trouver dans [Jourdan 1984] une description plus détaillée de cet algorithme.

Chapitre VI.

ETUDE COMPARATIVE ENTRE GEODE ET D'AUTRES SYSTEMES SIMILAIRES.

1. INTRODUCTION.

Le but de ce chapitre est de faire une étude comparative entre GEODE et d'autres systèmes similaires. Néanmoins, cette étude n'est pas exhaustive. Nous mettons l'accent sur l'aspect conception d'un environnement de programmation intégré, car nous considérons que c'est dans ce domaine que GEODE propose réellement des nouvelles approches. Nous ne négligerons pas pour autant l'aspect utilisation des environnements générés, bien que cet aspect ne soit pas le plus important de cette étude. Notre étude comparative portera sur les générateurs d'environnements de programmation que nous considérons les plus connus :

- CENTAUR [Clément 1986]
- MENTOR [Donzeau-Gouge 1980]
- CPS [Reps 1981983]
- ALOE [Medina-Mora 1981]
- CEPAGE [Meyer 1984]

Etant donné que le système CENTAUR a repris un certain nombre de formalismes ayant été développés dans le cadre du système MENTOR, ce dernier sera étudié dans la section consacrée à CENTAUR.

Il est important de comprendre que nous n'avons pas la prétention de nous comparer à des systèmes et/ou à des projets d'envergure internationale où plusieurs dizaines de personnes ont travaillé et collaboré pendant des années. Le but de notre étude est de montrer l'état de l'art dans le domaine de la génération d'environnements de programmation, et de montrer quels sont les secteurs où nous considérons que GEODE apporte de nouvelles approches, ou tout simplement des nouvelles manières d'utiliser des formalismes et/ou des méthodes ayant déjà été utilisées.

1.1 GRAMMAIRE ABSTRAITE VS GRAMMAIRE CONCRETE.

Dans le cadre de la génération d'environnements de programmation, le concept de grammaire abstraite est strictement lié à la construction d'arbres, que nous désignerons sous le nom d'*arbres abstraits*. Ces grammaires sont spécifiées par des algèbres de termes; pour un langage de programmation donné, on déclare un ensemble de *sortes*, et un ensemble d'*opérateurs* avec des opérandes sortés. Les opérateurs peuvent être définis avec une arité fixe, ou avec une arité variable. Dans ce dernier cas, les opérateurs sont associatifs et ils sont utilisés pour représenter des listes.

Voici, un extrait de la grammaire abstraite de PL0. Cette grammaire est complètement décrite dans les annexes.

Exemple 1. "Un extrait de la grammaire abstraite du langage PL0".

AssignStat	:	ident × expression → statement
CompoundStat	:	{statement}+ → statement
IfStat	:	condition × statement → statement
Equal	:	expression × expression → condition
NotEqual	:	expression × expression → condition
Less	:	expression × expression → condition
Greater	:	expression × expression → condition
LessEqual	:	expression × expression → condition
GreaterEqual	:	expression × expression → condition
BracketExp	:	→ expression
Plus	:	→ expression
Minus	:	→ expression
Add	:	expression × expression→ expression
Subtract	:	expression × expression→ expression
Multiply	:	expression × expression → expression
Divide	:	expression × expression → expression

ident, number ⊆ expression

Dans cette grammaire ident, number, statement, expression et condition représentent des sortes. Tandis que *AssignStat, CompoundStat, IfStat, Equal, NotEqual, Less, Greater, LessEqual, GreaterEqual, Plus, Minus, Add, Subtract, Multiply* et *Divide* représentent des opérateurs.

De même qu'un arbre de dérivation est associée à la notion de grammaire concrète, un arbre abstrait est lié à celle de grammaire abstraite. Ainsi, l'opérateur :

IfStat : condition × statement → statement

représente le sous-arbre abstrait suivant (Figure 6.1) :

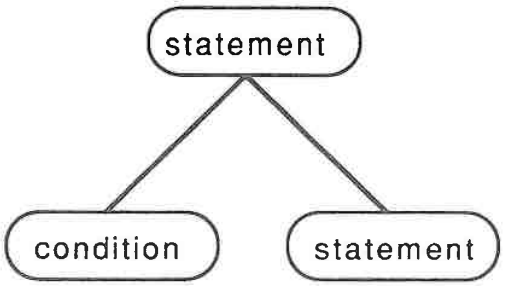


Figure 6.1 "L'opérateur '*IfStat*' et le sous-arbre abstrait qui lui est associé".

Un programme est donc construit en appliquant les divers opérateurs de la grammaire abstraite. Par exemple, considérons le programme PL0 suivant :

```
IF X > 0 THEN
BEGIN
    Y := Y * Z ;
    X := 0
END
```

l'arbre abstrait qui lui est associé est :

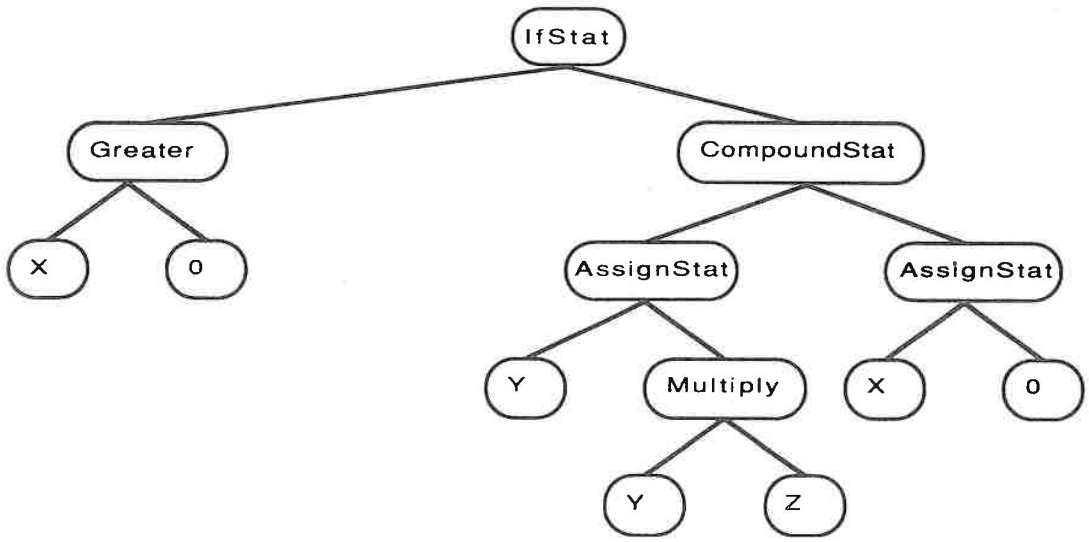


Figure 6.2 "L'arbre abstrait du programme PL0".

Afin de pouvoir analyser plus aisément les notions de grammaire abstraite et grammaire concrète, nous expliciterons la manière dont on peut associer une grammaire abstraite à une grammaire concrète.

Soit la grammaire concrète C = (Vt, Vn, P, St). On lui associe une grammaire abstraite A = (S, O, P, St) de la manière suivante :

Soit la production concrète $A \rightarrow \alpha$, où $\alpha \in (Vt \cup Vn)^*$, Y_i = (Vn ∪ Terminaux génériques) et a_j les symboles terminaux non-génériques apparaissant dans α, $\forall i=1,...,n$ et $\forall j=1,...,m$.

Si $i \geq 1$ alors, la production abstraite équivalente est :

$op : Y_1 \times Y_2 \times ... \times Y_n \rightarrow A$

où $Y_i \in S$ et $op \in O$.

Si i = 1 alors, il y a deux possibilités,

op : $Y_1 \rightarrow A$, ou bien,

op : $\rightarrow A$.

Ces transformations conduisent à une première grammaire abstraite. Celle-ci peut encore être transformée jusqu'à ce que l'on obtienne une grammaire adaptée aux besoins existants. L'obtention de la grammaire abstraite de PL0 dans le cadre de GEODE est présentée dans les annexes.

La grammaire abstraite est une algèbre de termes où S est l'ensemble de sortes, O est l'ensemble d'opérateurs définis sur les sortes et P est la signature. St représente l'axiome de la grammaire abstraite.

Exemple 2.

Soit la grammaire concrète C :

1. E → E + T | E - T | T
2. T → T * F | T / F | F
3. F → Id | (E)

On peut associer à C une grammaire abstraite A_1 :

1. Add : $E \times T \rightarrow E$
2. Sub : $E \times T \rightarrow E$
   $T \subseteq E$
3. Mult : $T \times F \rightarrow T$
4. Div : $T \times F \rightarrow T$
   $F \subseteq T$
5. Id : $ident \rightarrow F$
6. Brack : $E \rightarrow F$
   $ident \subseteq F$

Il est à remarquer que cette grammaire abstraite a été obtenue en utilisant la transformation op : $Y_1 \rightarrow A$, expliquée ci-dessus, sur la troisième production de la grammaire concrète.

L'ensemble de sortes est S = {E, T, F, ident}; l'ensemble d'opérateurs est O = {Add, Sub, Mult, Div, Id, Brack}; l'axiome étant la sorte E. L'ensemble P de productions est représenté par la signature de l'algèbre.

Considérons la chaîne : (a * (b + c))

L'arbre de dérivation associé à cette chaîne est :

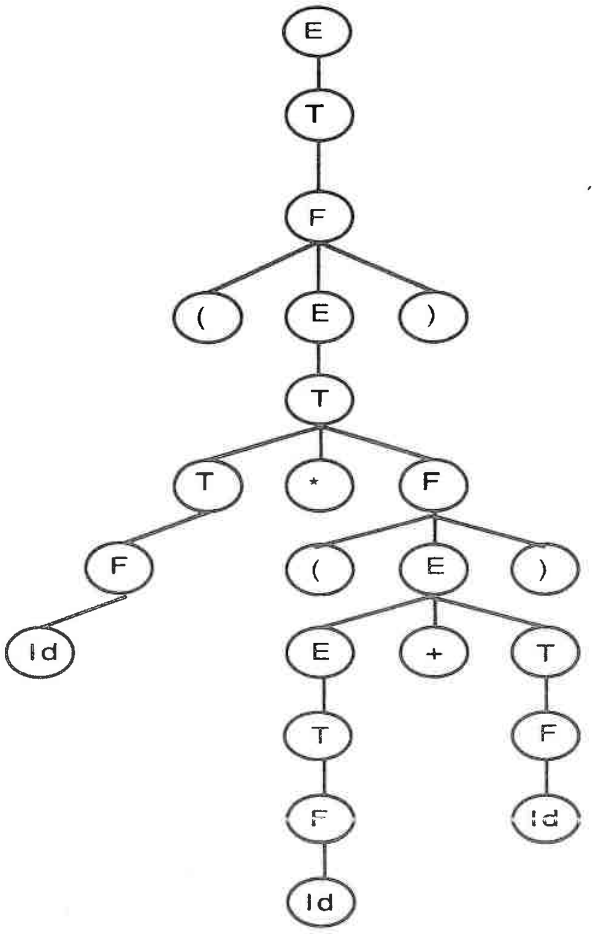


Figure 6.3 "L'arbre de dérivation de la chaîne ' (a * (b + c))' ".

Tandis que l'arbre abstrait associé à cette même chaîne est :

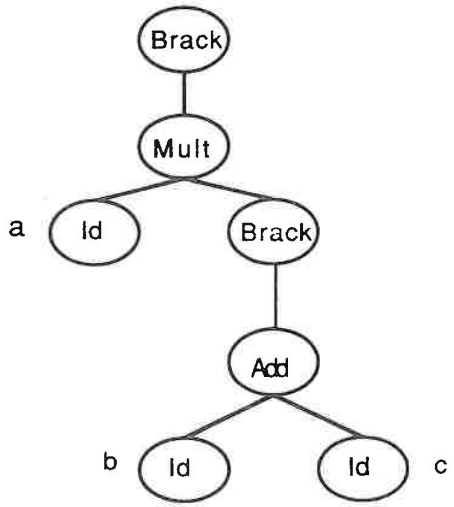


Figure 6.4 "L'arbre abstrait de la chaîne ' (a * (b + c))' issu de la grammaire A_1".

Il est important de noter que la taille d'un arbre abstrait est nettement moins importante que celle d'un arbre de dérivation. Cependant, il faut comprendre qu'une même grammaire concrète peut être associée à plusieurs grammaires abstraites, et que dans certains cas, il peut y avoir des pertes de sémantique.

Par exemple, une deuxième grammaire abstraite A_2 associée à la grammaire concrète C est :

1. Add : $E \times T \rightarrow E$
2. Sub : $E \times T \rightarrow E$
   $T \subseteq E$
3. Mult : $T \times F \rightarrow T$
4. Div : $T \times F \rightarrow T$
   $F \subseteq T$
   ident $\subseteq F$
   $E \subseteq F$

Cette grammaire abstraite a été obtenue en utilisant la transformation op : → A, sur la troisième production de la grammaire concrète.

Si l'on construit un nouvel arbre abstrait pour la chaîne ci-dessus, nous obtenons l'arbre suivant :

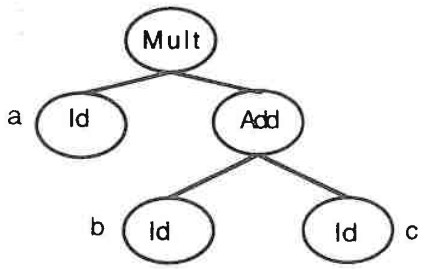


Figure 6.5 "L'arbre de dérivation de la chaîne ' (a * (b + c))' issu de la grammaire A_2".

S'il est vrai que cet arbre a une taille encore inférieure à celle de l'arbre abstrait qui a été construit précédemment, on doit remarquer qu'avec cet arbre l'on perd la sémantique d'association des opérateurs arithmétiques. Cette sémantique était implicitement définie dans la grammaire concrète, mais aussi dans la première grammaire abstraite que nous avons construit.

Pour illustrer ceci, considérons la chaîne : a * (b + c).

Cette chaîne est représentée par le même arbre abstrait que la chaîne (a * (b + c)), ce qui signifie que la sémantique définie implicitement par les parenthèses extérieures a été perdue. Remarquons aussi que cette sémantique est conservée par la première grammaire abstraite définie.

S'il est vrai que dans cet exemple la perte de sémantique n'aucune influence en ce qui concerne la valeur finale de l'expression arithmétique, elle devient très gênante dans le cadre de la décompilation de l'arbre abstrait. Ainsi, l'arbre abstrait de la figure 6.5 peut être décompilé de plusieurs manières :

- (a * (b + c))
- a * (b + c)
- ((a) * (b + c)), ... etc.
- et même la chaîne "a * b + c" !!!

Remarquons que pour détecter que la dernière expression ne peut pas être la décompilation de l'arbre 6.5 le décompilateur doit connaître la précédence des opérateurs arithmétiques. Ainsi, si le décompilateur sait que le symbole "+" est moins prioritaire que le symbole "*", il décompilera cet arbre comme la chaîne "a * (b + c))". Il n'empêche qu'il faudrait encore résoudre le problème concernant la décompilation des autres chaînes.

Dans le cadre de la génération d'environnements de programmation, faut-il donc utiliser la grammaire abstraite ou la grammaire concrète ?

Répondre a cette question est difficile, car il faut tenir compte de l'influence de chaque type de grammaire par rapport à l'utilisation mais aussi par rapport à la conception des environnements générés. Si nous nous plaçons du côté de l'utilisation des environnements de programmation, il nous paraît évident que la grammaire abstraite est le bon choix, car elle permet de représenter les programmes des utilisateurs avec des arbres abstraits, qui grâce à leur taille, sont beaucoup plus facilement manipulables que les arbres de dérivation. Lorsque que nous constatons que toute manipulation d'un utilisateur sur un programme est traduite, au niveau du système, par des greffes, élagages et parcours d'arbres, la taille de ces arbres nous semble un facteur déterminant.

Néanmoins, il y a la partie conception des environnements générés. Une première constatation est qu'en général le concept de grammaire concrète est beaucoup plus connu que celui de grammaire abstraite. La majorité de manuels décrivant un langage de programmation contiennent des diagrammes de syntaxe et/ou des règles sous la forme BNF. Une deuxième constatation est que construire une grammaire abstraite n'est pas facile, car il faut tenir compte de la manière dont cette grammaire influence les différents outils de l'environnement. Cette influence n'est pas prise en compte par la méthode de transformation grammaire concrète - grammaire abstraite que nous avons donné ci-dessus. Nous avons illustré

l'influence d'une grammaire abstraite sur les outils d'un environnement de programmation, en montrant ce qui risque d'arriver lors de la décompilation de l'arbre abstrait 6.5. Il va de soi que dans le cadre de l'utilisation d'autres outils de l'environnement, il faudra tenir compte d'autres aspects. En outre, il se peut que l'adjonction d'un nouvel outil implique l'apparition de nouveaux besoins entraînant des modifications de la grammaire abstraite et donc des possibles modifications des outils existants.

Une dernière constatation est que même si l'on adopte la notion de grammaire abstraite, il faudra utiliser la grammaire concrète, ne serait-ce que pour effectuer l'analyse syntaxique d'un texte tapé directement par un utilisateur et, bien évidemment, pour la décompilation d'un arbre abstrait. Un langage de programmation devra donc être décrit en utilisant deux formalismes différents.

A l'heure actuelle GEODE utilise exclusivement la grammaire concrète. Ceci pour plusieurs raisons. La première est que l'un des buts de GEODE (c.f. Introduction et Chapitre II) est d'utiliser un seul et unique formalisme pour la description d'un langage de programmation. Si l'on adopte l'approche grammaire abstraite - grammaire concrète, on aura deux formalismes différents, bien que liés d'une certaine manière. La deuxième raison est que, si nous avons une idée précise de la manière dont une grammaire abstraite affecte un éditeur structuré, nous n'avons pas d'idées précises quant à l'influence de ces grammaires sur d'autres outils comme les interprètes ou les compilateurs incrémentaux, par exemple.

Nous considérons, cependant, que la meilleure solution serait de trouver un compromis entre les deux formalismes. Bien que ce compromis ne soit pas encore totalement adopté, il apparaît déjà dans certains systèmes comme CENTAUR [Clément 1986] et CEPAGE [Meyer 1984]. Nos idées sur le domaine seront exposées dans la conclusion.

2. ETAT DE L'ART.

2.1 LE SYSTEME CENTAUR.

Le projet GIPE "Generation of Interactive Programming Environments" s'insère dans le cadre du projet ESPRIT de la CEE [Clément 1986]. Le but du projet GIPE est de générer des environnements de programmation interactifs à partir de la définition formelle d'un langage de programmation. Cette définition est constituée d'une grammaire abstraite, d'une grammaire concrète et par un ensemble de règles décrivant le paragraphage, la sémantique statique et la sémantique dynamique du langage concerné. Le langage utilisé pour construire cette définition est appelé LDF (Language Definition Formalism). Bien entendu, il est possible de générer des environnements pour manipuler LDF.

La figure 6.6 montre les relations qui existent entre la définition formelle d'un langage et les différents composants de l'environnement de programmation généré par CENTAUR.

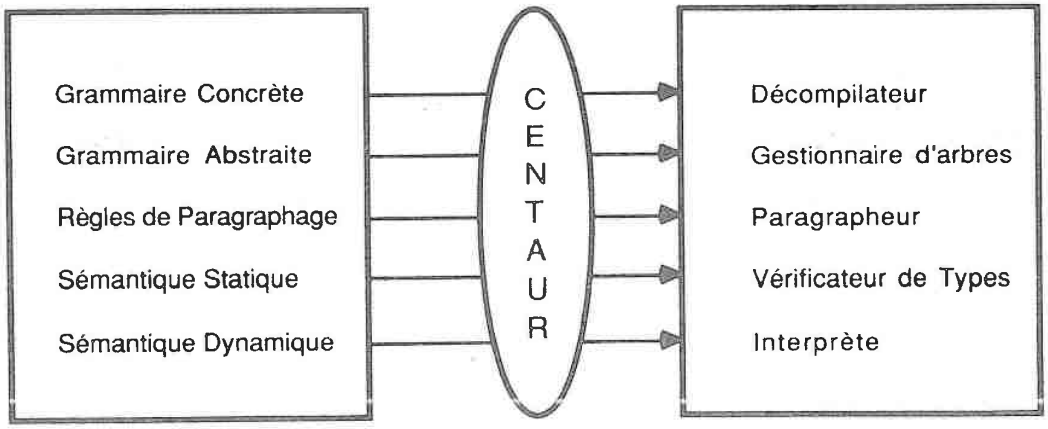


Figure 6.6 "Les relations entre la définition formelle d'un langage et les outils des environnements générés par CENTAUR".

Du point de vue architecture, CENTAUR est composé de trois modules principaux :

- Le "kernel", qui s'occupe de la représentation et manipulation d'objets structurés;
- Le module de spécification qui se charge de la définition des aspects syntaxiques et sémantiques du langage de programmation concerné;
- L'interface utilisateur qui s'occupe de la communication interactive entre les utilisateurs et le système.

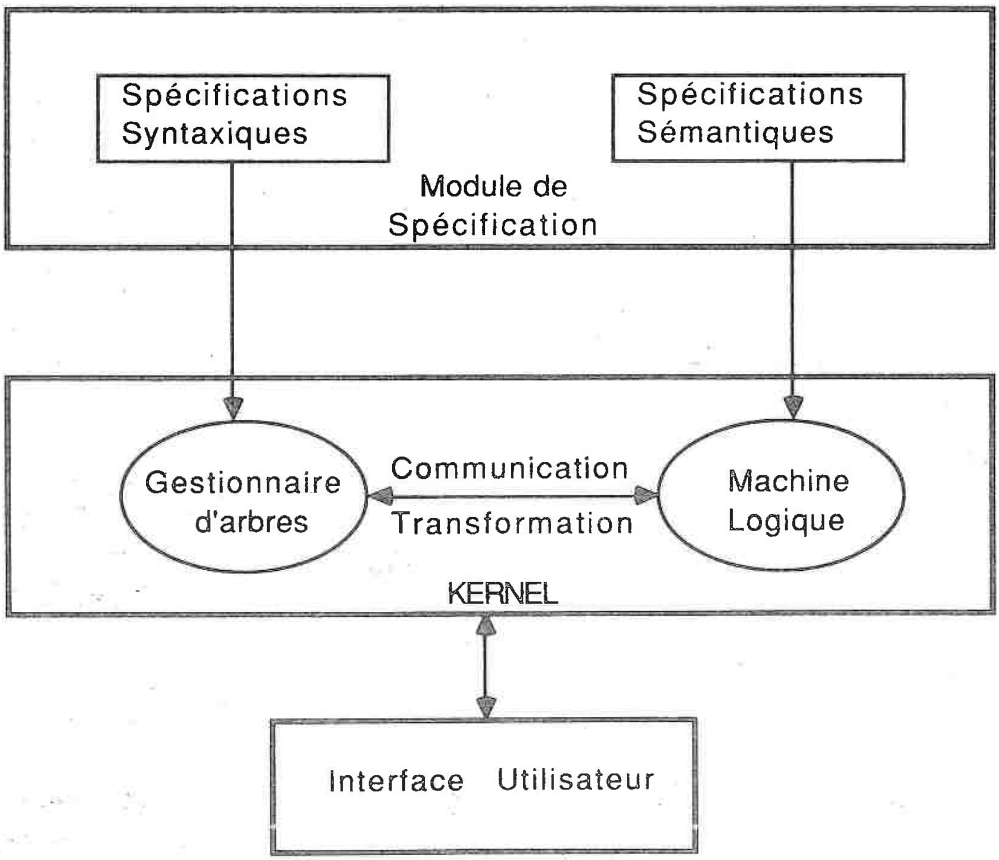


Figure 6.7 "L'architecture du système CENTAUR".

Le gestionnaire d'arbres du "kernel" est chargé des aspects concernant la manipulation d'objets structurés tels que les modules de définition du langage ou les programmes écrits en ce langage. Le gestionnaire d'arbres possède des primitives pour :

- Le parcours d'objets structurés;
- La sélection et la mémorisation de positions à l'intérieur de ces objets;
- La recherche structurelle ou l'identification;
- La modification et la construction des primitives;
- La documentation et la manipulation d'erreurs.

La machine logique (écrite en PROLOG) concerne les actions sémantiques pouvant être effectuées sur des objets structurés. Elle est utilisée lors de l'exécution de programmes générés par le module de spécification et inclue des interprètes, des compilateurs, des "optimiseurs", des outils de débogage, de trace, etc. La machine logique est aussi utilisée pour maintenir la cohérence des environnements générés.

Le module de spécification s'occupe de la syntaxe abstraite et de la syntaxe concrète d'un langage de programmation, ainsi que de sa sémantique. En ce qui concerne la syntaxe, deux approches sont comparés actuellement dans CENTAUR. La première approche consiste à proposer un formalisme différent pour décrire les trois composantes de la spécification syntaxique d'un langage : un système de types pour la grammaire abstraite, des règles BNF pour la grammaire concrète et finalement des règles de paragraphage pour la décompilation. Cette approche est représentée par le formalisme METAL [Kahn 1983]. La deuxième approche (qui nous semble être la plus prometteuse) définit ces trois composantes avec un seul formalisme. L'idée est de déduire les grammaires concrète et abstraite et les règles de paragraphage, à partir d'une spécification SDF (Syntax Definition Formalism) [Heering 1986]. Les aspects lexicographiques du langage sont aussi pris en compte dans la spécification SDF. En plus, une spécification SDF sert aussi à générer un analyseur syntaxique capable d'analyser une catégorie très large de grammaires à contexte libre [Rekers 1986]. Cet analyseur est utilisé pour vérifier la correction syntaxique d'un texte entré directement par l'utilisateur de l'environnement de programmation.

Afin d'illustrer la forme des spécifications METAL et SDF, nous allons revenir sur la

grammaire de l'exemple 2.

Soit la grammaire concrète C, une grammaire engendrant des expressions arithmétiques parenthésées :

1. E → E+T | E-T | T
2. T → T*F | T/F | F
3. F → Id | (E)

et la grammaire abstraite A_3 associée à C :

1. Add : E × E → E
2. Sub : E × E → E
3. Mult : E × E → E
4. Div : E × E → E
5. Ident : Id → E

METAL est le formalisme de définition de syntaxe du système MENTOR [Donzeau-Gouge 1975, 1980]. Une définition METAL spécifie : la lexicographie, la grammaire concrète et la grammaire abstraite d'un langage de programmation. L'implantation UNIX de METAL transforme cette définition en une définition LEX [Lesk 1979] et YACC [Johnson 1979].

La lexicographie du langage concerné est spécifiée dans le langage du générateur d'analyseurs lexicographiques du système hôte. Sous UNIX, le lexicographie est définie par des expressions régulières LEX. La grammaire concrète est définie en utilisant des règles du style BNF. Les symboles non-terminaux sont écrits entre "<" et ">". L'utilisateur spécifie aussi, dans cette définition BNF, l'arbre abstrait associé à chaque production de grammaire concrète. Finalement, la grammaire abstraite est définie en termes de sortes et d'opérateurs (phyla et opérateurs dans la terminologie METAL).

La spécification METAL de la grammaire des expressions arithmétiques est :

```
DEFINITION OF Expressions IS
    CHAPTER 'exp-rules'
        RULES
            <exp>           ::=  <term> ;
                <term>
            <exp>           ::=  <exp> #+ <term> ;
                add (<exp>, <term>)
            <exp>           ::=  <exp> #- <term> ;
                sub (<exp>, <term>)
            <term>          ::=  <factor> ;
                <factor>
            <term>          ::=  <term> #* <factor> ;
                mult (<term>, <factor>)
            <term>          ::=  <term> #/ <factor> ;
                div (<term>, <factor>)
            <factor>        ::=  <ident> ;
                <ident>
            <factor>        ::=  #( <exp> #) ;
                <exp>
            <ident>         ::=  %ID ;
                id-atom (%ID)
        ABSTRACT SYNTAX
            add             ->   EXP EXP ;
            sub             ->   EXP EXP ;
            mult            ->   EXP EXP ;
            div             ->   EXP EXP ;
            ident           ->   implemented as IDENT ;
            EXP             ::=  add sub mult div ;
            ID              ::=  ident ;
        END CHAPTER ;
END DEFINITION
```

SDF est un formalisme de spécification de langages basé sur une extension des

spécifications algébriques. Afin d'illustrer la définition de la syntaxe d'un langage à l'aide de SDF nous allons revenir sur l'exemple 2.

La définition SDF associée est la suivante :

```
MODULE Expressions
BEGIN
   LEXICAL SYNTAX
      SORTS Ident
   CONTEXT-FREE SYNTAX
      SORTS E
      PRIORITY ("+", "-") < ("*", "/")
      FUNCTIONS
         E "+" E  →  E        {par, left-assoc}
         E "-" E  →  E        {par, left-assoc}
         E "*" E  →  E        {par, left-assoc}
         E "/" E  →  E        {par, left-assoc}
         Ident    →  E
END Expressions
```

Où la sorte "Ident" est déclarée de la manière suivante :

```
MODULE Lexical-constants
BEGIN
   LEXICAL SYNTAX
      SORTS LETTER, ID, ID-TAIL
      FUNCTIONS
         [a-z]              →  LETTER
         [a-z0-9]           →  ID-TAIL
         LETTER ID-TAIL*    →  ID
END Lexical-constants
```

Il est à noter que SDF fournit des mécanismes pour définir l'associativité et/ou la

priorité des opérateurs, ainsi que la possibilité d'utiliser des expressions régulières pour la description des aspects lexicographiques d'un langage.

En ce qui concerne la sémantique d'un langage de programmation, CENTAUR travaille avec trois spécifications :

- Une spécification de la sémantique statique du langage, pour générer un vérificateur de types;
- Une spécification de la sémantique dynamique du langage, pour générer un interprète;
- Une spécification de la traduction du langage en un code machine quelconque et une spécification de la sémantique dynamique de ce code machine, pour générer un compilateur.

Ces trois spécifications seront décrites en utilisant soit le mécanisme appelé "Sémantique Naturelle" [Kahn 1986], représenté par le langage TYPOL [Clément 1985], soit le mécanisme appelé ASF (Algebraic Specification Formalism) [Bergstra 1986]. A notre connaissance seules les spécifications TYPOL peuvent être exécutées actuellement.

En fin, l'interface utilisateur de CENTAUR est bien entendu une interface hautement interactive. L'interface est un système de multi-fenêtres qui est guidé par des menus. Il est possible d'ouvrir plusieurs fenêtres pouvant contenir des objets structurés différents, ou bien, plusieurs vues du même objet. L'utilisateur peut manipuler les fenêtres ainsi que leurs contenus. Les principaux constituants de l'interface sont :

- Un gestionnaire d'entrée, pour le clavier et la souris;
- Un gestionnaire de sortie qui inclue un gestionnaire de fenêtres, pour la création et la manipulation de fenêtres de l'utilisateur et du système;
- Un gestionnaire de commandes, qui s'occupe de l'état de l'interface;
- Un interprète de commandes, pour exécuter les commandes de l'utilisateur et pour informer celui-ci des erreurs éventuelles;
- Des machines abstraites de formattage, pour s'occuper de toutes les sorties possibles du système (i.e. sorties textuelles, sorties graphiques, etc).

La figure 6.8 montre les composantes de l'interface utilisateur du système CENTAUR.

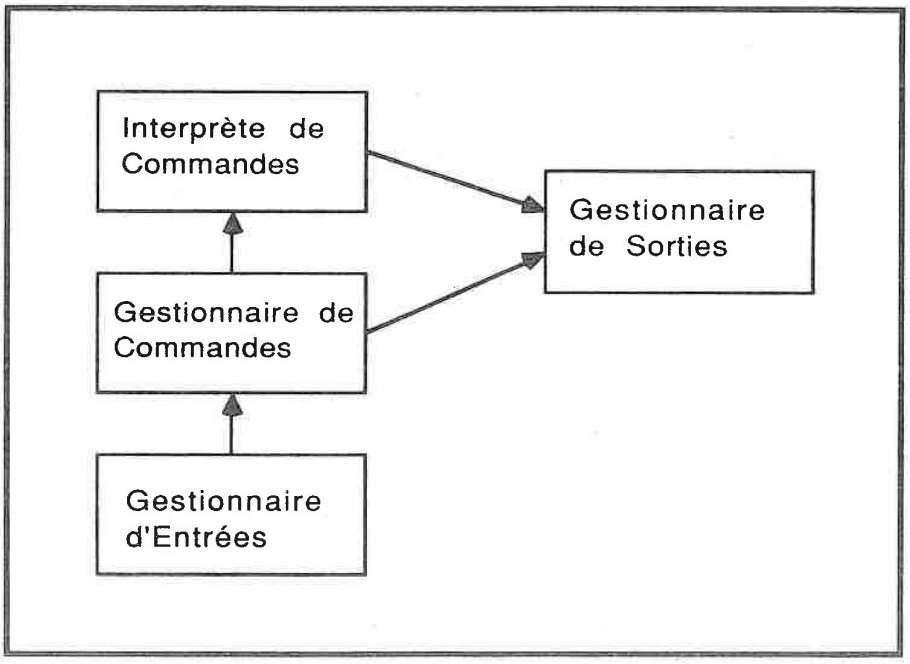


Figure 6.8 "L'interface utilisateur du système CENTAUR".

2.2 LE SYSTEME CPS.

Le système CPS (Cornell Program Synthesizer) est un outil pour la construction d'éditeurs basés sur et paramétrés par un langage de programmation donné [Reps 1984], [Reps 1985]. Le langage de programmation concerné est spécifié par : une grammaire concrète, une grammaire abstraite, un ensemble de relations contextuelles et un ensemble de règles de paragraphage. Cette spécification est faite à l'aide du langage SSL (Synthesizer Specification Language), lequel est basé sur deux concepts : l'algèbre de termes et les grammaires attribuées.

Du point de vue architecture, CPS est un système utilisant le mécanisme de génération de tables (c.f. figure 6.9) .

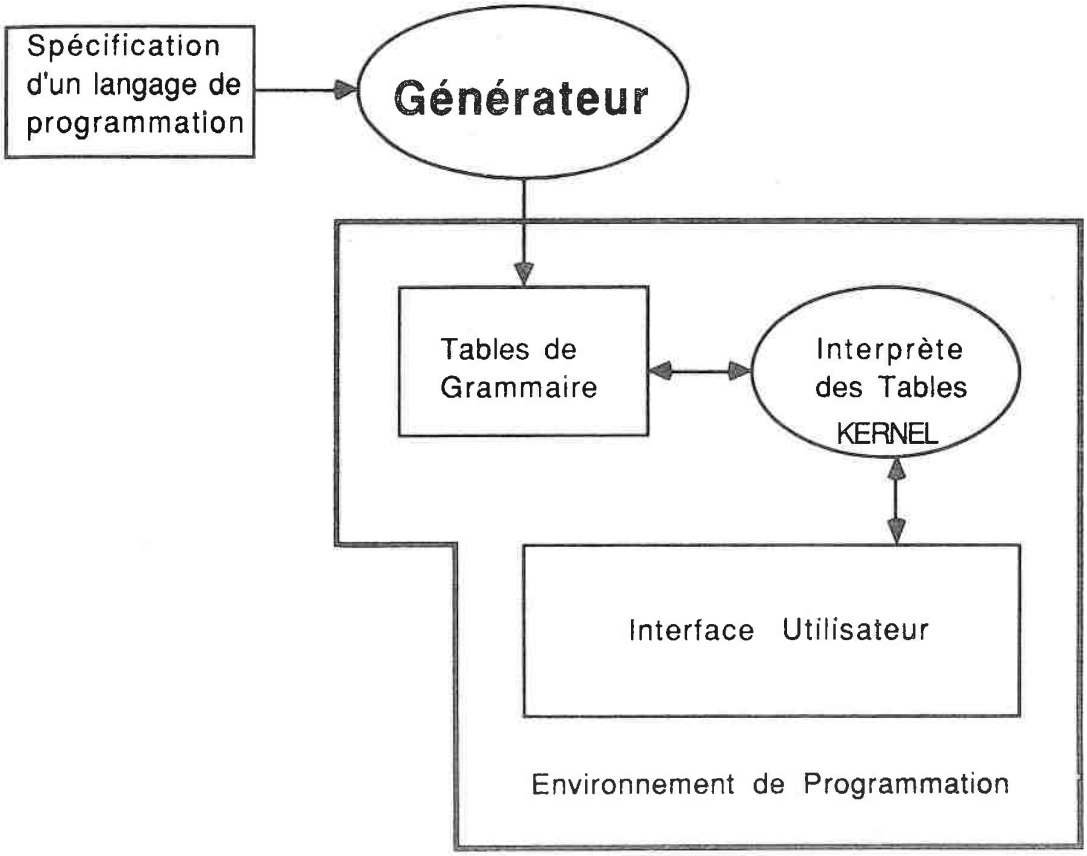


Figure 6.9 "L'architecture d'un système utilisant la génération de tables".

Le système CPS possède deux composantes :

- un traducteur, qui prend une spécification SSL en entrée et qui produit des tables de grammaire en sortie;
- un "kernel" qui s'occupe de la représentation et la manipulation d'arbres abstraits. Le "kernel" prend en entrée des commandes issues de la souris ou du clavier, et effectue les opérations nécessaires sur l'arbre courant.

Un programme "shell" UNIX nommé *sgen* s'occupe d'appeler le traducteur et de

produire un éditeur structuré à partir de tables de grammaire générées par ce dernier.

Une spécification SSL est constituée d'une liste de déclarations [Reps 1985]. La grammaire abstraite est définie par trois types de déclarations qui concernent, respectivement, les sortes, les propriétés des symboles (i.e. s'il s'agit d'un symbole optionnel, d'une liste d'éléments, etc ...) et l'axiome. Les relations contextuelles entre les différents sortes sont définies par des déclarations contenant des équations sémantiques, des attributs et des fonctions utilisées par les équations sémantiques. La décompilation est représentée par une liste de déclarations associées à chaque opérateur. La grammaire concrète est définie par une liste de déclarations appelées "parsing declarations". Ces déclarations définissent aussi la relation grammaire abstraite - grammaire concrète. La précédence et la priorité des symboles terminaux peuvent être définies par des déclarations de précédence. Enfin, les déclarations de transformation spécifient la manière dont un objet doit être restructuré.

A continuation nous donnerons quelques exemples illustrant une spécification SSL. Pour plus de détails on pourra consulter [Reps 1985].

Soit la grammaire concrète C_2 :

1. E → E + T | E - T | T
2. T → T * F | T / F | F
3. F → Const | (E)

et la grammaire abstraite A_4 associée à la grammaire C_2

1. Add : $E \times E \rightarrow E$
2. Sub : $E \times E \rightarrow E$
3. Muit : $E \times E \rightarrow E$
4. Div : $E \times E \rightarrow E$
5. Const : $INT \rightarrow E$

La déclaration SSL correspondante est :

```
E  :  Null()
   |  Add, Sub, Mult, Div (E E)
   |  Const (INT)
   ;
```

Où Null() est un terme indiquant que le symbole E n'a pas encore été développé. Ce terme est associé aux productions de la forme E → ¥ (c.f. chapitre III).

La sorte INT est déclarée de la manière suivante :

```
INT  :  Integer < [0-9]+ >;
```

Les relations contextuelles entre les différentes sortes sont définies par une grammaire attribuée. Dans les éditeurs générés par le système CPS, les relations contextuelles sont les relations inter-sortes qui dépendent de la forme de l'arbre abstrait. L'exemple suivant montre la définition de la relation contextuelle qui associe à chaque arbre abstrait représentant un expression arithmétique, la valeur décimale de cette dernière.

Exemple 3. "La valeur décimale d'une expression arithmétique".

```
E  :  Null ()     { E.v = 0; }
   |  Add         { E$1.v = E$2.v + E$3.v; }
   |  Sub         { E$1.v = E$2.v - E$3.v; }
   |  Mult        { E$1.v = E$2.v + E$3.v; }
   |  Quot        { LOCAL STR error;
                    error = (E$3.v == 0) ? "< --DIVISION BY ZERO-->" : "";
                    E$1.v = (E$3.v == 0) ? E$2.v : (E$2.v / E$3.v);
                  }
   |  Const       { E$1.v = INT; }
   ;
```

Où l'attribut "v" est déclaré de la forme suivante :

E {**SYNTHESIZED** INT v}

Cette déclaration signifie que l'attribut synthétisé "v" associé à la sorte E est de type entier. INT étant une sorte prédéfinie.

La manière dont un arbre abstrait est décompilé est décrite à l'aide des déclarations appelées "unparsing declarations". Ces déclarations définissent donc la manière dont un symbole devra être affiché, mais aussi quels sont les nœuds de l'arbre abstrait qui peuvent être développés. L'exemple suivant montre un schéma de décompilation associé à la grammaire abstraite définie ci-dessus.

Exemple 4. "La décompilation associée à la grammaire A_4".

```
E   :   Null ()         [ @ ::= "< exp >" ]
    |   Add              [^ ::= "(" ^ " + " ^ ")" ]
    |   Sub              [^ ::= "(" ^ " - " ^ ")" ]
    |   Mult             [^ ::= "(" ^ " * " ^ ")" ]
    |   Quot             [^ ::= "(" ^ " / " error ^ ")" ]
    |   Const            [@ ::= ^ ]
    ;
```

Chaque caractère @ ou ^ représente l'occurrence d'une sorte de la production en question. Le symbole apparaissant en partie gauche des règles est précédé par ::=. Les instances d'attribut peuvent aussi apparaître dans le schéma de décompilation. Elles sont alors affichées selon une représentation qui est définie par le concepteur de l'éditeur.

La grammaire concrète est décrite par des déclarations appelées "parsing declarations". Ces déclarations sont divisées en deux groupes : Les déclarations dites de spécification, qui définissent la grammaire concrète elle-même; et les déclarations dites de traduction, qui définissent les relations grammaire concrète - grammaire abstraite. L'exemple suivant montre les "parsing declarations" associées à notre exemple d'expressions arithmétiques.

Exemple 5. "La définition de la grammaire concrète C_2 et sa relation avec la grammaire abstraite A_4".

```
LEFT    '+'  '-';
LEFT    '*'  '/';

E   ::=  (E '+' E)        { E$1.abs = Add (E$2.abs, E$3.abs); }
     |   (E '-' E)        { E$1.abs = Sub (E$2.abs, E$3.abs); }
     |   (E '*' E)        { E$1.abs = Mult (E$2.abs, E$3.abs); }
     |   (E '/' E)        { E$1.abs = Quot (E$2.abs, E$3.abs); }
     |   ('(' E ')')      { E$1.abs = E$2.abs; }
     |   (INTEGER)        { E$1.abs = Const (STRtoINT (INTEGER)); }
     ;
```

La relation grammaire concrète - grammaire abstraite est définie par les équations sémantiques. Le mot clé "left" indique l'associativité des opérateurs qui sont déclarés dans un ordre de priorité croissante.

Finalement, les déclarations de transformation sont illustrées par l'exemple 6. Pour des questions de simplicité, seules l'addition et la multiplication ont été prises en compte.

Exemple 6. "La restructuration d'expressions arithmétiques selon les lois de commutativité et associativité".

```
TRANSFORM E
    ON "factor-left" Add (Mult (a, b)), Mult (a, c)) : Mult (a, Add (b, c)) ,
    ON "factor-right" Add (Mult (b, a), Mult (c, a)) : Mult (Add (b, c), a) ,
    ON "distribute-left" Mult (a, Add (b, c)) : Add (Mult (a, b), Mult (a, c)) ,
    ON "distribute-right" Mult (Add (b, c), a) : Add (Mult (b, a), Mult (c, a)) ,
    ON "commute" Add (a, b) : Add (b, a) ,
    ON "commute" Mult (a, b) : Mult (b, a) ,
    ;
```

2.3 LE SYSTEME ALOE.

Le système ALOE (A Language Oriented Editor) est un outil générant des éditeurs structurés [Medina-Mora 1981], qui s'insère dans le cadre du projet d'atelier logiciel GANDALF [Habermann 1979]. Ces éditeurs sont générés à partir d'une description grammaticale du langage de programmation à manipuler. ALOE possède l'architecture d'un système générateur de tables (c.f. figure 6.9).

Dans ALOE la description d'un langage de programmation est constituée principalement de :

- La grammaire abstraite du langage;
- Des schémas de décompilation;
- Des actions sémantiques associés à chaque opérateur du langage.

L'exemple qui suit montre la spécification ALOE des expressions arithmétiques (c.f. grammaires C_2 et A_4).

Exemple 7. "La spécification ALOE des expressions arithmétiques".

LANGUAGE NAME : Expression
ROOT OPERATOR : E

```
{               /* OPERATEURS  TERMINAUX */

Const   =   {c}                     /* type de terminal */
        |   (0) "@c"                /* schéma de décompilation */
        |   action: aINT            /* action sémantique associée */
        |   synonym: "#" ;          /* synonyme associé au terminal */

}

{       /* OPERATEURS NON-TERMINAUX */
```

```
Add     =   E E
        |   (0) @1 + @2
        |   action: <none>
        |   synonym: "+"
        |   precedence: 1
        |   Non-filenode;

Sub     =   E E
        |   (0) @1 - @2
        |   action: <none>
        |   synonym: "-"
        |   precedence: 1
        |   Non-filenode;

Mult    =   E E
        |   (0) @1 * @2
        |   action: <none>
        |   synonym: "*"
        |   precedence: 2
        |   Non-filenode;

Div     =   E E
        |   (0) @1 / @2
        |   action: NonZero
        |   synonym: "/"
        |   precedence: 2
        |   Non-filenode;
}

{       /* CLASSES */

E       =   Const Add Sub Mult Div;
}
```

Des vérifications de la sémantique statique du langage de programmation en question peuvent être effectuées à l'aide des actions sémantiques. On pourra consulter [Medina-Mora 1981] pour plus de précisions sur l'utilisation d'ALOE.

2.4 LE SYSTEME CEPAGE.

Le système CEPAGE [Meyer 1987b] est un générateur d'éditeurs structurés. Le système est dirigé par la définition grammaticale LDL (Language Description Language) [Meyer 1987c] du langage concerné. Les éditeurs générés peuvent construire de documents de deux manières : en utilisant des menus guidant le développement de symboles non-terminaux de la grammaire, ou bien, en entrant du texte directement, comme sous un éditeur de textes classique. La figure 6.10 montre l'architecture de CEPAGE.

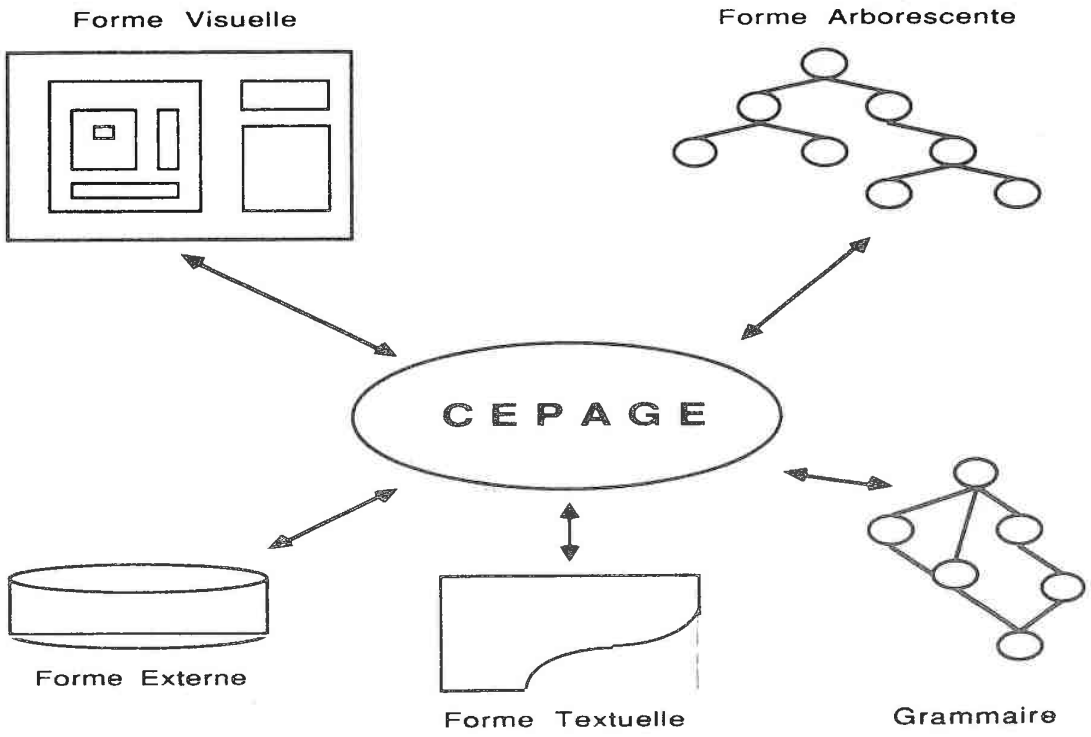


Figure 6.10 "L'architecture du système CEPAGE".

La "forme visuelle" représente l'interface entre le système CEPAGE et les utilisateurs. Cette interface est hautement interactive et est constituée d'un gestionnaire de fenêtres et de menus qui permettent à l'utilisateur de manipuler son(ses) programme(s). La "forme

arborescente" est la représentation CEPAGE des programmes sous forme d'arbres abstraits. La "forme externe" permet de sauvegarder un programme utilisateur sur un dispositif de stockage externe (p.e. disques, bande magnétique, etc ...). La "forme textuelle" peut être généré lorsqu'il s'agit d'un document complètement développé. Cette forme peut alors être utilisée par d'autres outils extérieures à CEPAGE. Enfin, la syntaxe du langage considéré est décrite par une grammaire abstraite-concrète qui est représentée sous forme d'un graphe.

Pour générer un éditeur structuré, on doit d'abord construire une spécification LDL. Une spécification LDL est une séquence de constructeurs (e.a. "constructs"). Un constructeur décrit la forme des instances d'un "type syntaxique". Par exemple, le constructeur suivant décrit l'instruction "while" de PASCAL [Meyer 1987b] :

Exemple 7. "Le constructeur CEPAGE représentant l'instruction 'while' de PASCAL".

```
CONSTRUCT while_loop                -- Nom du constructeur.
SHORT "while ..."                   -- Abréviation utilisée dans les menus.
AGGREGATE                           -- Définition de la grammaire abstraite associée
        test : Expression ;         -- au constructeur.
        body : Instruction
FORMAT                              -- Définition de la grammaire concrète associée
        "while" test "do" body      -- au constructeur. FORMAT spécifie aussi le
END                                 -- schéma de décompilation du constructeur.
```

Un constructeur est constitué de plusieurs éléments :

- un nom abrégé, "while ...", qui est utilisé pour représenter le constructeur dans les menus où l'utilisateur sélectionnera le développement de son choix. Ce nom sera aussi affiché lors de la décompilation, si le symbole syntaxique en question n'a pas encore été développé. Mot clé **SHORT** ;
- une description de la syntaxe abstraite qui est représentée par des "aggregates". Mot clé **AGGREGATE** ;
- une description de la syntaxe concrète qui est aussi utilisée lors de la décompilation d'un arbre abstrait. Mot clé **FORMAT** .

Si nous revenons sur l'exemple des expressions arithmétiques, nous obtenons la spécification LDL suivante :

Exemple 8. "La spécification CEPAGE (LDL) des expressions arithmétiques".

```
GLOBAL
        indent: 3                       -- définition de l'indentation souhaitée.
        root: E                         -- axiome de la grammaire abstraite.
        pagewidth: 80                   -- longueur maximum des lignes affichables par
                                        -- l'éditeur.
END

CONSTRUCT Add
SHORT "Add ..."
AGGREGATE
        opérateur_1  :  E ;
        opérateur_2  :  E
FORMAT
        opérateur_1 "+" opérateur_2
END

CONSTRUCT Sub
SHORT "Sub ..."
AGGREGATE
        opérateur_1  :  E ;
        opérateur_2  :  E
FORMAT
        opérateur_1 "-" opérateur_2
END
```

```
CONSTRUCT Mult
SHORT "Mult ..."
AGGREGATE
        opérateur_1  :  E ;
        opérateur_2  :  E
FORMAT
        opérateur_1 "*" opérateur_2
END

CONSTRUCT Div
SHORT "Div ..."
AGGREGATE
        opérateur_1  :  E ;
        opérateur_2  :  E
FORMAT
        opérateur_1 "/" opérateur_2
END

CONSTRUCT Const
SHORT "ConstanteEntière ..."
ATOMIC
FORMAT
        [ '+' | '-' ] + ('0' .. '9')
END

CONSTRUCT E
SHORT "Expression ..."
CHOICE
        Add, Sub, Mult, Div, Const
END
```

LDL offre aussi la possibilité de définir de constructeurs "listes" et/ou de constructeurs optionnels, ainsi que de constructeurs associés à la lexicographie du langage considéré (p.e. constructeur "Const").

Il est à noter qu'une définition LDL est seulement une description de la syntaxe du langage traité. Tout ce qui concerne l'affichage est le formattage de textes sur l'écran des utilisateurs est automatiquement géré par CEPAGE.

Enfin, CEPAGE génère aussi un analyseur syntaxique pour chaque éditeur structuré. Cet analyseur ne présuppose pas de contraintes sur la grammaire concrète du langage (i.e. LL, LR, LALR, ... etc). L'algorithme utilisé par CEPAGE peut être appliqué à n'importe quelle grammaire à contexte libre [Earley 1970], bien que pour des raisons d'efficacité on se limite généralement aux grammaires ambiguës finies [Aho 1973].

3. ETUDE COMPARATIVE.

Comme il a été spécifié au début dans l'introduction de ce chapitre, l'étude comparative est centrée sur la partie conception des environnements de programmation. Afin de comparer GEODE au systèmes présentés ci-dessus, nous allons travailler sur un exemple nous permettant d'illustrer les différentes approches. Cet exemple sera celui des expressions arithmétiques parenthésées. Les grammaires utilisées sont les suivantes :

La grammaire concrète C_2 :

1. E → E + T | E - T | T
2. T → T * F | T / F | F
3. F → Const | (E)

Construisons donc la spécification GEODE, et comparons-la aux systèmes présentés précédemment.

Exemple 9. "La définition GEODE (LDG - LDS) de la grammaire engendrant des expressions arithmétiques parenthésées".

La grammaire des expressions arithmétiques parenthésées est décrite par la définition LDG suivante :

GRAMMAR NAME : Expressions ;
START SYMBOL : Axiom ;
GENERIC TERMINALS : Const **WITH LEXICAL FUNCTION** : LexConst ;
SYNONYMS : "+" = Add,
"-" = Sub,
"*" = Mult,
"/" = Div,
"(" = pleft,
")" = pright ;

PRODUCTIONS :

```
Axiom  ::=   E ;
E      ::=   E "+" T | E "-" T | T ;
T      ::=   T "*" F | T "/" F | F ;
F      ::=   Const | "(" E ")" ;
```

END Expressions ;

Le décompilateur de cette grammaire est généré à partir de la définition LDS suivante :

TOOL NAME : Unparser ;
ASSOCIATED GRAMMAR : Expressions ;
ATTRIBUTS :
SYNTHESIZED : rep ;
ASSOCIATED TO : Axiom, E, T, F;
INHERITED : rep ;
ASSOCIATED TO : Const, Add, Sub, Mult, Div ;
INHERITED : str ;
ASSOCIATED TO : Const ;
SEMANTIC EQUATIONS :

```
Axiom  ::=   ? ;
   { Axiom.rep = display ("Axiom") ;                }
Axiom  ::=   E ;
   { Axiom.rep = E.rep ;                            }
E      ::=   ? ;
   { E.rep = display ("Expression") ;               }
E      ::=   E "+" T ;
   { E@1.rep = E@2.rep + Add.rep + T.rep ;
     Add.rep = display (" + ") ;                    }
E      ::=   E "-" T ;
   { E@1.rep = E@2.rep + Sub.rep + T.rep ;
     Sub.rep = display (" - ") ;                    }
E      ::=   T ;
   { E.rep = T.rep ;                                }
```

```
T       ::=     ? ;
    { T.rep = display ("Terme") ;                    }
T       ::=     T "*" F ;
    { T@1.rep = E@2.rep + Mult.rep + F.rep ;
      Mult. rep = display (" * ") ;                  }
T       ::=     T "/" F ;
    { T@1.rep = T@2.rep + Div.rep + F.rep ;
      Div.rep = display (" / ") ;                    }
T       ::=     F ;
    { T.rep = F.rep ;                                }
F       ::=     ? ;
    { F.rep = display ("Factor") ;                   }
F       ::=     "Const" ;
    { F.rep = Const.rep ;
      Const.str = LexConst ;
      Const.rep = display (Const_str) ;              }
F       ::=     "(" E ")" ;
    { F.rep = pleft.rep + E.rep + pright.rep ;
      pleft.rep = display ("(") ;
      pright.rep = display (")") ;                   }
END Unparser ;
```

Le gestionnaire arbre-écran est généré à partir de la définition LDS suivante :

```
TOOL NAME : ScreenManager ;
ASSOCIATED GRAMMAR : Expressions ;
ATTRIBUTS :
        SYNTHESIZED : pos ;
                ASSOCIATED TO : Axiom ;
        INHERITED : pos ;
                ASSOCIATED TO : E, T, F, Const, Add, Sub, Mult, Div ;
SEMANTIC EQUATIONS :
```

```
Axiom  ::=    ?;
   {  Axiom.pos = 0 ;                                          }
Axiom  ::=    E;
   {  Axiom.pos = 0 ;
      E.pos = Axiom.pos ;                                      }
E      ::=    ?;
E      ::=    E "+" T;
   {  E@2.pos = E@1.pos  ;
      Add.pos = E@2.pos + <Unparser>E@2.rep + 1 ;
      T.pos = Add.pos + <Unparser>Add.rep - 1 ;                }
E      ::=    E "-" T;
   {  E@2.pos = E@1.pos  ;
      Sub.pos = E@2.pos + <Unparser>E@2.rep + 1 ;
      T.pos = Sub.pos + <Unparser>Sub.rep - 1 ;                }
E      ::=    T;
   {  T.pos = E.pos ;                                          }
T      ::=    ? ;
T      ::=    T "*" F;
   {  T@2.pos = T@1.pos  ;
      Mult.pos = T@2.pos + <Unparser>T@2.rep + 1 ;
      F.pos = Mult.pos + <Unparser>Mult.rep - 1 ;              }
T      ::=    T "/" F;
   {  T@2.pos = T@1.pos  ;
      Div.pos = T@2.pos + <Unparser>T@2.rep + 1 ;
      F.pos = Div.pos + <Unparser>Div.rep - 1 ;                }
T      ::=    F;
   {  F.pos = T.pos ;                                          }
F      ::=    ? ;
F      ::=    "Const" ;
   {  Const.pos = F.pos ;                                      }
F      ::=    "(" E ")" ;
   {  pleft.pos = F.pos ;
      E.pos = pleft.pos + <Unparser>pleft.rep ;
```

```
            pright.pos = E.pos + <Unparser>E.rep ;                    }
END ScreenManager ;
```

On pourrait aussi penser à la construction d'un outil associant à toute expression arithmétique parenthésée, la valeur issue de son évaluation :

```
TOOL NAME : Evaluation ;
ASSOCIATED GRAMMAR : Expressions ;
ATTRIBUTS :
        SYNTHESIZED : val ;
                ASSOCIATED TO : Axiom, E, T, F;
        INHERITED : val ;
                ASSOCIATED TO : Const ;
SEMANTIC EQUATIONS :
        Axiom  ::=     ? ;
           {  Axiom.val = 0 ;                                          }
        Axiom  ::=     E ;
           {  Axiom.val = E.val ;                                      }
        E      ::=     ? ;
           {  E.val = 0 ;                                              }
        E      ::=     E "+" T ;
           {  E@1.val = E@2.val + T.val ;                              }
        E      ::=     E "-" T ;
           {  E@1.val = E@2.val - T.val ;                              }
        E      ::=     T ;
           {  E.val = T.val ;                                          }
        T      ::=     ? ;
           {  T.val = 0 ;                                              }
        T      ::=     T "*" F ;
           {  T@1.val = T@2.val * F.val ;                              }
        T      ::=     T "/" F ;
           {  T@1.val = T@2.val / F.val ;
              Assert : F.val <> 0 ;                                    }
```

```
T       ::=     F ;
    {T.val = F.val ;                                    }
F       ::=     "Const" ;
    {  F.val = Const.val ;
       Const.val = Value (<Unparser>Const.str) ;        }
F       ::=     "(" E ")" ;
    {  F.val = E.val ;                                  }
END Evaluation ;
```

Une première constatation est que GEODE se veut un générateur d'EPI au même titre que CPS et CENTAUR, et non un générateur d'éditeurs structurés comme CEPAGE et ALOE. Nous nous concentrerons donc sur les similitudes et les différences entre CENTAUR, CPS et GEODE.

La première différence concerne la définition de la grammaire : GEODE utilise uniquement la grammaire concrète avec les inconvénients et les avantages que ceci entraîne. (c.f. Introduction du chapitre VI). Cette définition est écrite en LDG et elle est transformée en un programme Le_Lisp (c.f. chapitre V). Une définition LDG est *grosso modo* une définition BNF complétée par des notations facilitant la définition de listes, de symboles optionnels, etc. Cette définition est indépendante de toute notion d'analyse syntaxique, de décompilation ou d'autres. A différence des systèmes présentés ci-dessus, la définition GEODE de la grammaire concrète ne génère aucun outil de l'environnement de programmation, elle guide uniquement la construction des arbres et des outils. Ceci peut être comparé à la définition de la grammaire abstraite dans les systèmes CENTAUR (SDF) et CPS.

Une fois que la grammaire concrète a été définie, il faut construire les outils de l'environnement. Ces outils doivent être construits en utilisant des équations sémantiques qui seront associées aux différentes règles de cette grammaire (c.f. exemple 9). Il faut insister sur le fait que même des outils comme le décompilateur et le paragrapheur doivent être décrits de cette manière. De même que CENTAUR, CPS et ALOE, GEODE génère le décompilateur et le paragrapheur à partir de la définition de ces outils. CEPAGE, par contre, utilise des outils de décompilation et de paragraphage qui font partie de CEPAGE lui-même. CEPAGE

fournit, bien entendu, des mécanismes permettant d'adapter ces outils aux environnements générés.

Comme nous l'avons montré, tous les outils de l'environnement généré sont décrits en utilisant un seul et unique formalisme : les grammaires attribuées. GEODE se rapproche ici de CPS et s'éloigne de CENTAUR.

Toutefois, CPS n'utilise pas toujours les grammaires attribuées. En effet, comme il a été montré dans l'exemple 4, la décompilation d'un arbre abstrait n'est pas faite par des attributs. En outre, CPS génère **un seul** évaluateur d'attributs. Dans GEODE nous avons utilisé une approche différente : il y a **plusieurs** évaluateurs d'attributs. Chaque outil de l'environnement est associé à un évaluateur. Si l'utilisateur décide de ne pas se servir d'un outil de l'environnement, l'évaluateur associé à cet outil ne sera pas utilisé (i.e. ceci est le concept de sous-environnement de programmation). Il est évident que si l'utilisateur décide de ne pas se servir d'un outil, et que d'autres outils font appel aux attributs de ce dernier, l'évaluation ne pourra être achevée et une condition d'erreur se produira. Aussi, certains outils de l'environnement, notamment ceux représentant l'édition syntaxique, ne pourront pas être supprimés de l'environnement généré. Il est à remarquer que c'est grâce à la définition séparée des outils que la création de sous-environnements est possible. L'intégration des outils ayant été décrits indépendamment les uns des autres, est obtenue en exécutant "simultanément" les différents évaluateurs qui les représentent.

A notre connaissance le concept de sous-environnement de programmation est une particularité de GEODE.

Cependant, le fait de tout représenter par des équations sémantiques implique un certain nombre d'inconvénients. Les grammaires d'attributs étant un mécanisme de bas niveau, il est souvent difficile de décrire les outils et très souvent les outils ainsi créés sont moins souples que les outils générés par d'autres formalismes. Nous étudions actuellement des nouveaux formalismes nous permettant de générer des outils moins rigides. Ceci sera traité dans la conclusion.

De plus, GEODE ne tient absolument pas compte de la sémantique dynamique des

langages de programmation, à la différence de CENTAUR. GEODE est strictement un système dirigé par la syntaxe, bien que certains aspects du ressort de la sémantique des langages puissent être traités en utilisant les équations sémantiques.

CONCLUSION

Conclusion.

Nous avons présenté le système GEODE. GEODE est un générateur d'environnements de programmation intégrés ayant les caractéristiques suivantes :

- Il y a un seul et unique formalisme pour la description du langage de programmation à traiter. Ce formalisme est représenté par une grammaire à contexte-libre et il est implanté par le langage LDG;
- Il y a un seul et unique formalisme pour la description des outils de l'environnement de programmation généré. Ce formalisme est représenté par les grammaires attribuées et il est implanté par le langage LDS;
- Il est possible de définir séparément les outils de l'environnement, tout en ayant des moyens de les réunir ultérieurement. Les définitions LDS des différents outils sont construites indépendamment les une des autres, bien que l'on dispose des moyens de les faire communiquer. Les différents outils sont intégrés en exécutant "simultanément" les évaluateurs qui les représentent;
- Il est possible de créer des sous-environnements de programmation. Si l'utilisateur de l'environnement généré ne souhaite pas se servir d'un outil de cet environnement, il a la possibilité de définir un sous-environnement où l'outil en question ne serait pas utilisé;
- Le concepteur des environnements de programmation dispose des outils de haut niveau lui facilitant sa tâche. Des environnements de programmation pour le traitement des définitions LDG et LDS peuvent en effet être construits par GEODE;
- Les environnements générés sont hautement interactifs. L'interface des environnements générés par GEODE est basée sur l'utilisation exhaustive des menus déroulants et des fenêtres évoluant dans un milieu de haute interactivité.

Nous croyons sincèrement que nous disposons des moyens nous permettant d'atteindre ces objectifs. Cependant, beaucoup de travail reste encore à faire et nous n'avons pas encore pu réaliser un certain nombre de composants de GEODE. Les tables suivantes montrent l'état actuel de GEODE.

Ce qui a été fait : (versions préliminaires) 

- Les constructeurs syntaxique et sémantique;
- Le générateur d'évaluateurs d'attributs;
- Le constructeur incrémental des programmes;
- Un environnement de conception et définition de grammaires à contexte-libre [Canals 1987];
- Une interface utilisateur;
- Parmi les outils générés figurent :
  + Editeur syntaxique;
  + Vérificateur de sémantique statique;
  + Outil de documentation.

Ce qui reste à faire : 

- Implantation de la nouvelle version de GEODE sur SUN;
- Implantation complète de l'algorithme d'évaluation incrémentale;
- Construction d'un environnement de conception d'outils, ainsi que d'un environnement plus complet de définition de grammaires à contexte-libre;
- Une interface utilisateur plus complète sur SUN;
- Un mécanisme d'analyse syntaxique pour l'éditeur;
- Optimisations ...

Deux versions préliminaires de GEODE ont été construites : une sur VAX/785 et l'autre sur Macintosh Plus. Les deux versions ont été programmées en Le_Lisp15.2 [Chailloux 1986] et Ceyx [Hullot 1984]. La troisième version, programmée toujours en Le_Lisp15.2, est en train d'être implantée sur SUN.

En ce qui concerne le futur de GEODE, nous explorons actuellement deux voies :

- Une voie théorique qui pourrait nous conduire vers la définition d'un nouveau formalisme pour la définition des outils;
- Une voie pratique nous permettant de générer des environnements de programmation plus souples et plus agréables à utiliser.

Analysons d'abord la première voie. L'utilisation des grammaires attribuées pour la description des outils dans GEODE est relativement simple et souple. Comme nous l'avons montré, ces grammaires possèdent les propriétés suivantes : intégration simple des outils (à travers la réunion des équations sémantiques), et un aspect déclaratif qui est indépendant de toute notion d'évaluation.

Néanmoins, les grammaires attribuées restent un mécanisme de très bas niveau, très "programmatoire" et elle nécessitent un travail important pour la description des outils. La première idée mise en œuvre dans GEODE pour remédier à ce problème est la construction de l'environnement de conception d'outils. Cependant, si cet environnement peut s'avérer d'un grand secours pour l'écriture et la mise au point des équations sémantiques, il n'est pas vraiment de haut niveau car il il est entièrement basé sur les grammaires attribuées. Notre objectif actuel est d'améliorer ce mécanisme de génération, en proposant un formalisme de plus haut niveau facilitant et réduisant la tâche de description d'outils.

La première étape vers ce formalisme est de constater l'existence d'invariants dans la description d'outils. Par invariant nous entendons l'existence de fonctions sémantiques caractéristiques d'un outil et indépendantes du langage de programmation. Nous pouvons alors proposer, pour chaque composant type d'un environnement, un ensemble de fonctions caractéristiques prédéfinies. La conception d'un outil pour un langage particulier revient alors à rassembler ces différentes fonctions de manière cohérente avec ce langage. Pourtant, si l'on garde le formalisme de grammaires attribuées, cet ensemble peut s'avérer complexe est fastidieux. C'est pourquoi, la deuxième idée, que nous explorons actuellement, consiste à définir un nouveau formalisme permettant d'associer des fonctions et des équations sémantiques non plus à des productions de la grammaire, mais à des schémas d'arbres. Le langage de description d'outils que nous obtenons a alors la forme d'un ensemble de règles :

```
Schéma      →     Fonctions  -  Equations Sémantiques
```

L'écriture des fonctions et des équations sémantiques se fait en utilisant les invariants d'outils dans un langage fortement modulaire favorisant la réutilisabilité, de manière à pouvoir combiner simplement les différentes fonctions déjà écrites.

L'avantage d'un tel formalisme par rapport aux grammaires attribuées, seraient les suivants :

- Circulation facilité de l'information dans l'arbre : En effet, il est possible de définir des transferts d'information entre deux nœuds n'appartenant pas à la même production;
- Meilleure prise en compte de la structure d'un programme : Le flux d'information dans l'arbre représente plus fidèlement les différentes parties du programme (i.e. déclarations, instructions, ...).

Il faut aussi souligner le fait que ce formalisme constitue un langage de description d'outils dirigés par la syntaxe, dans le sens où ces outils utilisent une représentation syntaxique du programme, mais à aucun moment ils n'utilisent ni manipulent une quelconque description de la sémantique du langage.

Nous souhaitons, de plus, avoir un mécanisme nous permettant, d'un côté, de décrire simplement un langage, et d'un autre côté de générer des arbres dont la taille serait beaucoup moins importante que celle des arbres de dérivation associés à la grammaire à contexte-libre. Le choix évident pour cela est la grammaire abstraite. Mais, comment faire en sorte que l'on ait pas à travailler avec deux formalismes (i.e. grammaires concrète et abstraite) de description de langages ? La réponse est d'avoir un formalisme nous permettant de définir les deux grammaires au même temps, comme dans CEPAGE et CENTAUR (SDF), ou bien, d'essayer de générer, à partir de la grammaire concrète, une grammaire abstraite. Dans GEODE, nous voudrions adopter la deuxième approche. Bien entendu, les productions abstraites auxquelles seraient associées les équations sémantiques, devraient être automatiquement générées à partir de la description faite dans ce nouveau formalisme.

En ce qui concerne les aspects pratiques que nous voulons implanter sur GEODE,

nous sommes partis des constations suivantes :

- Les éditeurs syntaxiques générés par GEODE sont assez rigides, dans le sens où un utilisateur éventuel devrait respecter, sans pouvoir la modifier, l'indentation établie à travers les équations sémantiques du gestionnaire arbre-écran;
- La taille des arbres de dérivation était trop importante (c.f. proposition du nouveau formalisme ci-dessus).

La figure C.1 montre les modifications de l'architecture de GEODE†, qui pourrait nous conduire vers des environnements de programmation plus efficaces et plus souples. Les principales modifications sont les suivantes :

- Tout en conservant un seul formalisme pour la description d'un langage (i.e. une grammaire à contexte-libre), GEODE générerait une grammaire abstraite à laquelle on associerait les équations sémantiques des outils de l'environnement à générer;
- Le gestionnaire arbre-écran ne serait plus généré par des équations sémantiques. Cet outil ferait donc partie des outils GEODE au même titre que les constructeurs syntaxique et sémantique, par exemple. Il est à remarquer que le décompilateur serait toujours généré par des équations sémantiques.

Les caractéristiques d'un tel système serait les suivantes :

- Edition syntaxique souple et efficace;
- Calcul automatique de la syntaxe abstraite;
- Les équations sémantiques sont définies sur les productions abstraites;
- Toutes les autres caractéristiques de GEODE restent les mêmes.

---

† **NOTA :** Un premier prototype de ce nouveau système, appelé GEPI (**G**énérateur d'**E**nvironnements de **P**rogrammation **I**ntégrés est en cours d'implantation sur SUN [Canals 1988b].

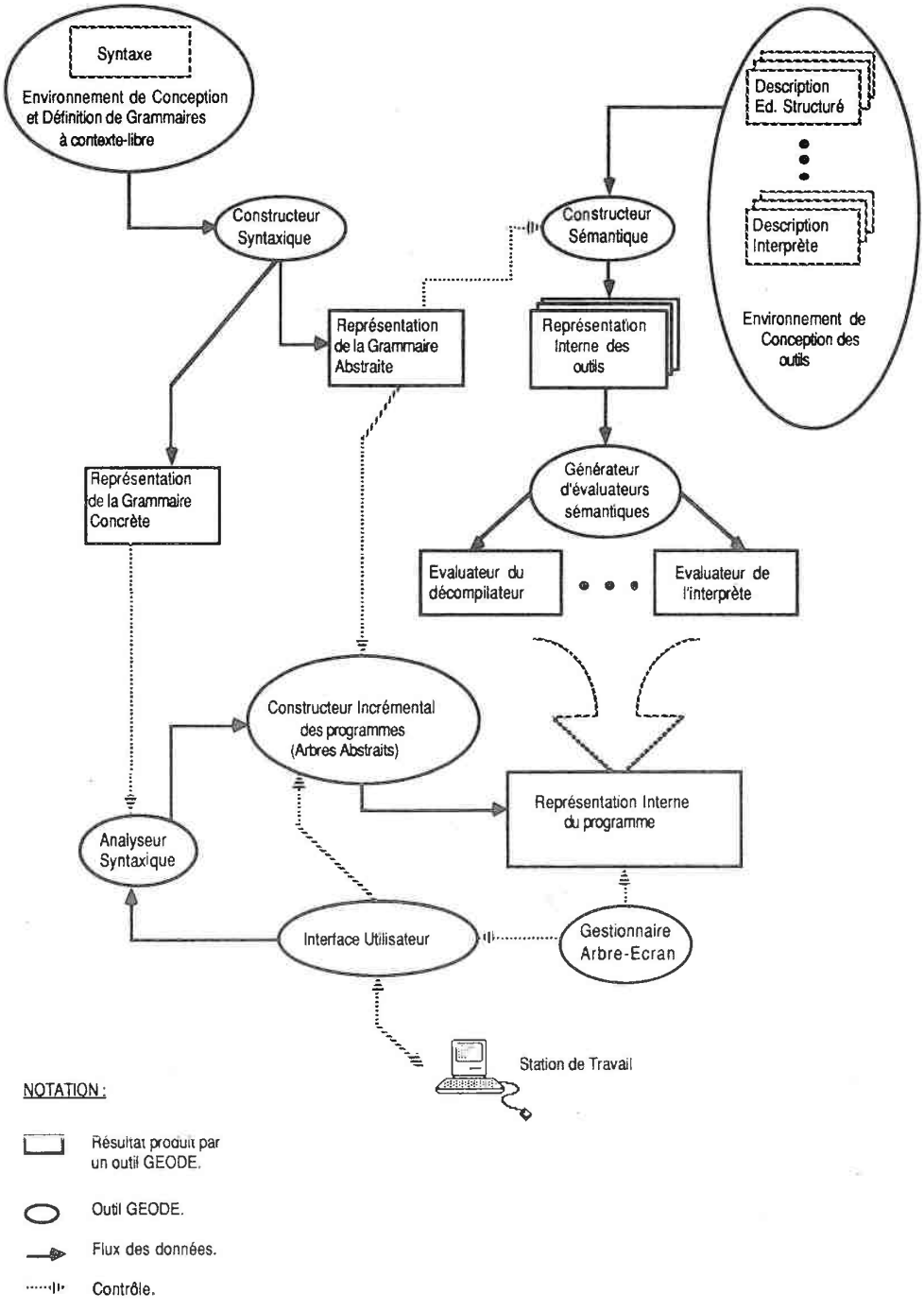


Figure C.1 "La nouvelle architecture du système GEODE".

BIBLIOGRAPHIE

VII. BIBLIOGRAPHIE.

[Aho 1973]
Aho A.V. and Ullman J.E.
"The Theory of Parsing, Translation and Compiling".
Volume I: "Parsing".
Prentice-Hall, Englewood Cliffs, N.J. 1973.

[Aho 1977]
Aho A.V. and Ullman J.E.
"Principles of Compiler Design".
Addison-Wesley 1977.

[ALF 1987]
ESPRIT Project 1520.
Advanced Software Engineering Environment Logistics Framework.
Technical Annex.
June 1987.

[Althoff 1981]
Althoff J.C., Deutsch L.P., Ingalls D.H., Galls D.H., Goldberg A., Kaehler T., Krasner G., Reenskaug M.H., Robson D., Ross J. and Tesler L.
"The Smalltalk-80 System".
Byte, Vol. 6, No. 8.
1981.

[André 1986]
André E., Moreau B. et Rougeot B.
"Vers un atelier flexible et intégré du logiciel : le projet CONCERTO".
Présentation Technique.
Perros-Guirec 1986.

[Arthur 1981]
Arthur J. and Ramanathan J.
"Design of analysers for selective program analysis".
IEEE Trans. Softw. Eng. SE-7, 39-51.
January 1981.

[Attard 1986]
Attard R. et Rault J.C.
"PNGL : Le Projet National de Génie Logiciel".
Révue de Génie Logiciel. No. 2. (Agence de l'informatique).
Novembre 1986.

[Barstow 1984]
Barstow D.R., Shrobe H.E. and Sandewall E.
"Interactive Programming Environments".
McGraw-Hill Book Company.
1984.

[Basili 1982]
Basili V.R. and Mills H.D.
"Understanding and Documenting Programs".
IEEE Trans. Soft. Eng. SE-8, No. 3, 270-283.
May 1982.

[Bell 1987]
Bell D., Morrey I. and Pugh J.
"Software Enginnering : A Programming Approach".
Prentice-Hall International.
1987.

[Bergstra 1986]
Bergstra J.A., Heering J. and Klint P.
"ASF - An Algebraic Specification Formalism"".
Annexe D7.A1 of Deliverable D7. Fourth Review.
GIPE : 348/A/T7/8
December 1986.

[Bochman 1976]
Bochman G.V.
"Semantic Evaluation from Left to Right".
Communications of the ACM. Vol. 19, No. 2.
February 1976.

[Bourguignon 1986]
Bourguignon J.P.
"La structure d'accueil EMERAUDE".
Révue de Génie Logiciel. No. 2. (Agence de l'informatique).
Novembre 1986.

[Canals 1987]
Canals G.
"GEPOL : L'éditeur syntaxique et l'environnement de conception".
Rapport DEA Informatique.
Université de Nancy I. (CRIN)
Septembre 1987.

[Canals 1988]
Canals G., Cruzlara Silva S. et Derniame J.C.
"GEODE : Un système pour la génération d'environnements de programmation intégrés".
Proc. Premier Séminaire International sur le génie logiciel.
10 - 12 octobre 1988. Oran, ALGERIE.

[Canals 1988b]
Canals G., Colnet D., Cruzlara Silva S. et Derniame J.C.
"GEPI : Un générateur d'environnements de programmation intégrés".
Proc. "Le Génie Logiciel et ses Applications".
Journées Internationales. Toulouse, 5 - 9 décembre 1988.

[Chailloux 1986]
Chailloux J., Devin M., Dupont F., Hullot J.M., Serpette B. et Vuillemin J.
"Le_Lisp de l'INRIA : Le manuel de référence".
Version 15.2.
Novembre 1986.

[Clément 1985]
Clément D., Despeyroux J., Despeyroux T., Hascoet L. and Kahn G.
"Natural Semantics on the Computer".
INRIA Sophia-Antipolis.
June 1985.

[Clément 1986]
Clément D., Heering J., Incerpi J., Kahn G., Klint P., Lang B. and Pascual V.
"Preliminary Design of an Environment Generator".
Deliverable D9. Fourth Review.
GIPE : CEC 348/A/T9/1.
December 23, 1986.

[Colnet 1987]
Colnet D., Masini G., Napoli A., Noiret Y. et Tombre K.
"Les langages orientés objets".
Rapport CRIN 86-R-077.
1987.

[Courcelle 1984]
Courcelle B.
"Attribute Grammars : Definitions, Analysis of Dependencies, Proof Methods".
GRECO de programmation.
July 1984.

[Cox 1984]
Cox B.J.
"Object-Oriented Programming : An Evolutionary Approach".
Addison-Wesley.
1984.

[Cruzlara 1986]
Cruzlara Silva S. et Derniame J.C.
"Une approche pour la construction d'environnements de programmation orientés vers un langage de programmation particulier".
Proc. JISI 1986, Tunis Tunisie.
Avril 1986.

[Cruzlara 1987]
Cruzlara Silva S. and Derniame J.C.
"Yet Another Programming Environment Generator Based on Attribute Grammars".
CRIN Report No. 87-R-079.
1987.

[Dahl 1966]
Dahl O.J. and Nygaard K.
"SIMULA : An ALGOL-based simulation language".
Communications of the ACM. Vol. 9, No. 9, pp. 671-678.
1966.

[Deransart 1985]
Deransart P., Jourdan M. and Lorho B.
"A Survey on Attribute Grammars".
Part III. Classified Bibliography.
Rapport de recherche No. 417.
INRIA Rocquencourt. Juin 1985.

[Donzeau-Gouge 1975]
Donzeau-Gouge V., Huet G., Kahn G., Lang B. and Lévy J.J.
"A Structured Oriented Program Editor: A first step towards computer assisted programming".
INRIA Rocquencourt.
Rapport de recherche No. 114. Avril 1975.

[Donzeau-Gouge 1980]
Donzeau-Gouge V., Huet G., Kahn G. and Lang B.
"Programming Environments based on Structured Editors : The MENTOR Experience".
INRIA Rocquencourt.
Rapport de recherche No. 26. Juillet 1980.

[Earley 1970]
Earley J.
"An Efficient Context-Free Parsing Algorithm".
Communications of the ACM. Vol. 13. No 2. pp. 94-102.
February 1970.

[Farrow 1982]
Farrow R.
"Yet Another Translator Writing System Based on Attribute Grammars".
Intel Corporation.
Proc. of the SIGPLAN 82 Symposium on Compiler Construction.
Boston, Mass. June 23-25, 1982.

[Habermann 1979]
Habermann A. Nico.
"The GANDALF research project".
In Computer Science Research Review 1978-79.
Carnegie-Mellon University.
1979.

[Hansen 1971]
Hansen W.
"Creation of Hierarchic Text with a Computer Display".
Ph.D dissertation. Departement of Computer Science.
Stanford University, California. June 1971.

[Heering 1986]
Heering J. and Klint P.
"A Syntax Definition Formalism".
Annexe D7.A4 of Deliverable D7. Fourth Review.
GIPE : CEC 348/A/T7/6
November 1986.

[Hullot 1983]
Hullot J.M.
"CEYX : Un environnement de programmation multiformalisme".
Premières Journées d'Etude sur les Langages Orientés Objets.
Le Cap d'Adge, BIGRE 37. 1983.

[Johnson 1979]
Johnson S.C.
"YACC : Yet Another Compiler - Compiler".
UNIX Programmer's Manual. Vol. 2B.
Bell Laboratories. 1979.

[Jourdan 1984]
Jourdan M.
"An optimal-time recursive evaluator for attribute grammars".
INRIA Rocquencourt.
Décembre 1984.

[Kahn 1983]
Kahn G., Lang B., Mélèse B. and Morcos E.
"METAL : A formalism to specify formalisms".
Science of Computer Programming, 3. pp. 151 - 188.
1983.

[Kahn 1986]
Kahn G.
"Natural Semantics".
Deliverable D7.A7 of Task T7.
GIPE : CEC 348/S/T7/4.
October 1986.

[Kant 1981]
Kant E. and Barstow R.D.
"The Refinement Paradigm : The Interaction of Coding and Efficiency Knowledge in Program Synthesis".
IEEE Transactions on Software Engineering, SE-7:5.
September 1981.

[Kastens 1980]
Kastens U.
"Ordered Attribute Grammars".
Acta Informatica, Vol. 13, No. 3, 229-256.
1980.

[Kennedy 1976]
Kennedy K. and Warren S.K.
"Automatic generation of efficient evaluators for attribute grammars".
In Conference Record of the 3rd ACM Symposium on Principles of Programming Languages. Atlanta, Ga. pp. 32-49.
January 1976.

[Kernighan 1981]
Kernighan B.W. and Mashey J.R.
"The UNIX Programming Environment".
Computer, 14:4. pp. 25-34.
April 1981.

[Knuth 1968]
Knuth D.E.
"Semantics of context-free languages".
Math. Syst. Theory Vol. 2, No. 2, 127-145.
June 1968.

[Knuth 1968b]
Knuth D.E.
"The Art of Computer Programming, vol1: Fundamental Algorithms".
Addison-Wesley, Reading Mass. USA.
1968.

[Lesk 1979]
Lesk M.E. and Schmidt E.
"LEX : A lexical analyzer generator".
UNIX Programmer's Manual. Vol 2B.
Bell Laboratories. 1979.

[Lewis 1974]
Lewis P.M., Rosenkratz D.J. and Stearns R.E.
"Attributed Translations".
J. Comput. Syst. Sci. Vol. 9, No. 3, 279-307.
December 1974.

[Livercy 1978]
J.P. Finance, M. Grandbastien, P. Lescanne, P. Marchand, R. Mohr, A. Quéré et J.L. Rémy.
"Théorie des programmes".
Dunod Informatique. Paris 1978.

[LNCS 1980]
Lectures Notes in Computer Science.
"Semantics-Directed Compiler Generation".
Proc. of a Workshop.
Aarhus, Denmark. January 1980.
Springer-Verlag.

[Lorho 1977]
Lorho B.
"Semantic attributes processing in the system DELTA".
Methods of Algorithmic Language Implementation.
A. Ershov, C.H.A. Koster Eds.
LNCS 47, Springer-Verlag 1977.

[Medina-Mora 1981]
Medina-Mora R. and Notkin D.S.
"ALOE User's and Implementors' Guide".
Department of Computer Science. Carnegie-Mellon University.
Pittsburgh, Pa. USA.
November 1981.

[Medina-Mora 1982]
Medina-Mora R.
"Syntax-directed editing: Towards integrated programming environments".
Ph.D dissertation, Departement of Computer Science, Carnegie-Mellon University.
Pittsburgh, Pa. March 1982.

[Meyer 1979]
Meyer B.
"Les concepts de SIMULA-67".
Atelier Logiciel EDF. 1979.

[Meyer 1987]
Meyer B.
"Eiffel : A Language and Environment for Software Engineering".
The Journal of Systems and Software.
1987.

[Meyer 1987b]
Meyer B.
"CEPAGE : Towards Computer-Aided Design of Software".
Interactive Software Engineering, Inc.
TR-CE-7/GP. Version 2.
March 1987.

[Meyer 1987c]
Meyer B.
"LDL : A Language Description Language".
Interactive Software Engineering, Inc.
TR-CE-8/LD. Version 2.2.
Mai 1987.

[Mills 1970]
Mills H.D.
"Syntax-directed documentation for PL-360".
Communications of the ACM, Vol. 13, No. 4.
April 1970.

[Moon 1974]
Moon D.
"MACLISP Reference Manual".
Massachusetts Institute of Technology, Projet MAC.
1974.

[Néel 1974]
Néel D., Amirchahy M. and Mazaud M.
"Optimization of Generated Code by means of Attributes: Local Elimination of Common Redundant Sub-expressions".
GI 4. Jahrestagung, Berlin Ed.
Siefkes LNCS 26, Springer-Verlag 1974.

[Pair 1976]
Pair C., Amirchahy M. et D. Néel.
"Preuves pour des descriptions de traitements de textes par attributs".
Rapport de recherche 163.
IRIA-Laboria. Rocquencourt 1976.

[Rekers 1986]
Rekers J.
"A Parser Generator for Finitely Ambigous Context-Free Grammars".
Annexe D8.A1 of Deliverable D8. Fourth Review.
GIPE : CEC 348/A/T8/1.
December 1986.

[Reps 1981]
Reps T.
"The Synthesizer Editor Generator: Reference Manual".
Dept. of Computer Science, Cornell University.
Ithaca, N.Y. September 1981.

[Reps 1983]
Reps T.
"Generating Language-Based Environments".
ACM Doctoral Dissertation Award 1983.
The MIT Press. 1983.

[Reps 1983b]
Reps T., Teitelbaum T. and Demers A.
"Incremental Context-Dependent Analysis for Language-Based Editors".
ACM Transactions on Programming Languages and Systems.
Vol. 5, No. 3, July 1983. pp. 449-477

[Reps 1984]
Reps T. and Teitelbaum T.
"The Synthesizer Generator".
In Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments.
Pittsburgh. P.A. USA. April 1984.

[Reps 1985]
Reps T. and Teitelbaum T.
"The Synthesizer Generator Reference Manual".
Departement of Computer Science.
Cornell University. Ithaca, N.Y. USA.
August 1985.

[Teitelbaum 1981]
Teitelbaum T. and Reps T.
"The Cornell Program Synthesizer : A syntax-directed programming environment".
Communications of the ACM. 24,9. pp. 563 - 573.
September 1981.

[Teitelman 1981]
Teitelman W. and Masinter L.
"The INTERLISP Programming Environment".
IEEE Computer, 14:4.
April 1981.

[Waters 1982]
Waters, R.C.
"The Programmer's Apprentice : Knowledge Based Program Editing".
IEEE Transactions on Software Engineering, SE-8:1.
January 1982.

[Winograd 1973]
Winograd T.
"Breaking the Complexity Barrier (Again)".
ACM SIGPLAN-SIGIR Interface Meeting on Programming Languages-Information Retrieval.
November 1973.

[Wirth 1971]
Wirth N.
"The Programming Language PASCAL".
Acta Informatica, Vol. 1, No. 1, 35-63. 1971.

[Wirth 1976]
Wirth N.
"Algorithms + Data Structures = Programs".
Prentice-Hall, Englewood Cliffs, N.J. 1976.

ANNEXES

Annexe 1.

LA GRAMMAIRE LDG.

1) Axiom → GrammarDefinition

2) GrammarDefinition → GrammarName
StartSymbol
GenericTerminals
Synonyms
Productions
"End" GrammarIdentifier ";"

3) GrammarName → "Grammar Name" ":" GrammarIdentifier ";"

4) StartSymbol → "Start Symbol" ":" NonterminalIdentifier ";"

5) GenericTerminals → "Generic Terminals" ":" GenericTerminalList | "empty"

6) GenericTerminalList → LexicalDeclaration GenericTerminalList |
LexicalDeclaration

7) LexicalDeclaration → GenericTerminalIdentifier "With Lexical Function" ":"
LexicalFunction ";"

8) Synonyms → "Synonyms" ":" SynonymList ";" | "empty"

9) SynonymList → SynonymDeclaration "," SynonymList |
SynonymDeclaration

10) SynonymDeclaration → TerminalIdentifier "=" SynonymIdentifier

11) Productions → "Productions" ":" ProductionList

12) ProductionList → ProductionDeclaration ProductionList |
ProductionDeclaration

13) ProductionDeclaration → NonterminalIdentifier "::=" ProductionRightSide ";"

14) ProductionRightSide → SymbolList "|" ProductionRightSide |
SymbolList

15) SymbolList → TerminalIdentifier |
GenericTerminalIdentifier |
NonterminalIdentifier |
TerminalIdentifier SymbolList |
NonterminalIdentifier SymbolList

Annexe 2.

LA GRAMMAIRE LDS.

1)	Axiom	→	ToolDefinition
2)	ToolDefinition	→	ToolName AssociatedGrammar Attributs SemanticEquations "End" ToolIdentifier
3)	ToolName	→	"Tool Name" ":" ToolIdentifier ";"
4)	AssociatedGrammar	→	"Associated Grammar" ":" GrammarIdentifier ";"
5)	Attributs	→	"Attributs" ":" AttributDeclaration
6)	AttributDeclaration	→	SynthesizedDeclaration | InheritedDeclaration
7)	SynthesizedDeclaration	→	SynthesizedDecl SynthesizedDeclaration | SynthesizedDecl
8)	SynthesizedDecl	→	"Synthesized" ":" AttributList ";" "Associated To" ":" SyntaxList ";" | "empty"
9)	AttributList	→	AttributIdentifier "," AttributList | AttributIdentifier
10)	SyntaxList	→	SyntaxSymbol "," SyntaxList | SyntaxSymbol
11)	SyntaxSymbol	→	NonterminalIdentifier | TerminalIdentifier
12)	InheritedDeclaration	→	InheritedDecl InheritedDeclaration | InheritedDecl
13)	InheritedDecl	→	"Inherited" ":" AttributList ";" "Associated To" ":" SyntaxList ";" | "empty"
14)	SemanticEquations	→	"Semantic Equations" ":" ProductionEquationList
15)	ProductionEquationList	→	Production EquationDecl ProductionEquationList | Production EquationDecl
16)	Production	→	NonterminalIdentifier "::=" ProductionRightSide ";"
17)	ProductionRightSide	→	NullDerivation | NotNullDerivation
18)	NullDerivation	→	"?"
19)	NotNullDerivation	→	SymbolList "|" NotNullDerivation | SymbolList
20)	SymbolList	→	TerminalIdentifier | NonterminalIdentifier | TerminalIdentifier SymbolList |

NonterminalIdentifier SymbolList

21) EquationDecl → "{" EquationList "}" | "empty"

22) EquationList → Equation EquationList | Equation

23) Equation → AttributeInstance "=" EquationRightSide ";" | Assertion ";"

24) EquationRightSide → Expression | FonctionCall

25) Expression → Term "+" Expression | Term "-" Expression | Term UnaryOperator Term

26) Term → Factor "*" Term | Factor "/" Term | Factor

27) Factor → AttributInstance | "(" AttributInstance ")" | NumericConstant

28) FunctionCall → FunctionIdentifier | FunctionIdentifier "(" ParameterList ")"

29) ParameterList → Parameter "," ParameterList | Parameter

30) Parameter → AttributeInstance | StringConstant | NumericConstant

31) Assertion → "Assert" ":" BoolExpression

32) BoolExpression → BoolTerm "Or" BoolExpression | BoolTerm

33) BoolTerm → BoolFactor "And" BoolTerm | BoolFactor

34) BoolFactor → AttributOrConstant RelOperator AttributOrConst | BoolExpression RelOperator BoolExpression | FunctionCall

35) AttributOrConstant → AttributInstance | NumericConstant

36) RelOperator → "=" | "<>" | "<" | "<=" | ">" | ">="

Annexe 3.

LA GRAMMAIRE CONCRETE DU LANGAGE PL0.

1)	start	→	¥
2)	start	→	program
3)	program	→	¥
4)	program	→	block "."
5)	block	→	¥
6)	block	→	decl_const decl_var decl_proc statement
7)	decl_const	→	¥
8)	decl_const	→	"CONST" decl_const_list ";"
9)	decl_const	→	"empty"
10)	decl_const_list	→	¥
11)	decl_const_list	→	"ident" "=" "number"
12)	decl_const_list	→	"ident" "=" "number" "," decl_const_list
13)	decl_var	→	¥
14)	decl_var	→	"VAR" decl_var_list ";"
15)	decl_var	→	"empty"
16)	decl_var_list	→	¥

17)	decl_var_list	→	"ident"
18)	decl_var_list	→	"ident" "," decl_var_list
19)	decl_proc	→	¥
20)	decl_proc	→	proc_list ";" decl_proc
21)	decl_proc	→	"empty"
22)	proc_list	→	¥
23)	proc_list	→	"PROCEDURE" decl_proc_list
24)	decl_proc_list	→	¥
25)	decl_proc_list	→	"ident" ";" block
26)	statement	→	¥
27)	statement	→	"ident" ":=" expression
28)	statement	→	"CALL" ident
29)	statement	→	"BEGIN" statement_list "END"
30)	statement	→	"IF" condition "THEN" statement
31)	statement	→	"WHILE" condition "DO" statement
32)	statement	→	"empty"
33)	statement_list	→	¥
34)	statement_list	→	statement
35)	statement_list	→	statement ";" statement_list
36)	condition	→	¥

37)	condition	→	"ODD" expression
38)	condition	→	expression cond_operators expression
39)	cond_operators	→	¥
40)	cond_operators	→	"="
41)	cond_operators	→	"<>"
42)	cond_operators	→	"<"
43)	cond_operators	→	">"
44)	cond_operators	→	"<="
45)	cond_operators	→	">="
46)	expression	→	¥
47)	expression	→	plus_minus_empty term_list
48)	plus_minus_empty	→	¥
49)	plus_minus_empty	→	"+"
50)	plus_minus_empty	→	"-"
51)	plus_minus_empty	→	"empty"
52)	term_list	→	¥
53)	term_list	→	term
54)	term_list	→	term add_or_sust term_list
55)	add_or_sust	→	¥
56)	add_or_sust	→	"+"

57)	add_or_sust	→	"-"
58)	term	→	¥
59)	term	→	factor_list
60)	factor_list	→	¥
61)	factor_list	→	factor
62)	factor_list	→	factor mult_or_div factor_list
63)	factor	→	¥
64)	factor	→	"ident"
65)	factor	→	"number"
66)	factor	→	"(" expression ")"
67)	mult_or_div	→	¥
68)	mult_or_div	→	"*"
69)	mult_or_div	→	"/"

Annexe 4.

LA GRAMMAIRE ABSTRAITE DU LANGAGE PL0.

Program	:	block → program
Block	:	decl_const × decl_var × decl_proc × statement → block
DeclConst	:	{ident × number}* → decl_const
DeclVar	:	{ident}* → decl_var
DeclProc	:	ident × block → decl_proc
AssignStat	:	ident × expression → statement
CallStat	:	ident → statement
CompoundStat	:	{statement}+ → statement
IfStat	:	condition × statement → statement
WhileStat	:	condition × statement → statement
OddCond	:	expression → condition
Equal	:	expression × expression → condition
NotEqual	:	expression × expression → condition
Less	:	expression × expression → condition
Greater	:	expression × expression → condition
LessEqual	:	expression × expression → condition
GreaterEqual	:	expression × expression → condition
BracketExp	:	→ expression
Plus	:	→ expression
Minus	:	→ expression
Add	:	expression × expression→ expression
Subtract	:	expression × expression→ expression
Multiply	:	expression × expression → expression
Divide	:	expression × expression → expression

ident, number ⊆ expression 1)

Annexe 5.

LE DECOMPILATEUR DU LANGAGE PL0.

```
1)   start          →   ¥
         start.rep = display ("START");

2)   start          →   program
         start.rep = program.rep;
         program.tab = 0;

3)   program        →   ¥
         program.rep = display (program.tab, "PROGRAM");

4)   program        →   block "."
         program.rep = block.tab + block.rep + display (cr) + point.rep;
         point.rep = display (cr);
         block.tab = program.tab;

5)   block          →   ¥
         block.rep = display  (block.tab, "BLOCK");

6)   block          →   decl_const decl_var decl_proc statement
         block.rep = decl_const.tab + decl_const.rep + display (cr) +
                     decl_var.tab + decl_var.rep + display (cr) +
                     decl_proc.tab + decl_proc.rep + display (cr) +
                     statement.tab + statement.rep;
         decl_const.tab = block.tab;
         decl_var.tab = block.tab;
         decl_proc.tab = block.tab;
         statement.tab = block.tab;

7)   decl_const     →   ¥
         decl_const.rep = display (decl_const.tab, "DECL_CONST");

8)   decl_const     →   "CONST" decl_const_list ";"
         decl_const.rep =  decl_const.tab + CONST.rep + display (cr) +
                           decl_const_list.rep + decl_const_list.tab +
                           semicolon.rep;
         CONST.rep = display (decl_const.tab, "CONST");
         semicolon.rep = display (";");
```

```
        decl_const_list.tab = 1 + decl_const.tab;

9)      decl_const          →    "empty"
        decl_const.rep = 0;

10)     decl_const_list     →    ¥
        decl_const_list.rep = display (decl_const_list.tab, "DECL_CONST_LIST");

11)     decl_const_list     →    "ident" "=" "number"
        decl_const_list.rep = decl_const_list.tab + ident.rep + equal.rep + number.rep;
        ident.rep = display (decl_const_list.tab, ident.str);
        ident.str = fonction_id;
        equal.rep = display ("=");
        number.rep = display (number.str);
        number.str = fonction_num;

12)     decl_const_list     →    "ident" "=" "number" "," decl_const_list
        decl_const_list@1.rep = decl_const_list@1.tab + ident.rep + equal.rep +
                                number.rep + comma.rep + display (cr) +
                                decl_const_list@2.tab + decl_const_list@2.rep;
        decl_const_list@2.tab = decl_const_list@1.tab;
        ident.rep = display (decl_const_list@1.tab, ident.str);
        ident.str = fonction_id;
        equal.rep = display ("=");
        number.rep = display (number.str);
        number.str = fonction_num;
        comma.rep = display (",");

13)     decl_var            →    ¥
        decl_var.rep = display (decl_var.tab, "DECL_VAR");

14)     decl_var            →    "VAR" decl_var_list ";"
        decl_var.rep = decl_var.tab + VAR.rep + display (cr) + decl_var_list.rep +
                       decl_var_list.tab + semicolon.rep;
        VAR.rep = display (decl_var.tab, "VAR");
        semicolon.rep = display (";");
        decl_var_list.tab = 1 + decl_var.tab;

15)     decl_var            →    "empty"
        decl_var.rep = 0;

16)     decl_var_list       →    ¥
        decl_var_list.rep = display (decl_var_list.tab, "DECL_VAR_LIST");

17)     decl_var_list       →    "ident"
        decl_var_list.rep = ident.rep + decl_var_list.tab;
```

```
        ident.rep = display (decl_var_list.tab, ident.str);
        ident.str = fonction_id;

18)  decl_var_list      →   "ident"  ","  decl_var_list
        decl_var_list@1.rep =  decl_var_list@1.tab + ident.rep + comma.rep +
                               display (cr) + decl_var_list@2.tab +
                               decl_var_list@2.rep;
        decl_var_list@2.tab = decl_var_list@1.tab;
        ident.rep = display (decl_var_list@1.tab, ident.str);
        ident.str = fonction_id;
        comma.rep = display (",");

19)  decl_proc          →   ¥
        decl_proc.rep = display (decl_proc.tab, "DECL_PROC");

20)  decl_proc          →   proc_list  ";"  decl_proc
        decl_proc@1.rep =  proc_list.tab + proc_list.rep + semicolon.rep +
                           display (cr) + decl_proc@2.tab + decl_proc@2.rep;
        semicolon.rep = display (";");
        proc_list.tab = decl_proc@1.tab;
        decl_proc@2.tab = 1 + decl_proc@1.tab;

21)  decl_proc          →   "empty"
        decl_proc.rep = 0;

22)  proc_list          →   ¥
        proc_list.rep = display (proc_list.tab, "PROC_LIST");

23)  proc_list          →   "PROCEDURE"  decl_proc_list
        proc_list.rep = proc_list.tab + PROCEDURE.rep + decl_proc_list.rep;
        PROCEDURE.rep = display (proc_list.tab, "PROCEDURE")
        decl_proc_list.tab = proc_list.tab;

24)  decl_proc_list     →   ¥
        decl_proc_list.rep = display (decl_proc_list.tab, "DECL_PROC_LIST");

25)  decl_proc_list     →   "ident"  ";"  block
        decl_proc_list.rep = decl_proc_list.tab + ident.rep + semicolon.rep +
                             display (cr) + block.tab + block.rep;
        ident.rep = display (decl_proc_list.tab, ident.str);
        ident.str = fonction_id;
        semicolon.rep = display (";");
        block.tab = 1 + decl_proc_list.tab;
26)  statement          →   ¥
        statement.rep = display (statement.tab, "STATEMENT");
```

```
27)   statement        →   "ident" ":=" expression
          statement.rep = statement.tab + ident.rep + assign.rep + expression.rep;
          assign.rep = display (":=");
          ident.rep = display (statement.tab, ident.str);
          ident.str = fonction_id;

28)   statement        →   "CALL" ident
          statement.rep = statement.tab + CALL.rep + ident.rep;
          CALL.rep = display (statement.tab, "CALL");
          ident.rep = display (ident.str);
          ident.str = fonction_id;

29)   statement        →   "BEGIN" statement_list "END"
          statement.rep =  statement.tab + BEGIN.rep + display (cr) +
                           statement_list.tab + statement_list.rep + display (cr) +
                           statement.tab + END.rep;
          BEGIN.rep = display (statement.tab, "BEGIN");
          END.rep = display (statement.tab, "END");
          statement_list.tab = 1 + statement.tab;

30)   statement        →   "IF" condition "THEN" statement
          statement@1.rep = statement@1.tab + IF.rep + condition.rep + THEN.rep +
                            display (cr) + statement@2.tab + statement@2.rep;
          IF.rep = display (statement@1.tab, "IF");
          THEN.rep = display ("THEN");
          statement@2.tab = 1 + statement@1.tab;

31)   statement        →   "WHILE" condition "DO" statement
          statement@1.rep = statement@1.tab + WHILE.rep + condition.rep + DO.rep +
                            display (cr) + statement@2.tab + statement@2.rep;
          WHILE.rep = display (statement@1.tab, "WHILE");
          DO.rep = display ("DO");
          statement@2.tab = 1 + statement@1.tab;

32)   statement        →   "empty"
          statement.rep = 0;

33)   statement_list   →   ¥
          statement_list.rep = display (statement_list.tab, "STATEMENT_LIST");

34)   statement_list   →   statement
          statement_list.rep = statement.tab + statement.rep;
          statement.tab = statement_list.tab;

35)   statement_list   →   statement ";" statement_list
          statement_list@1.rep =  statement.tab + statement.rep +  semicolon.rep +
```

```
                                    display (cr) + statement_list@2.tab +
                                    statement_list@2.rep;
        semicolon.rep = display (";");
        statement.tab = statement_list@1.tab;
        statement_list@2.tab = statement_list@1.tab;

36)  condition          →   ¥
        condition.rep = display ("CONDITION");

37)  condition          →   "ODD" expression
        condition.rep = condition.tab + ODD.rep + expression.rep;
        ODD.rep = display ("ODD");

38)  condition          →   expression cond_operators expression
        condition.rep =  expression@1.tab + expression@1.rep +
                         cond_operators.rep + expression@2.rep;

39)  cond_operators     →   ¥
        cond_operators.rep = display ("COND_OPERATORS");

40)  cond_operators     →   "="
        cond_operators.rep = equal.rep;
        equal.rep = display ("=");

41)  cond_operators     →   "<>"
        cond_operators.rep = not_equal.rep;
        not_equal.rep = display ("<>");

42)  cond_operators     →   "<"
        cond_operators.rep = less.rep;
        less.rep = display ("<");

43)  cond_operators     →   ">"
        cond_operators.rep = greater.rep;
        greater.rep = display (">");

44)  cond_operators     →   "<="
        cond_operators.rep = less_equal.rep;
        less_equal.rep = display ("<=");

45)  cond_operators     →   ">="
        cond_operators.rep = greater_equal.rep;
        greater_equal.rep = display (">=");

46)  expression         →   ¥
```

```
        expression.rep = display ("EXPRESSION");

47)  expression          →   plus_minus_empty  term_list
        expression.rep = plus_minus_empty.rep + term_list.rep;

48)  plus_minus_empty    →   ¥
        plus_minus_empty.rep = display ("PLUS_MINUS_EMPTY");

49)  plus_minus_empty    →   "+"
        plus_minus_empty.rep = plus.rep;
        plus.rep = display ("+");

50)  plus_minus_empty    →   "-"
        plus_minus_empty.rep = minus.rep;
        minus.rep = display ("-");

51)  plus_minus_empty    →   "empty"
        plus_minus_empty.rep = 0;

52)  term_list           →   ¥
        term_list.rep = display ("TERM_LIST");

53)  term_list           →   term
        term_list.rep = term.rep;

54)  term_list           →   term  add_or_sust  term_list
        term_list@1.rep = term.rep + add_or_sust.rep + term_list@2.rep;

55)  add_or_sust         →   ¥
        add_or_sust.rep = display ("ADD_OR_SUST");

56)  add_or_sust         →   "+"
        add_or_sust.rep = plus.rep;
        plus.rep = display ("+");

57)  add_or_sust         →   "-"
        add_or_sust.rep = minus.rep;
        minus.rep = display ("-");

58)  term                →   ¥
        term.rep = display ("TERM");

59)  term                →   factor_list
        term.rep = factor_list.rep;
```

```
60)   factor_list        →   ¥
          factor_list.rep = display ("FACTOR_LIST");

61)   factor_list        →   factor
          factor_list.rep = factor.rep;

62)   factor_list        →   factor  mult_or_div  factor_list
          factor_list@1.rep = factor.rep + mult_or_div.rep + factor_list@2.rep;

63)   factor             →   ¥
          factor.rep = display ("FACTOR");

64)   factor             →   "ident"
          factor.rep = ident.rep;
          ident.rep = display (ident.str);
          ident.str = fonction_id;

65)   factor             →   "number"
          factor.rep = number.rep;
          number.rep = display (number.str);
          number.str = fonction_num;

66)   factor             →   "("  expression  ")"
          factor.rep = pleft.rep + expression.rep + pright.rep;
          pleft.rep = display ("(");
          pright.rep = display (")")

67)   mult_or_div        →   ¥
          mult_or_div.rep = display ("MULT_OR_DIV");

68)   mult_or_div        →   "*"
          mult_or_div.rep = multiply.rep;
          multiply.rep = display ("*");

69)   mult_or_div        →   "/"
          mult_or_div.rep = divide.rep;
          divide.rep = display ("/");
```

Annexe 6.

LE GESTIONNAIRE ARBRE - ECRAN DU LANGAGE PL0.

```
1)   start          →   ¥
         start.pos = 0;

2)   start          →   program
         start.pos = 0;
         program.pos = start.pos;

3)   program        →   ¥

4)   program        →   block "."
         block.pos = program.pos;
         point.pos = block.pos + <decomp>block.rep + 1;

5)   block          →   ¥

6)   block          →   decl_const decl_var decl_proc statement
         decl_const.pos = block.pos;
         decl_var.pos = decl_const.pos + <decomp>decl_const.rep + 1;
         decl_proc.pos = decl_var.pos + <decomp>decl_var.rep + 1;
         statement.pos = decl_proc.pos + <decomp>decl_proc.rep + 1;

7)   decl_const     →   ¥

8)   decl_const     →   "CONST" decl_const_list ";"
         CONST.pos = decl_const.pos;
         decl_const_list.pos = CONST.pos + <decomp>CONST.rep +
                               <decomp>decl_const_list.tab + 1;
         semicolon.pos = decl_const_list.pos + <decomp>decl_const_list.rep + 1;

9)   decl_const     →   "empty"

10)  decl_const_list →   ¥

11)  decl_const_list →   "ident" "=" "number"
         ident.pos = decl_const_list.pos;
         equal.pos = ident.pos + <decomp>ident.rep + 1;
```

```
            number.pos = equal.pos + <decomp>equal.rep + 1;

12)   decl_const_list   →   "ident" "=" "number" "," decl_const_list
            ident.pos = decl_const_list@1.pos;
            equal.pos = ident.pos + <decomp>ident.rep + 1;
            number.pos = equal.pos + <decomp>equal.rep + 1;
            comma.pos = number.pos + <decomp>number.rep + 1;
            decl_const_list@2.pos = comma.pos + <decomp>comma.rep +
                                    <decomp>decl_const_list@2.tab + 1;

13)   decl_var          →   ¥

14)   decl_var          →   "VAR" decl_var_list ";"
            VAR.pos = decl_var.pos;
            decl_var_list.pos =  VAR.pos + <decomp>VAR.rep +
                                 <decomp>decl_var_list.tab + 1;
            semicolon.pos = decl_var_list.pos + <decomp>decl_var_list.rep + 1;

15)   decl_var          →   "empty"

16)   decl_var_list     →   ¥

17)   decl_var_list     →   "ident"
            ident.pos = decl_var_list.pos;

18)   decl_var_list     →   "ident" "," decl_var_list
            ident.pos = decl_var_list@1.pos;
            comma.pos = ident.pos + <decomp>ident.rep + 1;
            decl_var_list@2.pos = comma.pos + <decomp>comma.rep +
                                  <decomp>decl_var_list@2.tab + 1;

19)   decl_proc         →   ¥

20)   decl_proc         →   proc_list ";" decl_proc
            proc_list.pos = decl_proc@1.pos;
            semicolon.pos = proc_list.pos + <decomp>proc_list.rep + 1;
            decl_proc@2.pos = semicolon.pos + <decomp>semicolon.rep +
                              <decomp>decl_proc@2.tab + 1;

21)   decl_proc         →   "empty"

22)   proc_list         →   ¥

23)   proc_list         →   "PROCEDURE" decl_proc_list
```

```
        PROCEDURE.pos = proc_list.pos;
        decl_proc_list.pos = PROCEDURE.pos + <decomp>PROCEDURE.rep + 1;

24)  decl_proc_list  →  ¥

25)  decl_proc_list  →  "ident" ";" block
        ident.pos = decl_proc_list.pos;
        semicolon.pos = ident.pos + <decomp>ident.rep + 1;
        block.pos = semicolon.pos + <decomp>semicolon.rep + <decomp>bloc.tab + 1;

26)  statement  →  ¥

27)  statement  →  "ident" ":=" expression
        ident.pos = statement.pos;
        assign.pos = ident.pos + <decomp>ident.rep + 1;
        expression.pos = assign.pos + <decomp>assign.rep + 1;

28)  statement  →  "CALL" ident
        CALL.pos = statement.pos;
        ident.pos = CALL.pos + <decomp>CALL.rep + 1;

29)  statement  →  "BEGIN" statement_list "END"
        BEGIN.pos = statement.pos;
        statement_list.pos = BEGIN.pos + <decomp>BEGIN.rep +
                             <decomp>statement_list.tab + 1;
        END.pos = statement_list.pos + <decomp>statement_list.rep + 1;

30)  statement  →  "IF" condition "THEN" statement
        IF.pos = statement@1.pos;
        condition.pos = IF.pos + <decomp>IF.rep + 1;
        THEN.pos = condition.pos + <decomp>condition.rep + 1;
        statement@2.pos = THEN.pos + <decomp>THEN.rep +
                          <decomp>statement@2.tab + 1;

31)  statement  →  "WHILE" condition "DO" statement
        WHILE.pos = statement@1.pos;
        condition.pos = WHILE.pos + <decomp>WHILE.rep + 1;
        DO.pos = condition.pos + <decomp>condition.rep + 1;
        statement@2.pos =  DO.pos + <decomp>DO.rep +
                           <decomp>statement@2.tab + 1;

32)  statement  →  "empty"

33)  statement_list  →  ¥
```

```
34)  statement_list  →  statement
         statement.pos = statement_list.pos;

35)  statement_list  →  statement ";" statement_list
         statement.pos = statement_list.pos;
         semicolon.pos = statement.pos + <decomp>statement.rep + 1;
         statement_list@2.pos = semicolon.pos + <decomp>semicolon.rep + 1;

36)  condition  →  ¥

37)  condition  →  "ODD" expression
         ODD.pos = condition.pos;
         expression.pos = condition.pos + <decomp>condition.rep + 1;

38)  condition  →  expression cond_operators expression
         expression@1.pos = condition.pos;
         cond_operators.pos = expression@1.pos + <decomp>expression@1.rep + 1;
         expression@2.pos = cond_operators.pos + <decomp>cond_operators.rep + 1;

39)  cond_operators  →  ¥

40)  cond_operators  →  "="
         equal.pos = cond_operators.pos;

41)  cond_operators  →  "<>"
         not_equal.pos = = cond_operators.pos;

42)  cond_operators  →  "<"
         less.pos = = cond_operators.pos;

43)  cond_operators  →  ">"
         greater.pos = cond_operators.pos;

44)  cond_operators  →  "<="
         less_equal.pos = = cond_operators.pos;

45)  cond_operators  →  ">="
         greater_equal.pos = = cond_operators.pos;

46)  expression  →  ¥

47)  expression  →  plus_minus_empty term_list
         plus_minus_empty.pos = expression.pos;
```

```
        term_list.pos = plus_minus_empty.pos + <decomp>plus_minus_empty.rep + 1;

48)  plus_minus_empty  →  ¥

49)  plus_minus_empty  →  "+"
        plus.pos = plus_minus_empty.pos;

50)  plus_minus_empty  →  "-"
        minus.pos = plus_minus_empty.pos;

51)  plus_minus_empty  →  "empty"

52)  term_list         →  ¥

53)  term_list         →  term
        term.pos = term_list.pos;

54)  term_list         →  term  add_or_sust  term_list
        term.pos = term_list@1.pos;
        add_or_sust.pos = term.pos + <decomp>term.rep + 1;
        term_list@2.pos = add_or_sust.pos + <decomp>add_or_sust.rep + 1;

55)  add_or_sust       →  ¥

56)  add_or_sust       →  "+"
        plus.pos = add_or_sust.pos;

57)  add_or_sust       →  "-"
        minus.pos = add_or_sust.pos;

58)  term              →  ¥

59)  term              →  factor_list
        factor_list.pos = term.pos;

60)  factor_list       →  ¥

61)  factor_list       →  factor
        factor.pos = factor_list.pos;

62)  factor_list       →  factor  mult_or_div  factor_list
        factor.pos = factor_list@1.pos;
```

```
        mult_or_div.pos = factor.pos + <decomp>factor.rep +1;
        factor_list@2.pos = mult_or_div.pos + <decomp>mult_or_div.rep + 1;

63)  factor          →  ¥

64)  factor          →  "ident"
        ident.pos = factor.pos;

65)  factor          →  "number"
        number.pos = factor.pos;

66)  factor          →  "(" expression ")"
        pleft.pos = factor.pos;
        expression.pos = pleft.pos + <decomp>pleft.rep + 1;
        pright.pos = expression.pos + <decomp>expression.rep + 1;

67)  mult_or_div     →  ¥

68)  mult_or_div     →  "*"
        multiply.pos = mult_or_div.pos;

69)  mult_or_div     →  "/"
        divide.pos = mult_or_div.pos;
```

Annexe 7.

LE DOCUMENTALISTE DU LANGAGE PL0.

1) start → ¥
   start.tsym = {∅};
   start.acproc = "Programme_Principal";

2) start → program
   start.tsym = program.tsym;
   program.acproc = start.acproc;

3) program → ¥
   program.tsym = {∅};

4) program → block "."
   program.tsym = block.decl ∪ block.util;
   block.acproc = program.acproc;
   block.iconst = {∅};
   block.ivar = {∅};

5) block → ¥
   block.decl = block.iconst ∪ block.ivar;
   block.util = {∅};

6) block → decl_const decl_var decl_proc statement
   block.decl = (decl_const.sconst ∪ decl_var.svar ∪ decl_proc.sproc) ⊕
   (block.iconst ∪ block.ivar);
   block.util = statement.util;
   decl_proc.iconst = decl_const.sconst ⊕ (block.iconst ∪ block.ivar);
   decl_proc.ivar = decl_var.svar ⊕ (block.iconst ∪ block.ivar);
   statement.iconst = decl_const.sconst ⊕ (block.iconst ∪ block.ivar);
   statement.ivar = decl_var.svar ⊕ (block.iconst ∪ block.ivar);

statement.iproc = decl_proc.sproc;
decl_const.acproc = block.acproc;
decl_var.acproc = block.acproc;
decl_proc.acproc = block.acproc;
statement.acproc = block.acproc;
decl_const.util = statement.util;
decl_var.util = statement.util;
ASSERT: decl_const.sconst ∩ decl_var.svar = {∅};
ASSERT: decl_const.sconst ∩ decl_proc.sproc = {∅};
ASSERT: decl_var.svar ∩ decl_proc.sproc = {∅};

7) decl_const → ¥
decl_const.sconst = {∅};

8) decl_const → "CONST" decl_const_list ";"
decl_const.sconst = decl_const_list.sconst;
decl_const_list.acproc = decl_const.acproc;
decl_const_list.util = decl_const.util;

9) decl_const → "empty"
decl_const.sconst = {∅};

10) decl_const_list → ¥
decl_const_list.sconst = {∅};

11) decl_const_list → "ident" "=" "number"
ident.decl = decl_const_list.acproc;
ident.valeur = <decomp>number.str;
ident.dligne = ligne (<scrmgr>ident.pos);
decl_const_list.sconst = {<decomp>ident.str, ident.decl, ident.valeur, ident.dligne};
ident.util = prendre_util (<decomp>ident.str, decl_const_list.util);
ident.uligne = prendre_uligne (<decomp>ident.str, decl_const_list.util);

12) decl_const_list → "ident" "=" "number" "," decl_const_list
ASSERT: {<decomp>ident.str} ∉ decl_const_list@2.sconst;

ident.decl = decl_const_list@1.acproc;
ident.valeur = <decomp>number.str;
ident.dligne = ligne (<scrmgr>ident.pos);
decl_const_list@1.sconst = {<decomp>ident.str, ident.decl, ident.valeur, ident.dligne} ∪ decl_const_list@2.sconst;
ident.util = prendre_util (<decomp>ident.str, decl_const_list@1.util);
ident.uligne = prendre_uligne (<decomp>ident.str, decl_const_list@1.util);
decl_const_list@2.acproc = decl_const_list@1.acproc;
decl_const_list@2.util = decl_const_list@1.util;

13) decl_var → ¥
decl_var.svar = {∅};

14) decl_var → "VAR" decl_var_list ";"
decl_var.svar = decl_var_list.svar;
decl_var_list.acproc = decl_var.acproc;
decl_var_list.util = decl_var.util;

15) decl_var → "empty"
decl_var.svar = {∅};

16) decl_var_list → ¥
decl_var_list.svar = {∅};

17) decl_var_list → "ident"
ident.decl = decl_var_list.acproc;
ident.dligne = ligne (<scrmgr>ident.pos);
decl_var_list.svar = {<decomp>ident.str, ident.decl, ident.dligne};
ident.util = prendre_util (<decomp>ident.str, decl_var_list.util);
ident.uligne = prendre_uligne (<decomp>ident.str, decl_var_list.util);

18) decl_var_list → "ident" "," decl_var_list
ASSERT: {<decomp>ident.str} ∉ decl_var_list@2.svar;
ident.decl = decl_var_list@1.acproc;
ident.dligne = ligne (<scrmgr>ident.pos);

decl_var_list@1.svar = {<decomp>ident.str, ident.decl, ident.dligne} ∪ decl_var_list@2.svar;
ident.util = prendre_util (<decomp>ident.str, decl_var_list@1.util);
ident.uligne = prendre_uligne (<decomp>ident.str, decl_var_list@1.util);
decl_var_list@2.acproc = decl_var_list@1.acproc;
decl_var_list@2.util = decl_var_list@1.util;

19) decl_proc → ¥
decl_proc.sproc = {∅};

20) decl_proc → proc_list ";" decl_proc
ASSERT: proc_list.sproc ∉ decl_proc@2.sproc;
decl_proc@1.sproc = proc_list.sproc ∪ decl_proc@2.sproc;
decl_proc@2.iconst = decl_proc@1.iconst;
decl_proc@2.ivar = decl_proc@1.ivar;
decl_proc@2.acproc = decl_proc@1.acproc;
proc_list.iconst = decl_proc@1.iconst;
proc_list.ivar = decl_proc@1.ivar;
proc_list.acproc = decl_proc@1.acproc;

21) decl_proc → "empty"
decl_proc.sproc = {∅};

22) proc_list → ¥
proc_list.sproc = {∅};

23) proc_list → "PROCEDURE" decl_proc_list
proc_list.sproc = decl_proc_list.sproc;
decl_proc_list.iconst = proc_list.iconst;
decl_proc_list.ivar = proc_list.ivar;
decl_proc_list.acproc = proc_list.acproc;

24) decl_proc_list → ¥
decl_proc_list.sproc = {∅};

25) decl_proc_list → "ident" ";" block

block.iconst = decl_proc_list.iconst;
block.ivar = decl_proc_list.ivar;
block.acproc = <decomp>ident.str;
ident.decl = decl_proc_list.acproc;
ident.dligne = ligne (<scrmgr>ident.pos);
decl_proc_list.sproc = {<decomp>ident.str, ident.decl, ident.ligne};

26) statement → ¥

27) statement → "ident" ":=" expression
ASSERT: {<decomp>ident.str} ∈ statement.ivar;
ASSERT: {<decomp>ident.str} ∉ statement.iconst;
ASSERT: {<decomp>ident.str} ∉ statement.iproc;
expression.iconst = statement.iconst;
expression.ivar = statement.ivar;
expression.acproc = statement.acproc;
ident.decl = prendre_decl (<decomp>ident.str, statement.iconst, statement.ivar);
ident.dligne = prendre_dligne (<decomp>ident.str, (statement.iconst, statement.ivar);
ident.util = statement.acproc;
ident.uligne = ligne (<scrmgr>ident.pos);
statement.util = {<decomp>ident.str, ident.util, ident.uligne} ∪ expression.util;

28) statement → "CALL" ident
ASSERT: {<decomp>ident.str} ∉ statement.ivar;
ASSERT: {<decomp>ident.str} ∉ statement.iconst;
ASSERT: {<decomp>ident.str} ∈ statement.iproc;
ident.decl = prendre_decl (<decomp>ident.str, statement.iproc);
ident.util = statement.acproc;
ident.uligne = ligne (<decomp>ident.pos);
statement.util = {<decomp>ident.str, ident.util, ident.uligne);

29) statement → "BEGIN" statement_list "END"
statement_list.iconst = statement.iconst;
statement_list.ivar = statement.ivar;
statement_list.iproc = statement.iproc;

statement_list.acproc = statement.acproc;
statement.util = statement_list.util;

30) statement → "IF" condition "THEN" statement
condition.iconst = statement.iconst;
condition.ivar = statement.const;
condition.acproc = statement@1.acproc;
statement@2.iconst = statement@1.iconst;
statement@2.ivar = statement@1.ivar;
statement@2.iproc = statement@1.iproc;
statement@2.acproc = statement@1.acproc;
statement@1.util = condition.util ∪ statement@2.util;

31) statement → "WHILE" condition "DO" statement
condition.iconst = statement.iconst;
condition.ivar = statement.const;
condition.acproc = statement@1.acproc;
statement@2.iconst = statement@1.iconst;
statement@2.ivar = statement@1.ivar;
statement@2.iproc = statement@1.iproc;
statement@2.acproc = statement@1.acproc;
statement@1.util = condition.util ∪ statement@2.util;

32) statement → "empty"

33) statement_list → ¥

34) statement_list → statement
statement.iconst = statement_list.iconst;
statement.ivar = statement_list.ivar;
statement.iproc = statement_list.iproc;
statement.acproc = statement_list.acproc;
statement_list.util = statement.util;

35) statement_list → statement ";" statement_list
statement.iconst = statement_list@1.iconst;

statement.ivar = statement_list@1.ivar;
statement.iproc = statement_list@1.iproc;
statement_list@2.iconst = statement_list@1.iconst;
statement_list@2.ivar = statement_list@1.ivar;
statement_list@2.iproc = statement_list@1.iproc;
statement.acproc = statement_list@1.acproc;
statement_list@2.acproc = statemen_list@1.acproc;
statement_list@1.util = statement.util ∪ statement_list@2.util;

36) condition → ¥

37) condition → "ODD" expression
expression.iconst = condition.iconst;
expression.ivar = condition.ivar;
expression.acproc = condition.acproc;
condition.util = expression.util;

38) condition → expression cond_operators expression
expression@1.iconst = condition.iconst;
expression@1.ivar = condition.ivar;
expression@1.acproc = condition.acproc;
expression@2.iconst = condition.iconst;
expression@2.ivar = condition.ivar;
expression@2.acproc = condition.acproc;
condition.util = expression@1.util ∪ expression@2.util;

39) cond_operators → ¥

40) cond_operators → "="

41) cond_operators → "<>"

42) cond_operators → "<"

43) cond_operators → ">"

44) cond_operators → "<="

45) cond_operators → ">="

46) expression → ¥

47) expression → plus_minus_empty term_list
term_list.iconst = expression.iconst;
term_list.ivar = expression.ivar;
term_list.acproc = expression.acproc;
expression.util = term_list.util;

48) plus_minus_empty → ¥

49) plus_minus_empty → "+"

50) plus_minus_empty → "-"

51) plus_minus_empty → "empty"

52) term_list → ¥

53) term_list → term
term.iconst = term_list.iconst;
term.ivar = term_list.ivar;
term.acproc = term_list.acproc;
term_list.util = term.util;

54) term_list → term add_or_sust term_list
term.iconst = term_list@1.iconst;
term.ivar = term_list@1.ivar;
term.acproc = term_list@1.acproc;
term_list@2.iconst = term_list@1.ivar;
term_list@2.ivar = term_list@1.ivar;
term_list@2.acproc = term_list@1.acproc;
term_list@1.util = term.util ∪ term_list@2.util;

55) add_or_sust → ¥

56) add_or_sust → "+"

57) add_or_sust → "-"

58) term → ¥

59) term → factor_list
factor_list.iconst = term.iconst;
factor_list.ivar = term.ivar;
factor_list.acproc = term.acproc;
term.util = factor_list.util;

60) factor_list → ¥

61) factor_list → factor
factor.iconst = factor_list.iconst;
factor.ivar = factor_list.ivar;
factor.acproc = factor_list.acproc;
factor_list.util = factor.util;

62) factor_list → factor mult_or_div factor_list
factor.iconst = factor_list@1.iconst;
factor.ivar = factor_list@1.ivar;
factor.acproc = factor_list@1.acproc;
factor_list@2.iconst = factor_list@1.iconst;
factor_list@2.ivar = factor_list@1.ivar;
factor_list@2.acproc = factor_list@1.acproc;
factor_list@1.util = factor.util ∪ factor_list@2.util;

63) factor → ¥

64) factor → "ident"
ASSERT: {<decomp>ident.str} ∈ {factor.iconst ∪ factor.ivar};
ident.decl = prendre_decl (<decomp>ident.str, factor.iconst, factor.ivar);

ident.dligne = prendre_dligne (<decomp>ident.str, factor.iconst, factor.ivar);
ident.valeur = prendre_val (<decomp>ident.str, factor.iconst, factor.ivar);
ident.util = factor.acproc;
ident.uligne = ligne (<scrmgr>ident.pos);
factor.util = {<decomp>ident.str, ident.util, ident.uligne};

65) factor → "number"

66) factor → "(" expression ")"
expression.iconst = factor.iconst;
expression.ivar = factor.ivar;
expression.acproc = factor.acproc;

67) mult_or_div → ¥

68) mult_or_div → "*"

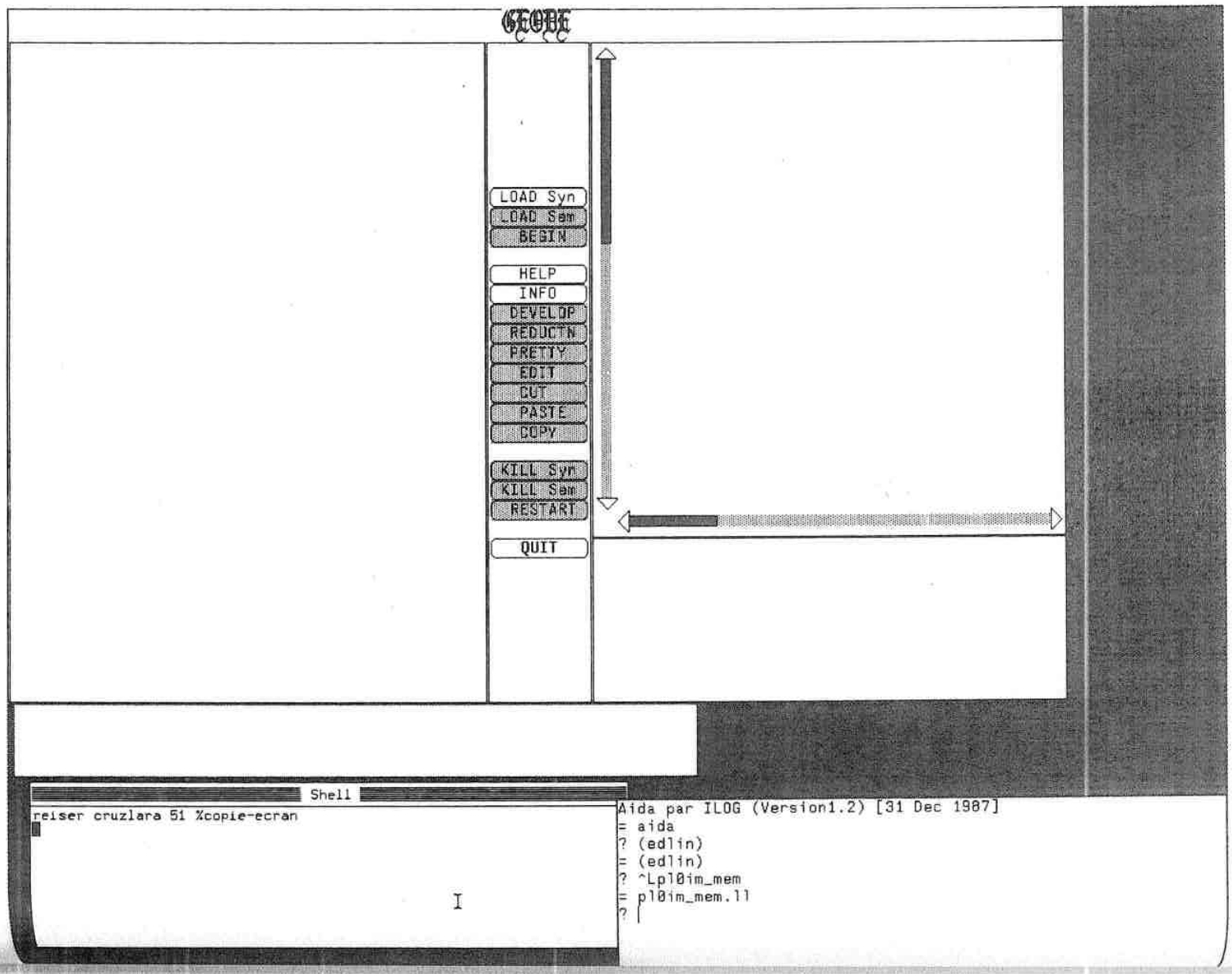
69) mult_or_div → "/"

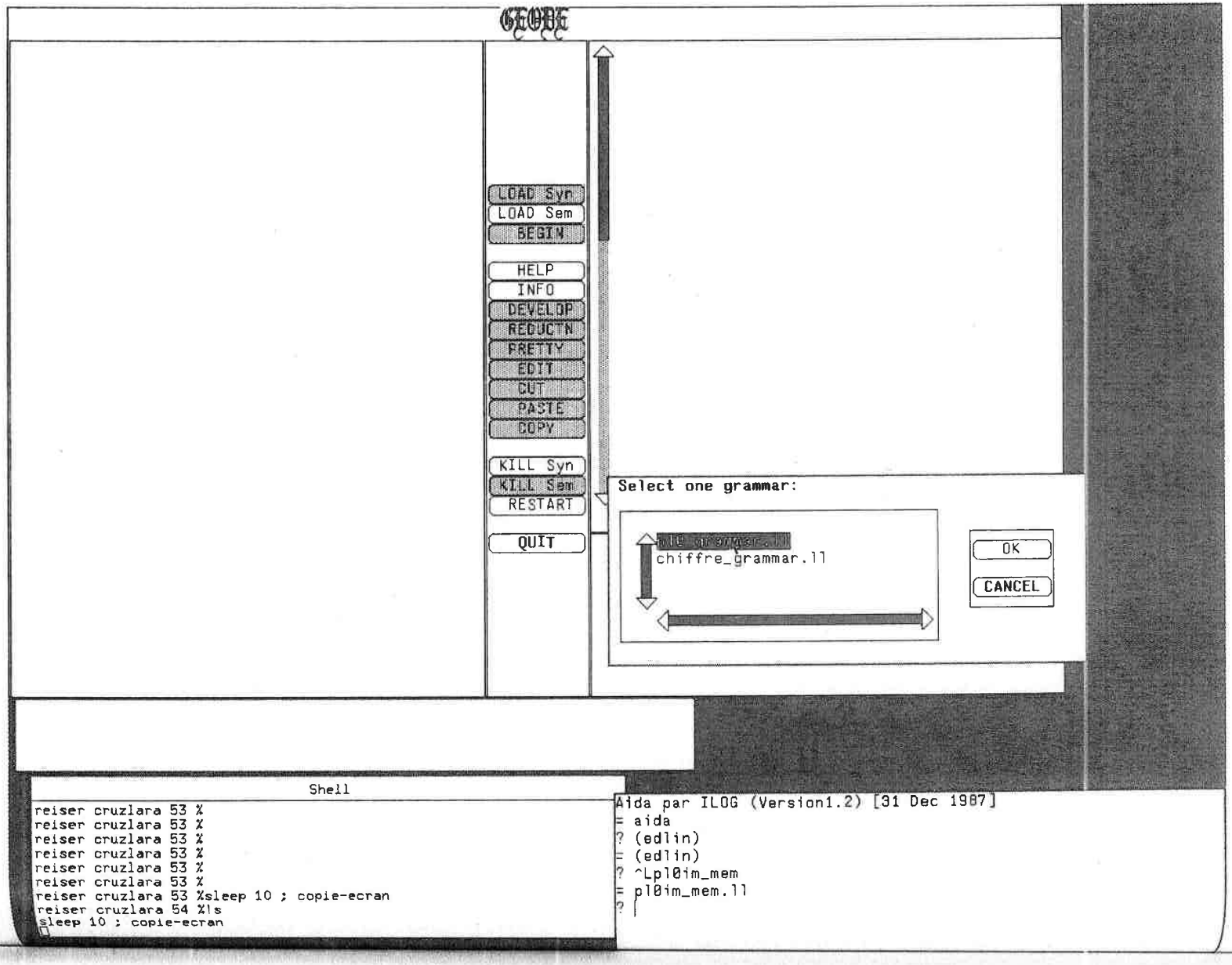
Annexe 8.

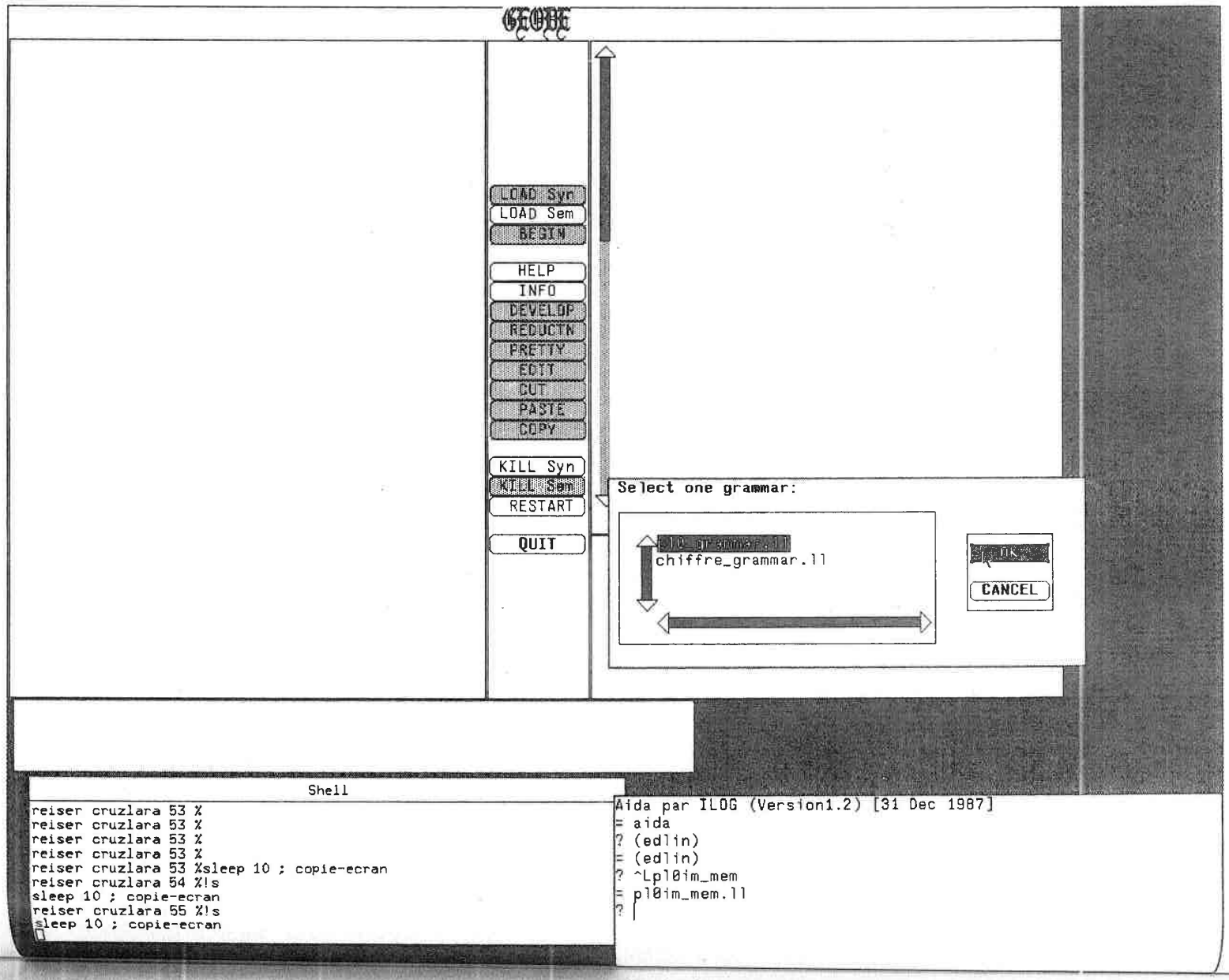
L'INTERFACE UTILISATEUR SUN DE GEODE.

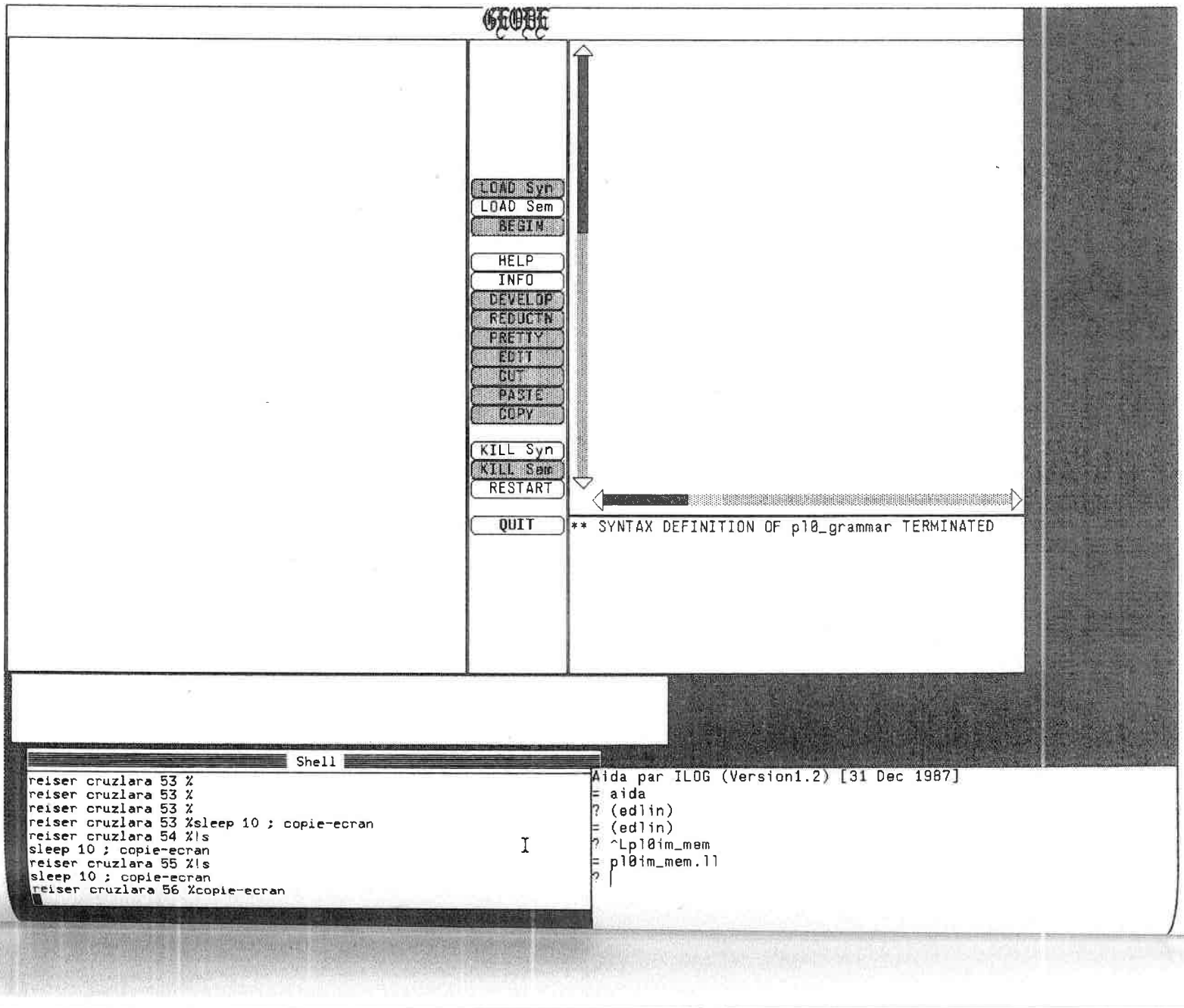
Nous montrons ici l'interface utilisateur de GEODE sur une machine SUN sous UNIX. L'interface a été implantée en Le_Lisp 15.2 [Chailloux 1986] et Aïda 1.2 [ILOG 1987].

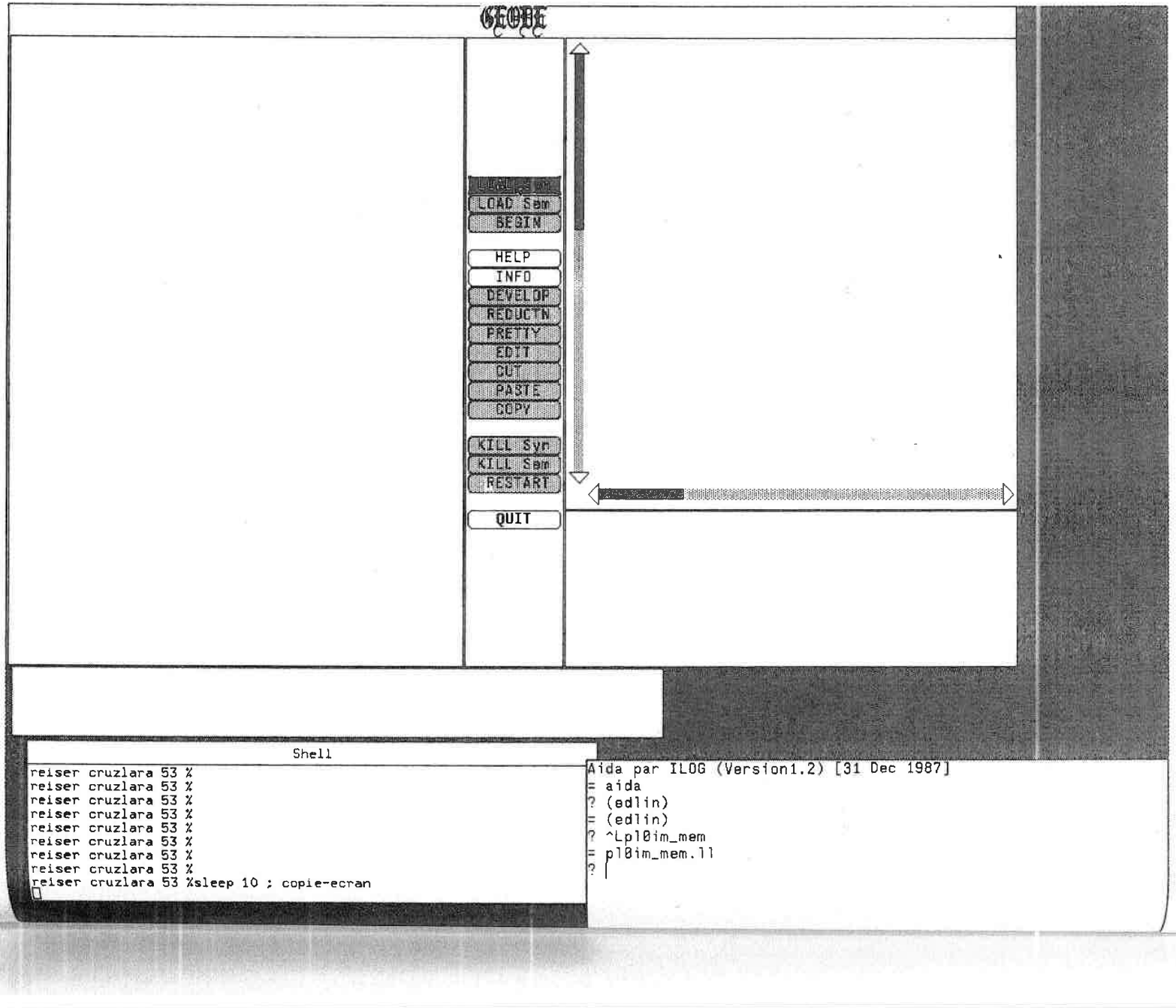
Cette annexe est constituée de plusieurs copies d'écran, qui montrent le déroulement d'une session utilisateur sous un environnement généré par GEODE, pour le langage de programmation PL0.

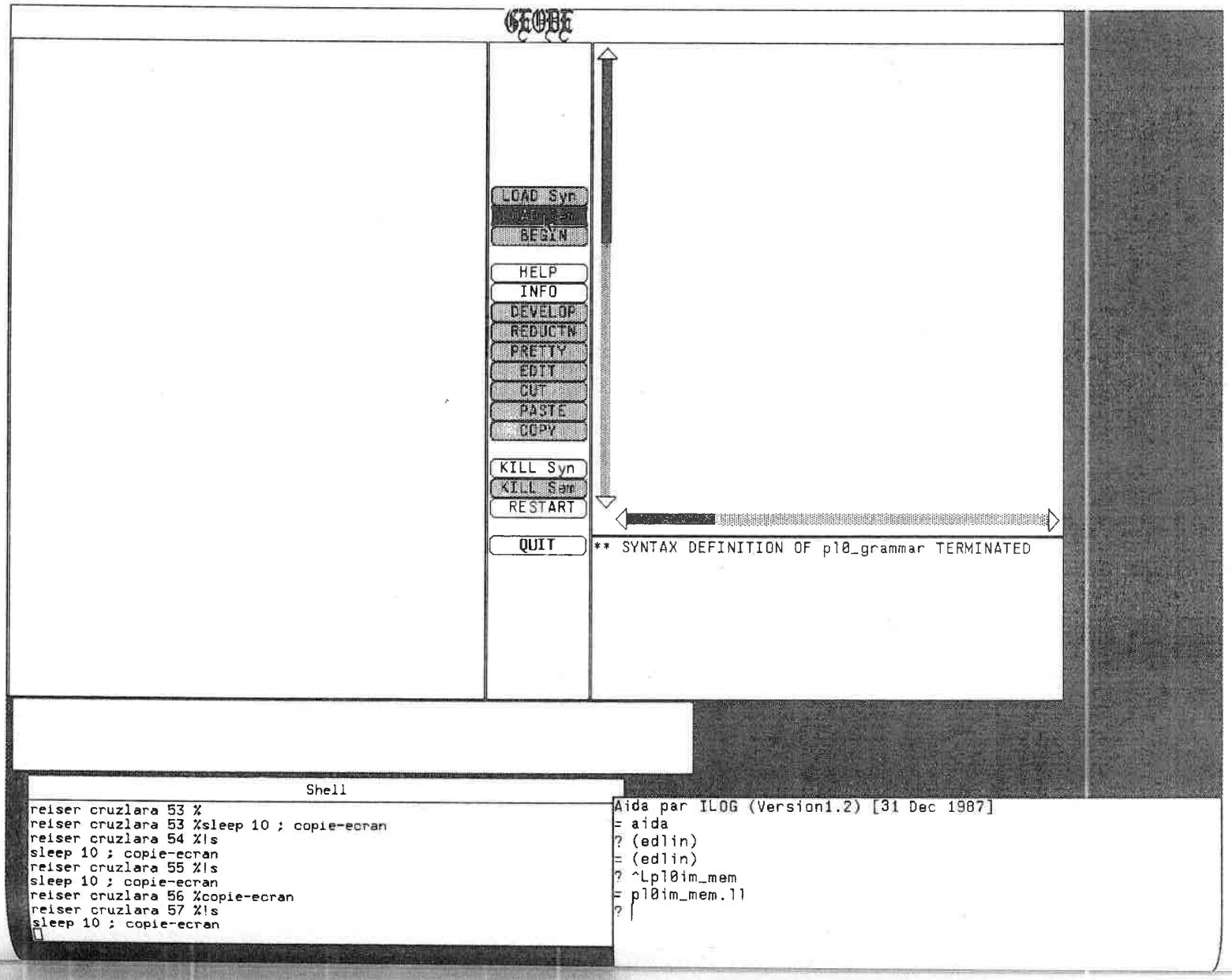

GEODE
LOAD Syn
LOAD Sem
BEGIN
HELP
INFO
DEVELOP
REDUCTN
PRETTY
EDIT
CUT
PASTE
COPY
KILL Syn
KILL Sem
RESTART
QUIT
Shell
reiser cruzlara 51 %copie-ecran
Aida par ILOG (Version1.2) [31 Dec 1987]
= aida
? (edlin)
= (edlin)
? ^Lpl0im_mem
= pl0im_mem.ll
?

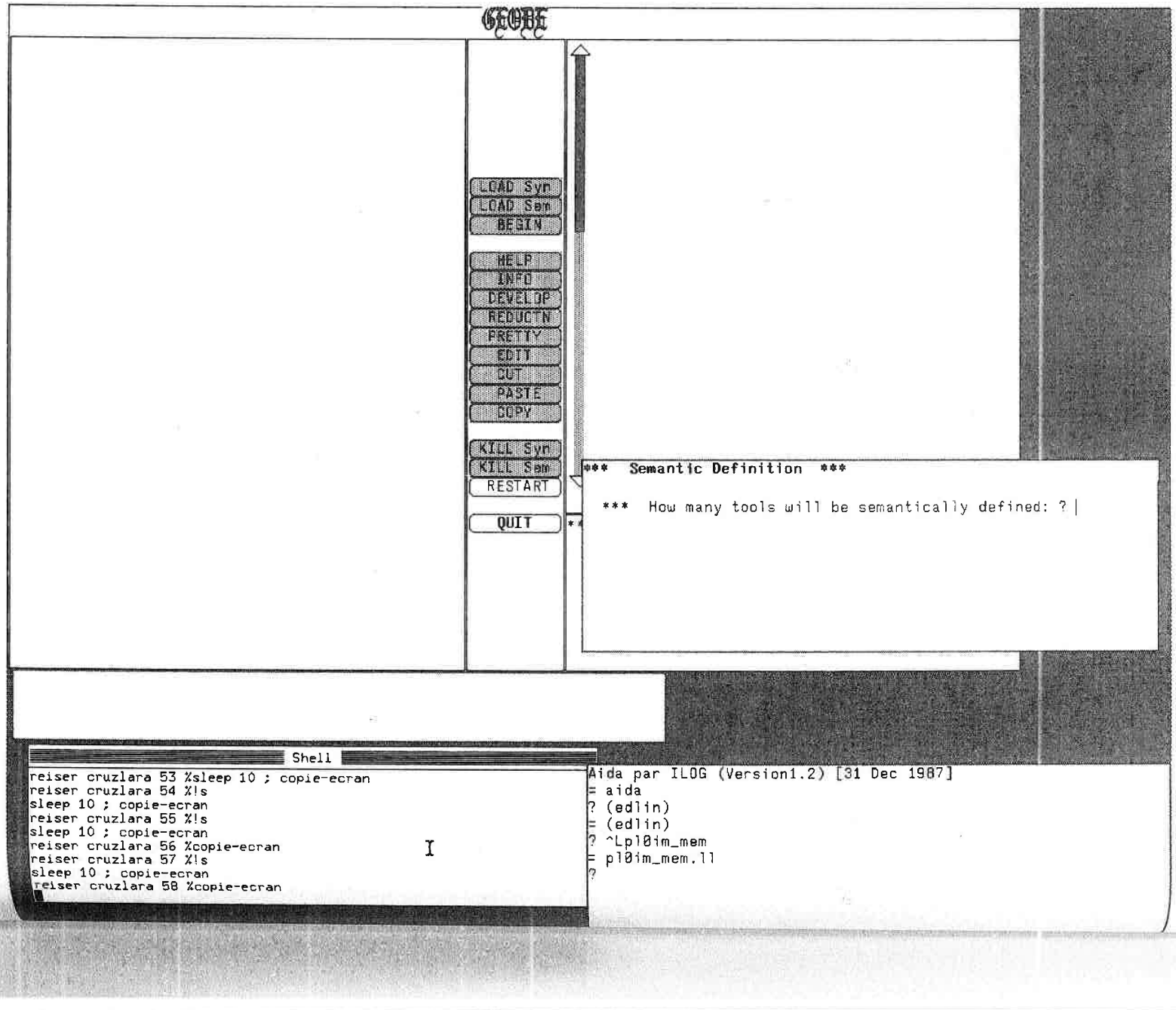

GLOBE
LOAD Syn
LOAD Sem
BEGIN
HELP
INFO
DEVELOP
REDUCTN
PRETTY
EDIT
CUT
PASTE
COPY
KILL Syn
KILL Sem
RESTART
QUIT
Select one grammar:
chiffre_grammar.ll
OK
CANCEL
Shell
reiser cruzlara 53 %
reiser cruzlara 53 %
reiser cruzlara 53 %
reiser cruzlara 53 %
reiser cruzlara 53 %
reiser cruzlara 53 %
reiser cruzlara 53 %sleep 10 ; copie-ecran
reiser cruzlara 54 %ls
sleep 10 ; copie-ecran
Aida par ILOG (Version1.2) [31 Dec 1987]
= aida
? (edlin)
= (edlin)
? ^Lpl0im_mem
= pl0im_mem.ll
?

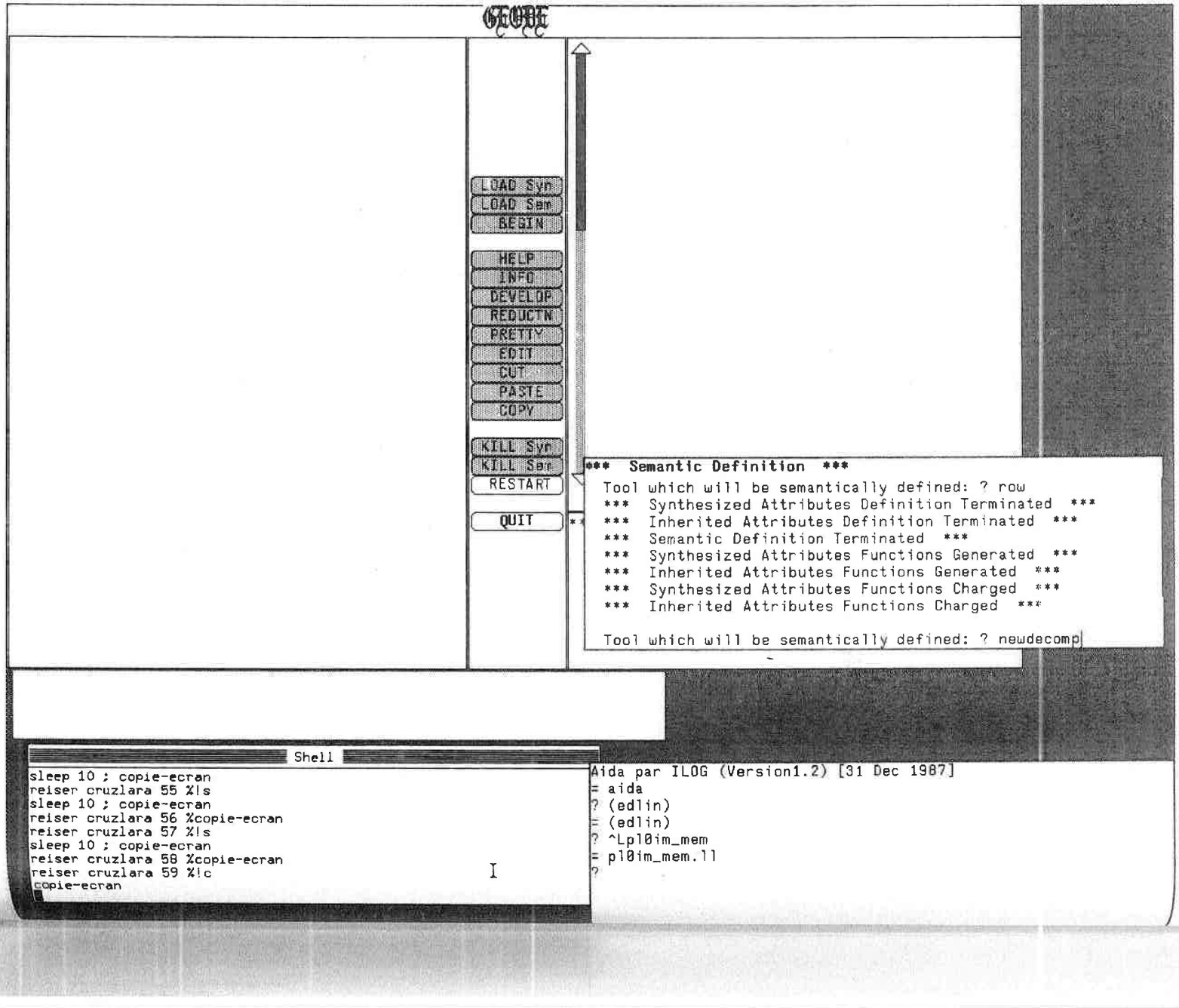

GLOBE
LOAD Syn
LOAD Sem
BEGIN
HELP
INFO
DEVELOP
REDUCTN
PRETTY
EDIT
CUT
PASTE
COPY
KILL Syn
KILL Sem
RESTART
QUIT
Select one grammar:
chiffre_grammar.ll
OK
CANCEL
Shell
reiser cruzlara 53 %
reiser cruzlara 53 %
reiser cruzlara 53 %
reiser cruzlara 53 %
reiser cruzlara 53 %sleep 10 ; copie-ecran
reiser cruzlara 54 %ls
sleep 10 ; copie-ecran
reiser cruzlara 55 %ls
sleep 10 ; copie-ecran
Aida par ILOG (Version1.2) [31 Dec 1987]
= aida
? (edlin)
= (edlin)
? ^Lpl0im_mem
= pl0im_mem.ll
?

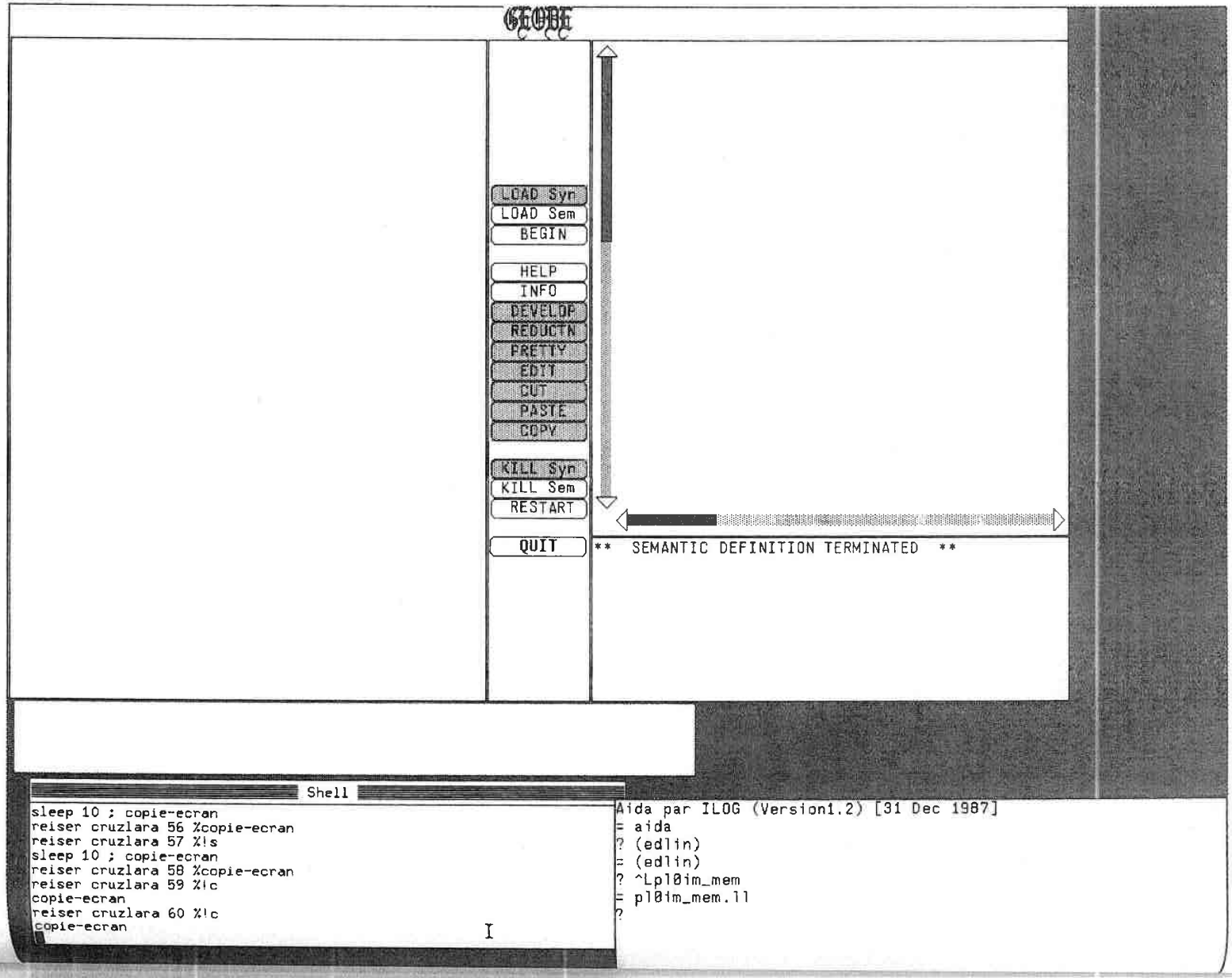

GLOBE
LOAD Syn
LOAD Sem
BEGIN
HELP
INFO
DEVELOP
REDUCTN
PRETTY
EDIT
CUT
PASTE
COPY
KILL Syn
KILL Sem
RESTART
QUIT
** SYNTAX DEFINITION OF p10_grammar TERMINATED
Shell
reiser cruzlara 53 %
reiser cruzlara 53 %
reiser cruzlara 53 %
reiser cruzlara 53 %sleep 10 ; copie-ecran
reiser cruzlara 54 %ls
sleep 10 ; copie-ecran
reiser cruzlara 55 %ls
sleep 10 ; copie-ecran
reiser cruzlara 56 %copie-ecran
Aida par ILOG (Version1.2) [31 Dec 1987]
= aida
? (edlin)
= (edlin)
? ^Lp10im_mem
= p10im_mem.ll
?

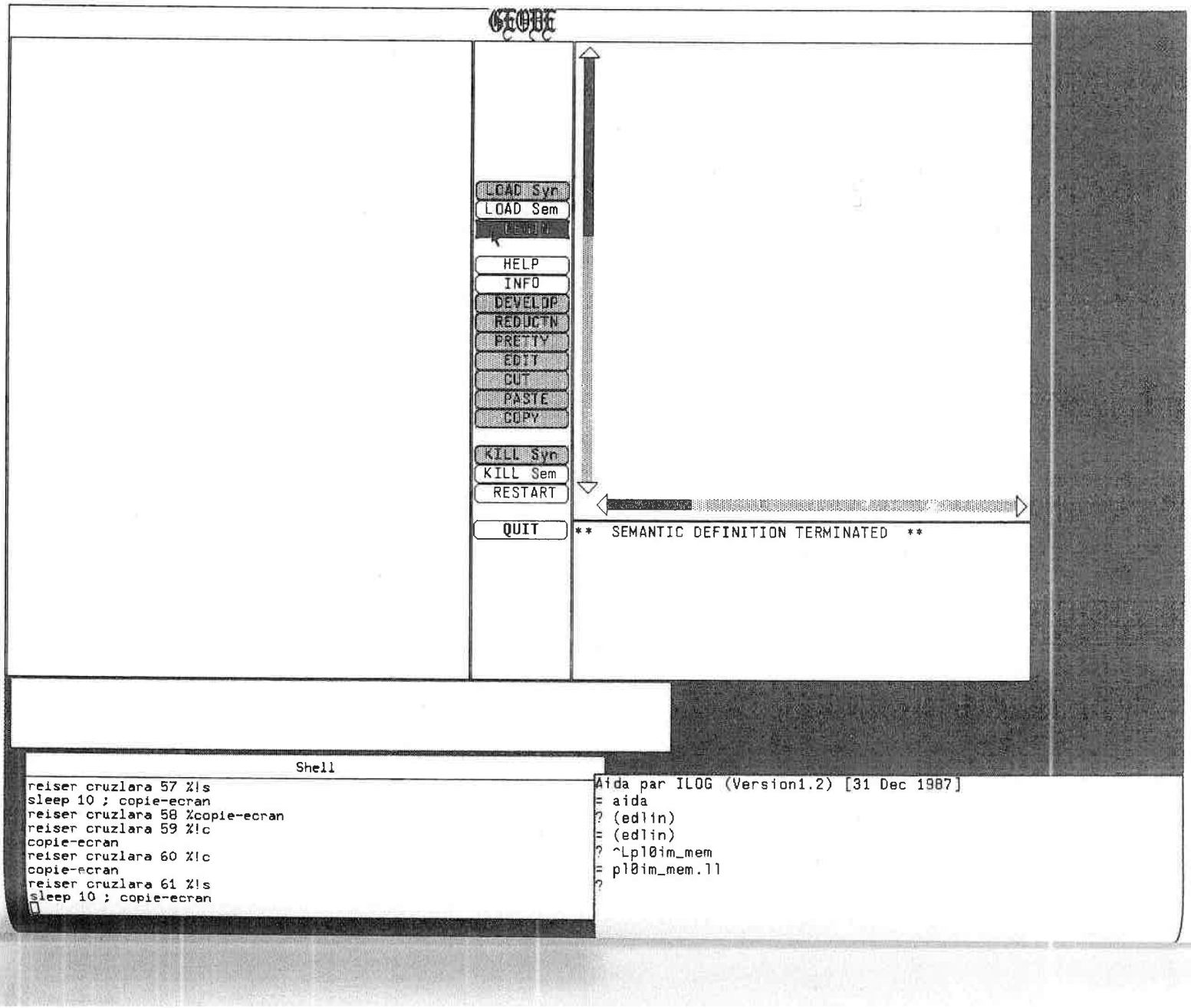

GEODE
LOAD Sem
BEGIN
HELP
INFO
DEVELOP
REDUCTN
PRETTY
EDIT
CUT
PASTE
COPY
KILL Svr
KILL Sem
RESTART
QUIT
Shell
reiser cruzlara 53 %
reiser cruzlara 53 %
reiser cruzlara 53 %
reiser cruzlara 53 %
reiser cruzlara 53 %
reiser cruzlara 53 %
reiser cruzlara 53 %
reiser cruzlara 53 %
reiser cruzlara 53 %sleep 10 ; copie-ecran
Aida par ILOG (Version1.2) [31 Dec 1987]
= aida
? (edlin)
= (edlin)
? ^Lpl0im_mem
= pl0im_mem.ll
?

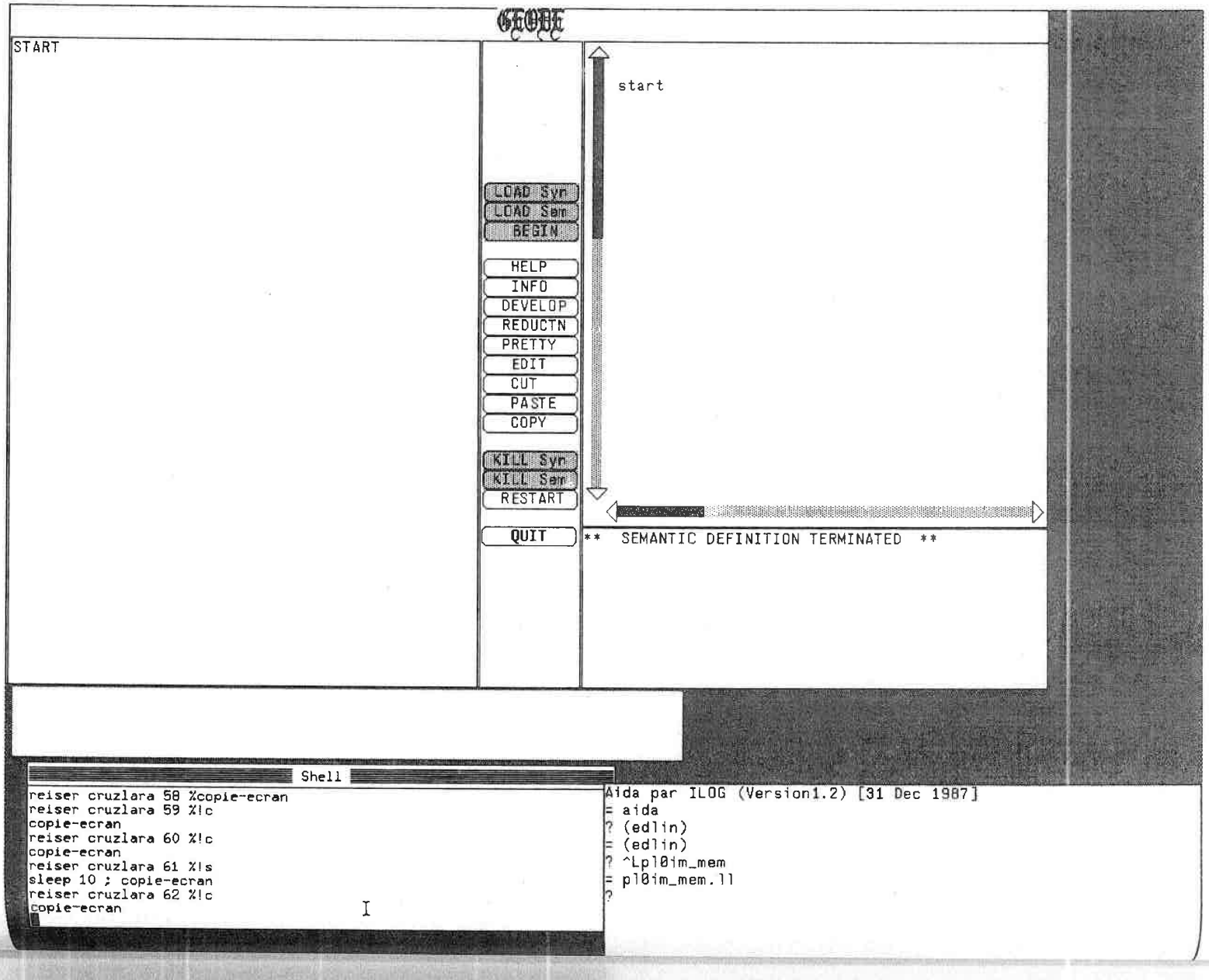

GLOBE
LOAD Syn
LOAD Sem
BEGIN
HELP
INFO
DEVELOP
REDUCTN
PRETTY
EDIT
CUT
PASTE
COPY
KILL Syn
KILL Sem
RESTART
QUIT
** SYNTAX DEFINITION OF p10_grammar TERMINATED
Shell
reiser cruzlara 53 %
reiser cruzlara 53 %sleep 10 ; copie-ecran
reiser cruzlara 54 %ls
sleep 10 ; copie-ecran
reiser cruzlara 55 %ls
sleep 10 ; copie-ecran
reiser cruzlara 56 %copie-ecran
reiser cruzlara 57 %ls
sleep 10 ; copie-ecran
Aida par ILOG (Version1.2) [31 Dec 1987]
= aida
? (edlin)
= (edlin)
? ^Lp10im_mem
= p10im_mem.ll
?

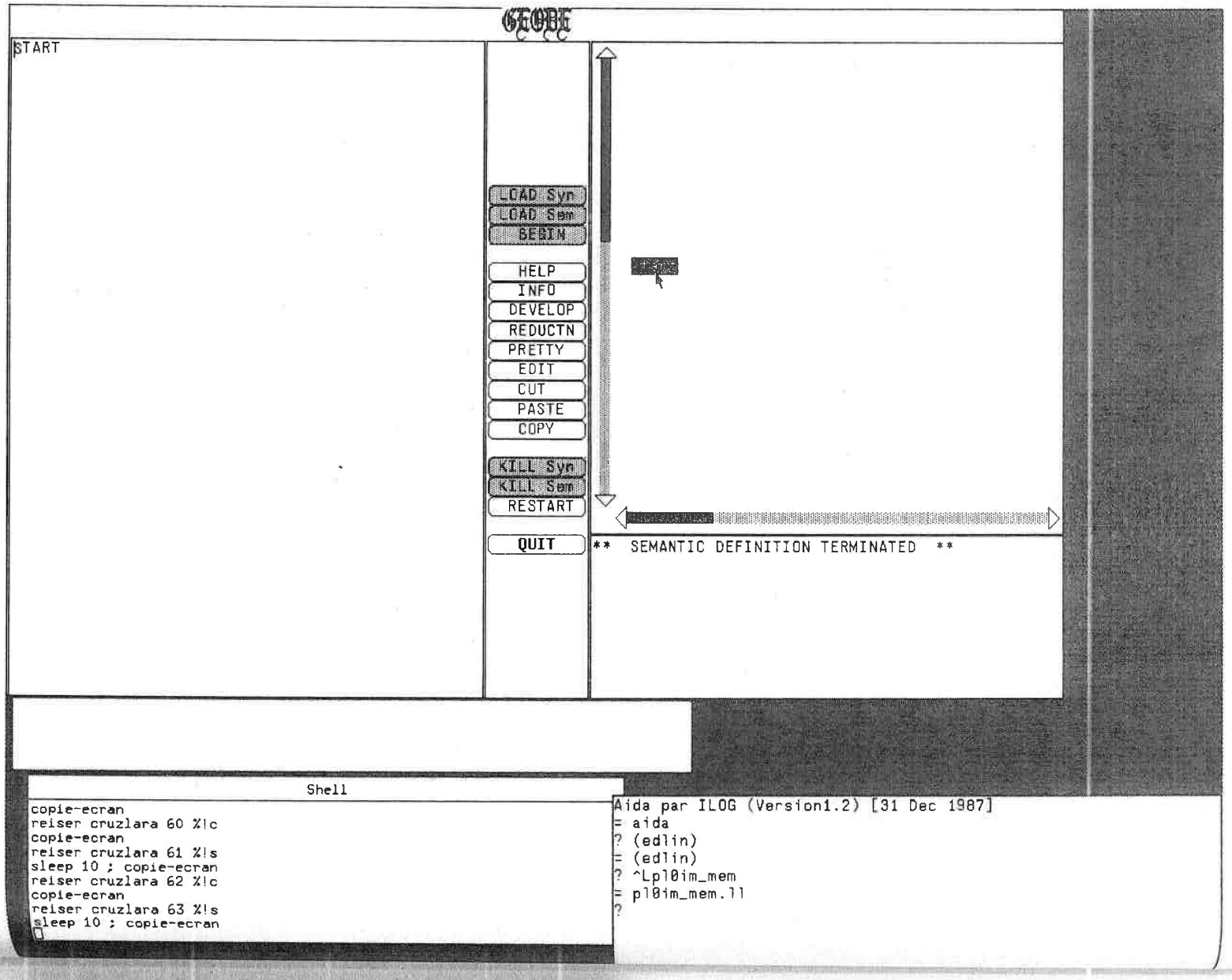

GLOBE
LOAD Syn
LOAD Sem
BEGIN
HELP
INFO
DEVELOP
REDUCTN
PRETTY
EDIT
CUT
PASTE
COPY
KILL Syn
KILL Sem
RESTART
QUIT
*** Semantic Definition ***
*** How many tools will be semantically defined: ?
Shell
reiser cruzlara 53 %sleep 10 ; copie-ecran
reiser cruzlara 54 %!s
sleep 10 ; copie-ecran
reiser cruzlara 55 %!s
sleep 10 ; copie-ecran
reiser cruzlara 56 %copie-ecran
reiser cruzlara 57 %!s
sleep 10 ; copie-ecran
reiser cruzlara 58 %copie-ecran
Aida par ILOG (Version1.2) [31 Dec 1987]
= aida
? (edlin)
= (edlin)
? ^Lpl0im_mem
= pl0im_mem.ll
?

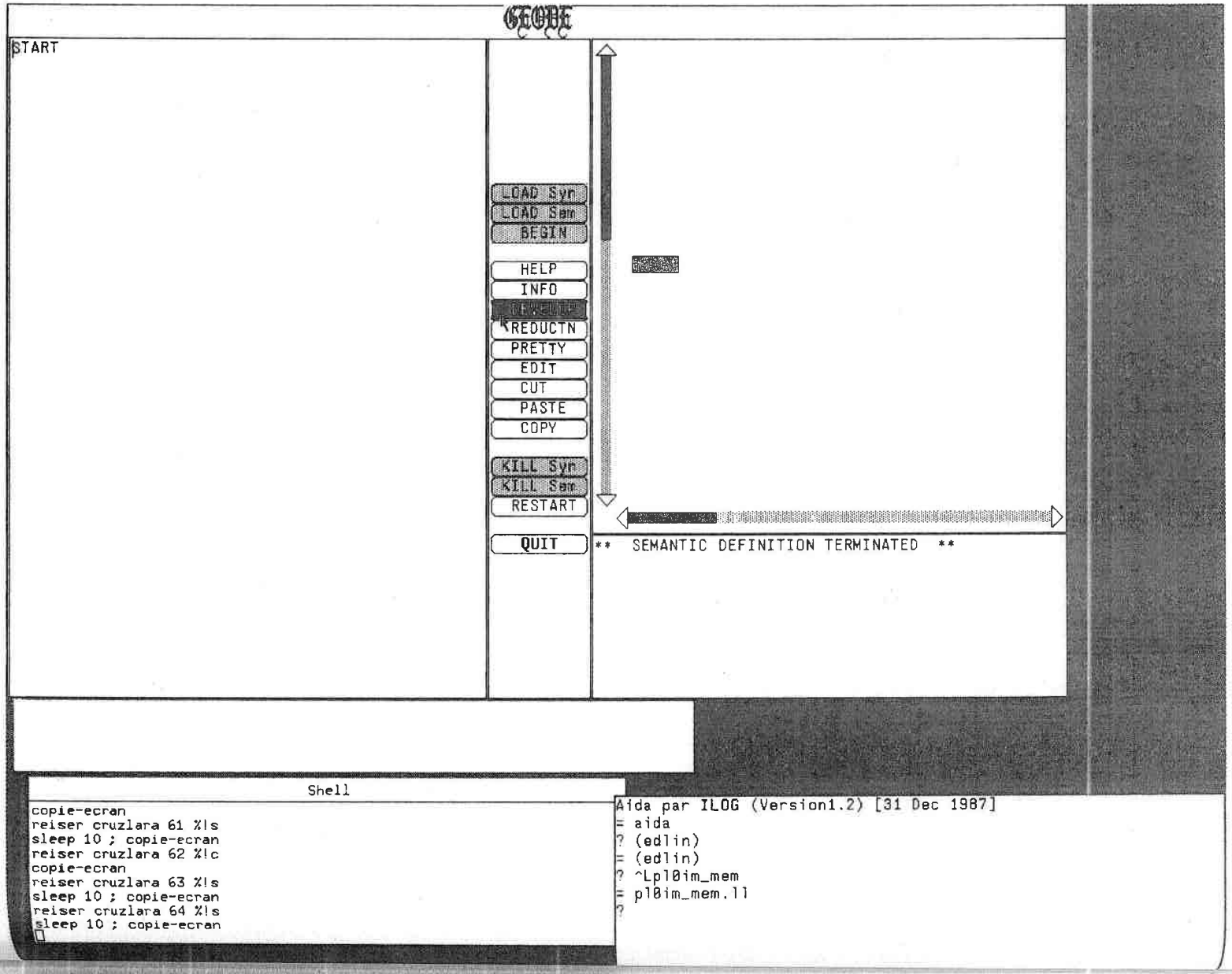

GLOBE
LOAD Syn
LOAD Sem
BEGIN
HELP
INFO
DEVELOP
REDUCTN
PRETTY
EDIT
CUT
PASTE
COPY
KILL Syn
KILL Sem
RESTART
QUIT
*** Semantic Definition ***
Tool which will be semantically defined: ? row
*** Synthesized Attributes Definition Terminated ***
*** Inherited Attributes Definition Terminated ***
*** Semantic Definition Terminated ***
*** Synthesized Attributes Functions Generated ***
*** Inherited Attributes Functions Generated ***
*** Synthesized Attributes Functions Charged ***
*** Inherited Attributes Functions Charged ***
Tool which will be semantically defined: ? newdecomp
Shell
sleep 10 ; copie-ecran
reiser cruzlara 55 %ls
sleep 10 ; copie-ecran
reiser cruzlara 56 %copie-ecran
reiser cruzlara 57 %ls
sleep 10 ; copie-ecran
reiser cruzlara 58 %copie-ecran
reiser cruzlara 59 %!c
copie-ecran
Aida par ILOG (Version1.2) [31 Dec 1987]
= aida
? (edlin)
= (edlin)
? ^Lpl0im_mem
= pl0im_mem.ll
?

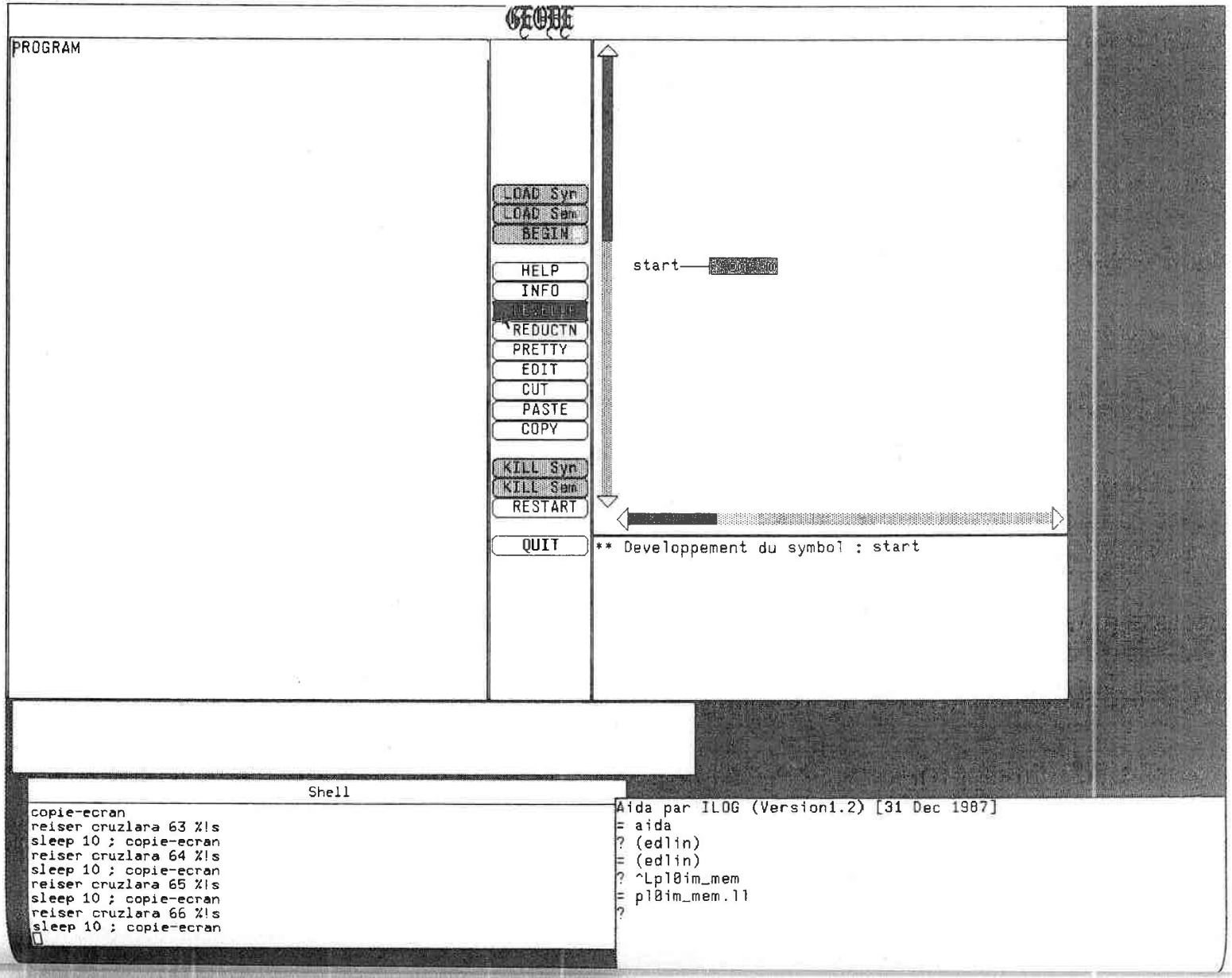

GLOBE
LOAD Syn
LOAD Sem
BEGIN
HELP
INFO
DEVELOP
REDUCTN
PRETTY
EDIT
CUT
PASTE
COPY
KILL Syn
KILL Sem
RESTART
QUIT
** SEMANTIC DEFINITION TERMINATED **
Shell
sleep 10 ; copie-ecran
reiser cruzlara 56 %copie-ecran
reiser cruzlara 57 %!s
sleep 10 ; copie-ecran
reiser cruzlara 58 %copie-ecran
reiser cruzlara 59 %!c
copie-ecran
reiser cruzlara 60 %!c
copie-ecran
Aida par ILOG (Version1.2) [31 Dec 1987]
= aida
? (edlin)
= (edlin)
? ^Lpl0im_mem
= pl0im_mem.ll
?

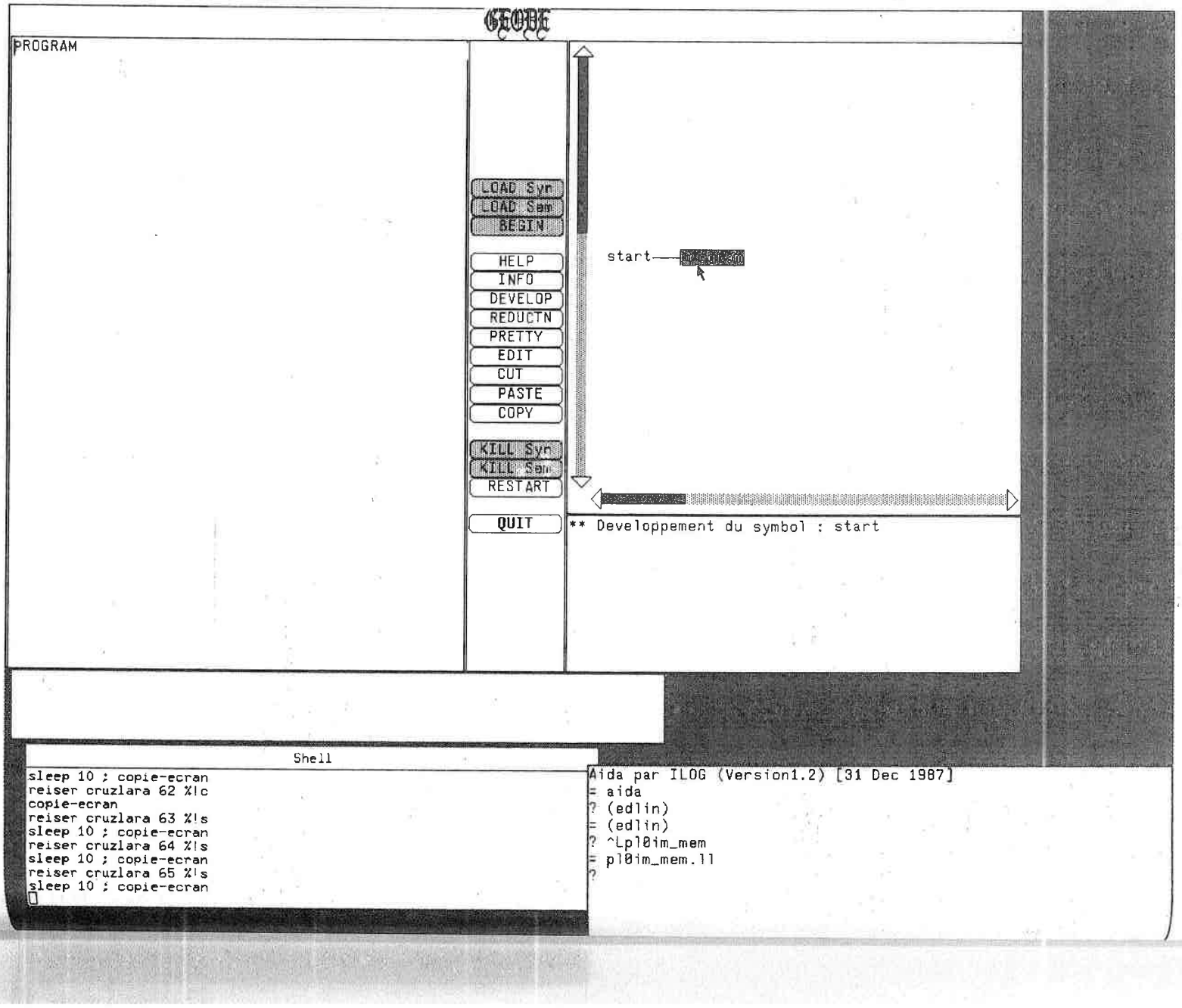

GLOBE
LOAD Syn
LOAD Sem
HELP
INFO
DEVELOP
REDUCTN
PRETTY
EDIT
CUT
PASTE
COPY
KILL Syn
KILL Sem
RESTART
QUIT
** SEMANTIC DEFINITION TERMINATED **
Shell
reiser cruzlara 57 %!s
sleep 10 ; copie-ecran
reiser cruzlara 58 %copie-ecran
reiser cruzlara 59 %!c
copie-ecran
reiser cruzlara 60 %!c
copie-ecran
reiser cruzlara 61 %!s
sleep 10 ; copie-ecran
Aida par ILOG (Version1.2) [31 Dec 1987]
= aida
? (edlin)
= (edlin)
? ^Lpl0im_mem
= pl0im_mem.ll
?

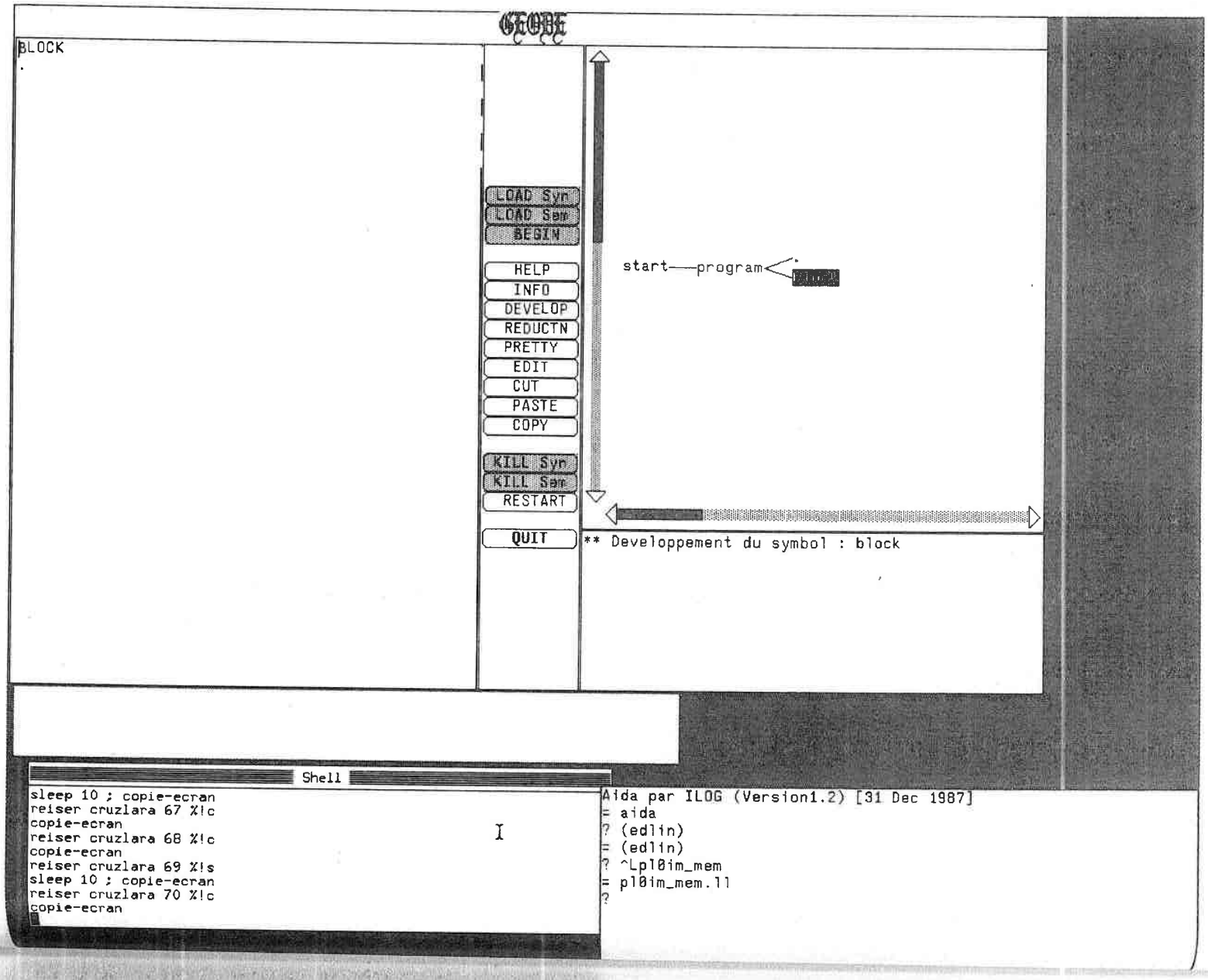

GEODE
START
start
LOAD Syn
LOAD Sem
BEGIN
HELP
INFO
DEVELOP
REDUCTN
PRETTY
EDIT
CUT
PASTE
COPY
KILL Syn
KILL Sem
RESTART
QUIT
** SEMANTIC DEFINITION TERMINATED **
Shell
reiser cruzlara 58 %copie-ecran
reiser cruzlara 59 %!c
copie-ecran
reiser cruzlara 60 %!c
copie-ecran
reiser cruzlara 61 %!s
sleep 10 ; copie-ecran
reiser cruzlara 62 %!c
copie-ecran
Aida par ILOG (Version1.2) [31 Dec 1987]
= aida
? (edlin)
= (edlin)
? ^Lpl0im_mem
= pl0im_mem.ll
?

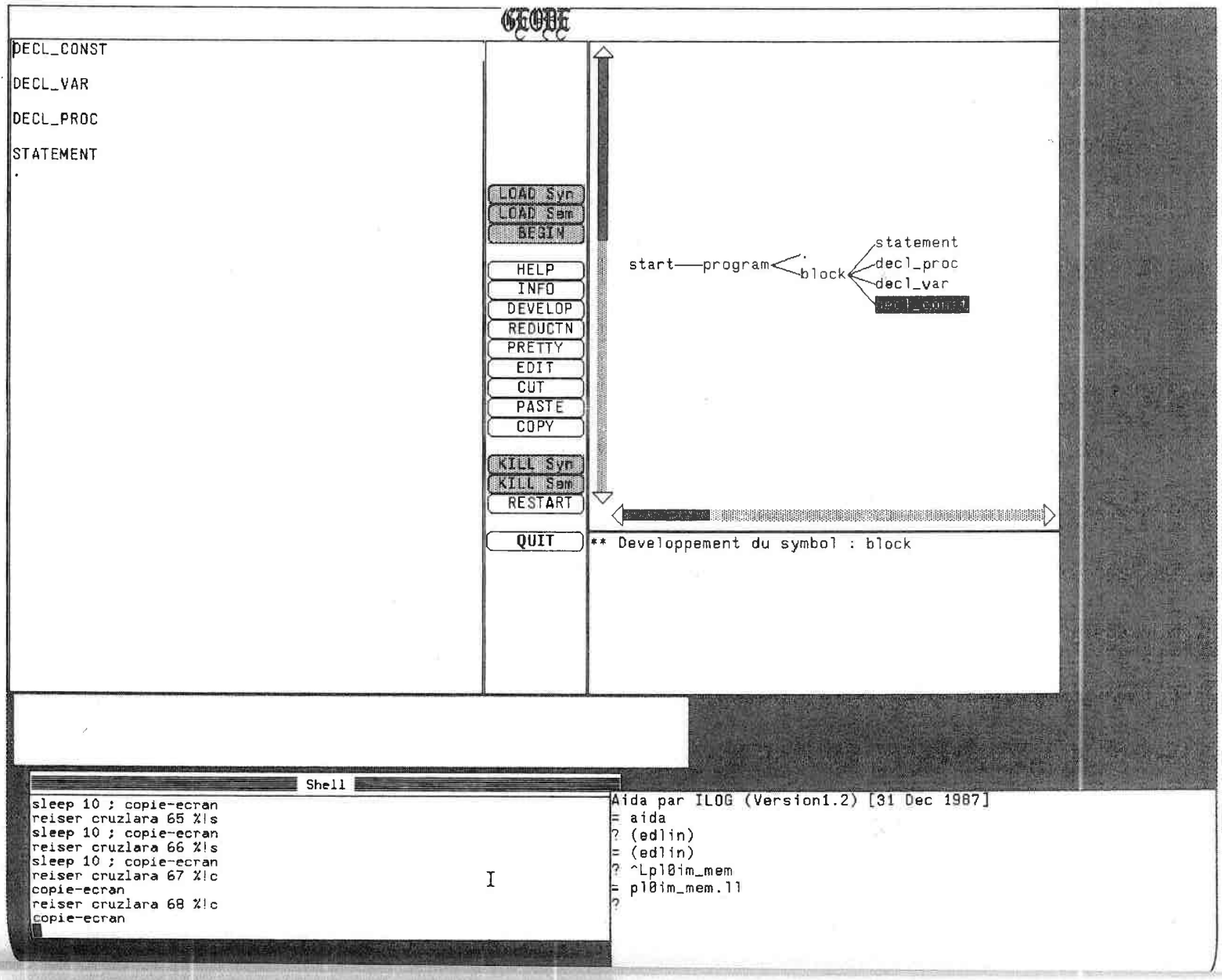

GLOBE
START
LOAD Syn
LOAD Sem
BEGIN
HELP
INFO
DEVELOP
REDUCTN
PRETTY
EDIT
CUT
PASTE
COPY
KILL Syn
KILL Sem
RESTART
QUIT
** SEMANTIC DEFINITION TERMINATED **
Shell
copie-ecran
reiser cruzlara 60 %!c
copie-ecran
reiser cruzlara 61 %!s
sleep 10 ; copie-ecran
reiser cruzlara 62 %!c
copie-ecran
reiser cruzlara 63 %!s
sleep 10 ; copie-ecran
Aida par ILOG (Version1.2) [31 Dec 1987]
= aida
? (edlin)
= (edlin)
? ^Lpl0im_mem
= pl0im_mem.ll
?

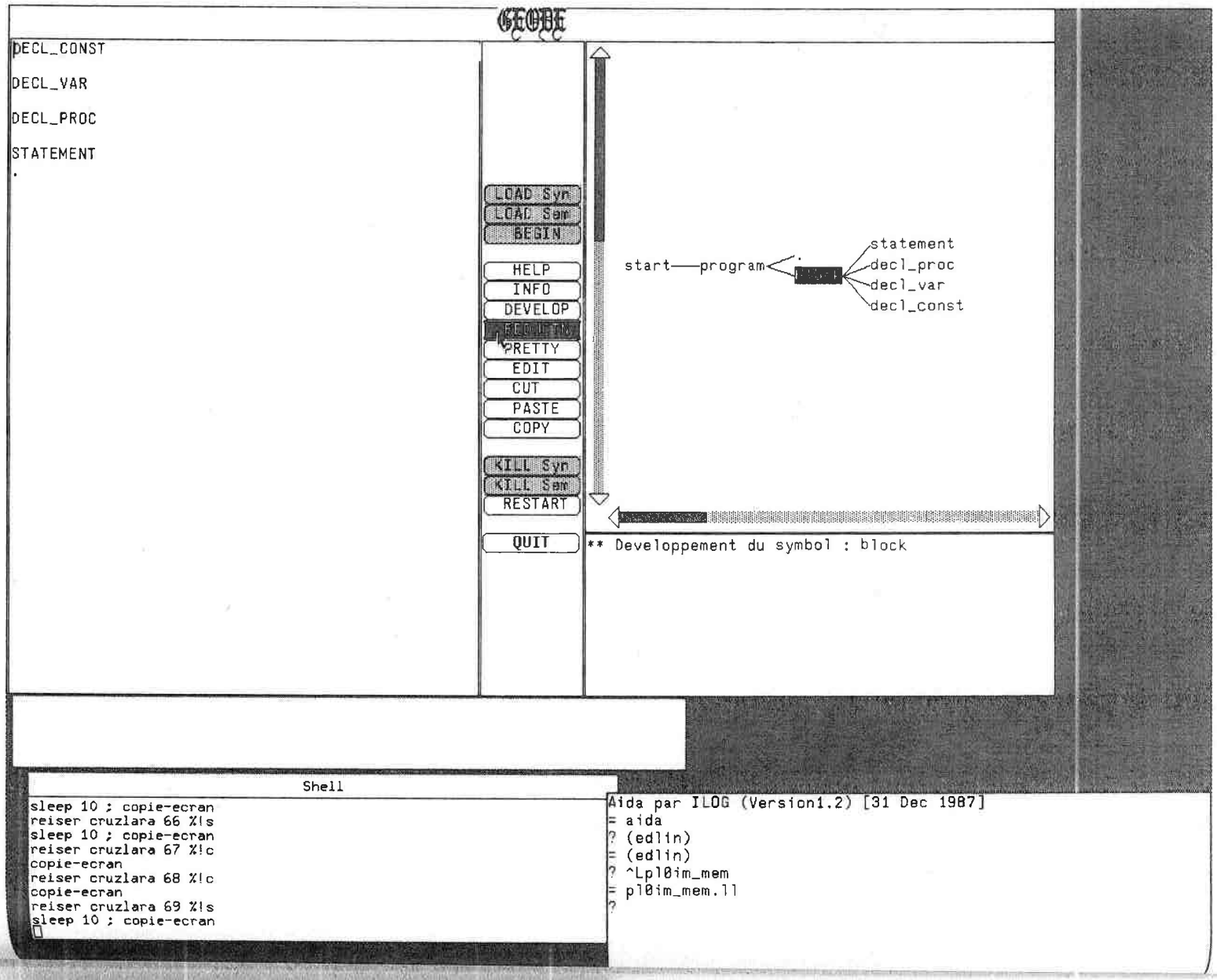

GEODE
START
LOAD Syn
LOAD Sem
BEGIN
HELP
INFO
REDUCTN
PRETTY
EDIT
CUT
PASTE
COPY
KILL Syn
KILL Sem
RESTART
QUIT
** SEMANTIC DEFINITION TERMINATED **
Shell
copie-ecran
reiser cruzlara 61 %!s
sleep 10 ; copie-ecran
reiser cruzlara 62 %!c
copie-ecran
reiser cruzlara 63 %!s
sleep 10 ; copie-ecran
reiser cruzlara 64 %!s
sleep 10 ; copie-ecran
Aida par ILOG (Version1.2) [31 Dec 1987]
= aida
? (edlin)
= (edlin)
? ^Lpl0im_mem
= pl0im_mem.ll
?

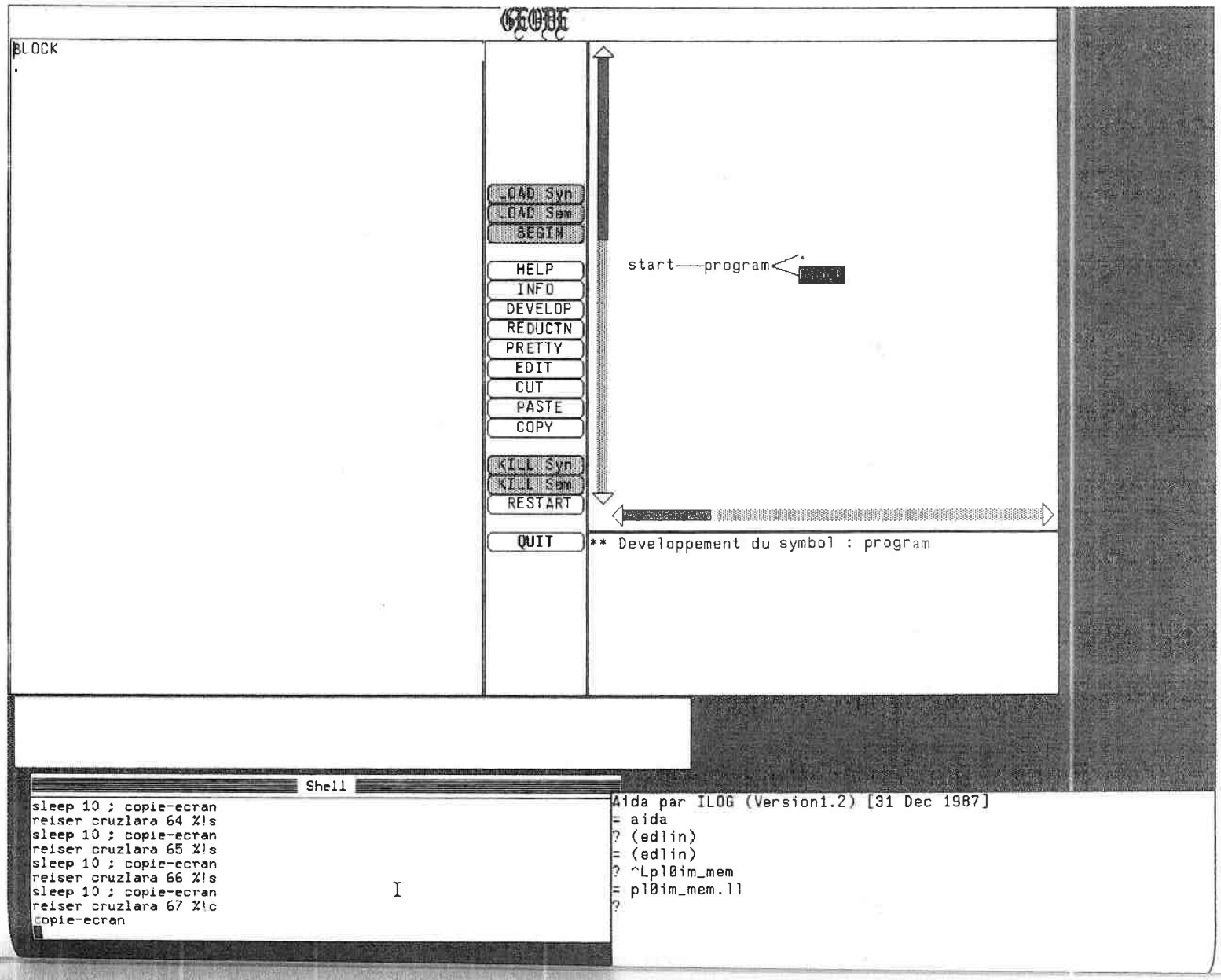

GEODE
PROGRAM
LOAD Syn
LOAD Sem
BEGIN
HELP
INFO
REDUCTN
PRETTY
EDIT
CUT
PASTE
COPY
KILL Syn
KILL Sem
RESTART
QUIT
start
** Developpement du symbol : start
Shell
copie-ecran
reiser cruzlara 63 %!s
sleep 10 ; copie-ecran
reiser cruzlara 64 %!s
sleep 10 ; copie-ecran
reiser cruzlara 65 %!s
sleep 10 ; copie-ecran
reiser cruzlara 66 %!s
sleep 10 ; copie-ecran
Aida par ILOG (Version1.2) [31 Dec 1987]
= aida
? (edlin)
= (edlin)
? ^Lpl0im_mem
= pl0im_mem.ll
?


GEODE
PROGRAM
LOAD Syn
LOAD Sem
BEGIN
HELP
INFO
DEVELOP
REDUCTN
PRETTY
EDIT
CUT
PASTE
COPY
KILL Syn
KILL Sem
RESTART
QUIT
start
** Developpement du symbol : start
Shell
sleep 10 ; copie-ecran
reiser cruzlara 62 %!c
copie-ecran
reiser cruzlara 63 %!s
sleep 10 ; copie-ecran
reiser cruzlara 64 %!s
sleep 10 ; copie-ecran
reiser cruzlara 65 %!s
sleep 10 ; copie-ecran
Aida par ILOG (Version1.2) [31 Dec 1987]
= aida
? (edlin)
= (edlin)
? ^Lpl0im_mem
= pl0im_mem.ll
?


GEODE
BLOCK
LOAD Syn
LOAD Sem
BEGIN
HELP
INFO
DEVELOP
REDUCTN
PRETTY
EDIT
CUT
PASTE
COPY
KILL Syn
KILL Sem
RESTART
QUIT
start—program
** Developpement du symbol : block
Shell
sleep 10 ; copie-ecran
reiser cruzlara 67 %!c
copie-ecran
reiser cruzlara 68 %!c
copie-ecran
reiser cruzlara 69 %!s
sleep 10 ; copie-ecran
reiser cruzlara 70 %!c
copie-ecran
Aida par ILOG (Version1.2) [31 Dec 1987]
= aida
? (edlin)
= (edlin)
? ^Lpl0im_mem
= pl0im_mem.ll
?


GEODE
DECL_CONST
DECL_VAR
DECL_PROC
STATEMENT
LOAD Syn
LOAD Sem
BEGIN
HELP
INFO
DEVELOP
REDUCTN
PRETTY
EDIT
CUT
PASTE
COPY
KILL Syn
KILL Sem
RESTART
QUIT
start
program
block
statement
decl_proc
decl_var
** Developpement du symbol : block
Shell
sleep 10 ; copie-ecran
reiser cruzlara 65 %!s
sleep 10 ; copie-ecran
reiser cruzlara 66 %!s
sleep 10 ; copie-ecran
reiser cruzlara 67 %!c
copie-ecran
reiser cruzlara 68 %!c
copie-ecran
Aida par ILOG (Version1.2) [31 Dec 1987]
= aida
? (edlin)
= (edlin)
? ^Lpl0im_mem
= pl0im_mem.ll
?


GEODE
DECL_CONST
DECL_VAR
DECL_PROC
STATEMENT
.
LOAD Syn
LOAD Sem
BEGIN
HELP
INFO
DEVELOP
EDIT
PRETTY
EDIT
CUT
PASTE
COPY
KILL Syn
KILL Sem
RESTART
QUIT
start
program
statement
decl_proc
decl_var
decl_const
** Developpement du symbol : block
Shell
sleep 10 ; copie-ecran
reiser cruzlara 66 %!s
sleep 10 ; copie-ecran
reiser cruzlara 67 %!c
copie-ecran
reiser cruzlara 68 %!c
copie-ecran
reiser cruzlara 69 %!s
sleep 10 ; copie-ecran
Aida par ILOG (Version1.2) [31 Dec 1987]
= aida
? (edlin)
= (edlin)
? ^Lpl0im_mem
= pl0im_mem.ll
?


GEODE
BLOCK
LOAD Syn
LOAD Sem
BEGIN
HELP
INFO
DEVELOP
REDUCTN
PRETTY
EDIT
CUT
PASTE
COPY
KILL Syn
KILL Sem
RESTART
QUIT
start——program
** Developpement du symbol : program
Shell
sleep 10 ; copie-ecran
reiser cruzlara 64 %!s
sleep 10 ; copie-ecran
reiser cruzlara 65 %!s
sleep 10 ; copie-ecran
reiser cruzlara 66 %!s
sleep 10 ; copie-ecran
reiser cruzlara 67 %!c
copie-ecran
Aida par ILOG (Version1.2) [31 Dec 1987]
= aida
? (edlin)
= (edlin)
? ^Lpl0im_mem
= pl0im_mem.ll
?

Annexe 9.

LA DEFINITION LE_LISP DE LA SYNTAXE DU LANGAGE PL0.

```
({create_grammar} pl0_grammar)

({create_start_symbol} pl0_grammar 'start)

({create_generic_terminal} pl0_grammar "ident" 'fonction_id)
({create_generic_terminal} pl0_grammar "number" 'fonction_num)

({define_synonime} pl0_grammar "point" ".")
({define_synonime} pl0_grammar "semicolon" ";")
({define_synonime} pl0_grammar "equal" "=")
({define_synonime} pl0_grammar "comma" ",")
({define_synonime} pl0_grammar "assign" ":=")
({define_synonime} pl0_grammar "not_equal" "<>")
({define_synonime} pl0_grammar "less" "<")
({define_synonime} pl0_grammar "greater" ">")
({define_synonime} pl0_grammar "less_equal" "<=")
({define_synonime} pl0_grammar "greater_equal" ">=")
({define_synonime} pl0_grammar "plus" "+")
({define_synonime} pl0_grammar "minus" "-")
({define_synonime} pl0_grammar "pleft" "(")
({define_synonime} pl0_grammar "pright" ")")
({define_synonime} pl0_grammar "multiply" "*")
({define_synonime} pl0_grammar "divide" "/")

({define_syntax} pl0_grammar
        '(
          (start                program)

          (program              block
                                "point")

          (block                decl_const
                                decl_var
                                decl_proc
                                statement)

          (decl_const           "CONST"
                                decl_const_list
                                "semicolon")
```

```
(decl_const          "empty")

(decl_const_list     "ident"
                     "equal"
                     "number")

(decl_const_list     "ident"
                     "equal"
                     "number"
                     "comma"
                     decl_const_list)

(decl_var            "VAR"
                     decl_var_list
                     "semicolon")

(decl_var            "empty")

(decl_var_list       "ident")

(decl_var_list       "ident"
                     "comma"
                     decl_var_list)

(decl_proc           proc_list
                     "semicolon"
                     decl_proc)

(decl_proc           "empty")

(proc_list           "PROCEDURE"
                     decl_proc_list)

(decl_proc_list      "ident"
                     "semicolon"
                     block)

(statement           "ident"
                     "assign"
                     expression)

(statement           "CALL"
                     "ident")

(statement           "BEGIN"
                     statement_list
                     "END")

(statement           "IF"
```

```
                        condition
                        "THEN"
                        statement)

(statement              "WHILE"
                        condition
                        "DO"
                        statement)

(statement              "empty")

(statement_list         statement)

(statement_list         statement
                        "semicolon"
                        statement_list)

(condition              "ODD"
                        expression)

(condition              expression
                        cond_operators
                        expression)

(cond_operators         "equal")

(cond_operators         "not_equal")

(cond_operators         "less")

(cond_operators         "greater")

(cond_operators         "less_equal")

(cond_operators         "greater_equal")

(expression             plus_minus_or_empty
                        term_list)

(plus_minus_or_empty    "plus")

(plus_minus_or_empty    "minus")

(plus_minus_or_empty    "empty")

(term_list              term)

(term_list              term
                        add_or_sust
                        term_list)
```

```
        (add_or_sust        "plus")
        (add_or_sust        minus)
        (term               factor_list)
        (factor_list        factor)
        (factor_list        factor
                            mult_or_div
                            factor_list)
        (factor             "ident")
        (factor             "number")
        (factor             "pleft"
                            expression
                            "pright")
        (mult_or_div        "multiply")
        (mult_or_div        "divide")
    )
)
```

Annexe 10.

LA DEFINITION LE_LISP DU DECOMPILATEUR DU LANGAGE PL0.

```
({define_synt_attributes} pl0_grammar 'decomp
        '(
           (start                  rep)
           (program                rep)
           (block                  rep)
           (decl_const             rep)
           (decl_const_list        rep)
           (decl_var               rep)
           (decl_var_list          rep)
           (decl_proc              rep)
           (proc_list              rep)
           (decl_proc_list         rep)
           (statement              rep)
           (expression             rep)
           (plus_minus_or_empty    rep)
           (term_list              rep)
           (term                   rep)
           (factor_list            rep)
           (factor                 rep)
           (mult_or_div            rep)
           (add_or_sust            rep)
           (statement_list         rep)
           (condition              rep)
           (cond_operators         rep)
           (ident                  rep)
           (number                 rep)
        )

)

({define_inhe_attributes pl0_grammar 'decomp
        '(
           (program                tab)
           (block                  tab)
           (decl_const             tab)
           (decl_var               tab)
           (decl_proc              tab)
           (statement              tab)
           (decl_const_list        tab)
```

```
            (decl_var_list          tab)
            (proc_list              tab)
            (decl_proc_list         tab)
            (statement_list         tab)
        )

)

(   (1      (setatt   start.rep
                      (display "START")))

    (2      (setatt   start.rep
                      program.rep)
            (setatt   program.tab
                      0))

    (3      (setatt   program.rep
                      (display (program.tab  "PROGRAM"))))

    (4      (setatt   program.rep
                      (+ block.tab  block.rep  (display #\cr)  point.rep))
            (setatt   point.rep
                      (display #\cr))
            (setatt   block.tab
                      program.tab))

    (5      (setatt   block.rep
                      (display  block.tab  "BLOCK")))

    (6      (setatt   block.rep
                      (+  decl_const.tab  decl_const.rep  (display #\cr)  decl_var.tab
                          decl_var.rep  (display #\cr)  decl_proc.tab  decl_proc.rep
                          (display #\cr)  statement.tab  statement.rep))
            (setatt   decl_const.tab
                      block.tab)
            (setatt   decl_var.tab
                      block.tab)
            (setatt   decl_proc.tab
                      block.tab)
            (setatt   statement.tab
                      block.tab))

    (7      (setatt   decl_const.rep
                      (display  decl_const.tab  "DECL_CONST")))

    (8      (setatt   decl_const.rep
                      (+  decl_const.tab  CONST.rep  (display #\cr)  decl_const_list.rep
                          decl_const_list.tab  semicolon.rep))
            (setatt   CONST.rep
                      (display  decl_const.tab  "CONST"))
```

```
        (setatt  semicolon.rep
                 (display ";"))
        (setatt  decl_const_list.tab
                 (+ 1 decl_const_list.tab)))

(9      (setatt  decl_const.rep
                 0))

(10     (setatt  decl_const_list.rep
                 (display decl_const_list.tab "DECL_CONST_LIST")))

(11     (setatt  decl_const_list.rep
                 (+ decl_const_list.tab ident.rep equal.rep number.rep))
        (setatt  ident.rep
                 (display decl_const_list.tab ident.str))
        (setatt  ident.str
                 (fonction_id))
        (setatt  equal.rep
                 (display "="))
        (setatt  number.rep
                 (dispaly number.str))
        (setatt  number.str
                 (fonction_num)))

(12     (setatt  decl_const_list@1.rep
                 (+ decl_const_list@1.tab ident.rep equal.rep number.rep comma.rep
                    (display #\cr) decl_const_list@2.tab decl_const_list@2.rep))
        (setatt  decl_const_list@2.tab
                 decl_const_list@1.tab)
        (setatt  ident.rep
                 (display decl_const_list@1.tab ident.str))
        (setatt  ident.str
                 (fonction_id))
        (setatt  equal.rep
                 (display "="))
        (setatt  number.rep
                 (display number.str))
        (setatt  number.str
                 (fonction_num))
        (setatt  comma.rep
                 (display ",")))

(13     (setatt  decl_var.rep
                 (display decl_var.tab "DECL_VAR")))

(14     (setatt  decl_var.rep
                 (+ decl_var.tab VAR.rep (display #\cr) decl_var_list.rep
                    decl_var_list.tab semicolon.rep))
        (setatt  VAR.rep
                 (display decl_var.tab "VAR"))
```

```
        (setatt  semicolon.rep
                 (display ";"))
        (setatt  decl_var_list.tab
                 (+ 1 decl_var.tab)))

(15     (setatt  decl_var.rep
                 0))

(16     (setatt  decl_var_list.rep
                 (display decl_var_list.tab "DECL_VAR_LIST")))

(17     (setatt  decl_var_list.rep
                 (+ ident.rep decl_var_list.tab))
        (setatt  ident.rep
                 (display decl_var_list.tab ident.str))
        (setatt  ident.str
                 (fonction_id)))

(18     (setatt  decl_var_list@1.rep
                 (+  decl_var_list@1.tab ident.rep comma.rep (display #\cr)
                     decl_var_list@2.tab decl_var_list@2.rep))
        (setatt  decl_var_list@2.tab
                 decl_var_list@1.tab)
        (setatt  ident.rep
                 (display decl_var_list@1.tab ident.str))
        (setatt  ident.str
                 (fonction_id))
        (setatt  comma.rep
                 (display ",")))

(19     (setatt  decl_proc.rep
                 (display decl_proc.tab "DECL_PROC")))

(20     (setatt  decl_proc@1.rep
                 (+  proc_list.tab proc_list.rep semicolon.rep (display #\cr)
                     decl_proc@2.tab decl_proc@2.rep))
        (setatt  semicolon.rep
                 (display ";"))
        (setatt  proc_list.tab
                 decl_proc@1.tab)
        (setatt  decl_proc@2.tab
                 (+  1 decl_proc@1.tab)))

(21     (setatt  decl_proc.rep
                 0))

(22     (setatt  proc_list.rep
                 (display proc_list.tab "PROC_LIST")))

(23     (setatt  proc_list.rep
```

```
                (+  proc_list.tab PROCEDURE.rep decl_proc_list.rep))

        (setatt PROCEDURE.rep
                (display proc_list.tab "PROCEDURE"))
        (setatt decl_proc_list.tab
                proc_list.tab))

(24     (setatt decl_proc_list.rep
                (display decl_proc_list.tab "DECL_PROC_LIST")))

(25     (setatt decl_proc_list.rep
                (+  decl_proc_list.tab ident.rep semicolon.rep (display #\cr)
                    block.tab block.rep))
        (setatt ident.rep
                (display decl_proc_list.tab ident.str))
        (setatt ident.str
                (fonction_id))
        (setatt semicolon.rep
                (display ";"))
        (setatt block.tab
                (+  1 decl_proc_list.tab)))

(26     (setatt statement.rep
                (display statement.tab "STATEMENT")))

(27     (setatt statement.rep
                (+  statement.tab ident.rep assign.rep expression.rep))
        (setatt assign.rep
                (display ":="))
        (setatt ident.rep
                (display statement.tab ident.str))
        (setatt ident.str
                (fonction_id)))

(28     (setatt statement.rep
                (+  statement.tab CALL.rep ident.rep))
        (setatt CALL.rep
                (display statement.tab "CALL"))
        (setatt ident.rep
                (display ident.str))
        (setatt ident.str
                (fonction_id)))

(29     (setatt statement.rep
                (+  statement.tab BEGIN.rep (display #\cr) statement_list.tab
                    statement_list.rep (display #\cr) statement.tab END.rep))
        (setatt BEGIN.rep
                (display statement.tab "BEGIN"))
        (setatt END.rep
                (display statement.tab "END"))
```

```
        (setatt  statement_list.tab
                 (+  1  statement.tab)))

(30     (setatt  statement@1.rep
                 (+  statement@1.tab IF.rep condition.rep THEN.rep (display #\cr)
                     statement@2.tab statement@2.rep))
        (setatt  IF.rep
                 (display statement@1.tab "IF"))
        (setatt  THEN.rep
                 (display "THEN"))
        (setatt  statement@2.tab
                 (+  1  statement@1.tab)))

(31     (setatt  statement@1.rep
                 (+  statement@1.tab WHILE.rep condition.rep DO.rep (display #\cr)
                     statement@2.tab statement@2.rep))
        (setatt  WHILE.rep
                 (display statement@1.tab "WHILE"))
        (setatt  DO.rep
                 (display "DO"))
        (setatt  statement@2.tab
                 (+  1  statement@1.tab)))

(32     (setatt  statement.rep
                 0))

(33     (setatt  statement_list.rep
                 (display statement_list.tab "STATEMENT_LIST")))

(34     (setatt  statement_list.rep
                 (+  statement.tab statement.rep))
        (setatt  statement.tab
                 statement_list.tab))

(35     (setatt  statement_list@1.rep
                 (+  statement.tab statement.rep semicolon.rep (display #\cr)
                     statement_list@2.tab statement_list@2.rep))
        (setatt  semicolon.rep
                 (display ";"))
        (setatt  statement.tab
                 statement_list@1.tab)
        (setatt  statement@2.tab
                 statement_list@1.tab))

(36     (setatt  condition.rep
                 (display "CONDITION")))

(37     (setatt  condition.rep
                 (+  condition.tab ODD.rep expression.rep))
        (setatt  ODD.rep
```

```
                (display "ODD")))

(38     (setatt condition.rep
                (+ expression@1.tab expression@1.rep cond_operators.rep
                   expression@2.rep)))

(39     (setatt cond_operators.rep
                (display "COND_OPERATORS")))

(40     (setatt cond_operators.rep
                equal.rep)
        (setatt equal.rep
                (display "=")))

(41     (setatt cond_operators.rep
                not_equal.rep)
        (setatt not_equal.rep
                (display "<>")))

(42     (setatt cond_operators.rep
                less.rep)
        (setatt less.rep
                (display "<")))

(43     (setatt cond_operators.rep
                greater.rep)
        (setatt greater.rep
                (display ">")))

(44     (setatt cond_operators.rep
                less_equal.rep)
        (setatt less_equal.rep
                (display "<=")))

(45     (setatt cond_operators.rep
                greater_equal.rep)
        (setatt greater_equal.rep
                (display ">=")))

(46     (setatt expression.rep
                (display "EXPRESSION")))

(47     (setatt expression.rep
                (+ plus_minus_or_empty.rep term_list.rep)))

(48     (setatt plus_minus_or_empty.rep
                (display "PLUS_MINUS_OR_EMPTY")))

(49     (setatt plus_minus_or_empty.rep
                plus.rep)
```

```
        (setatt  plus.rep
                 (display  "+")))

(50     (setatt  plus_minus_or_empty.rep
                 minus.rep)
        (setatt  minus.rep
                 (display  "-")))

(51     (setatt  plus_minus_or_empty.rep
                 0))

(52     (setatt  term_list.rep
                 (display  "TERM_LIST")))

(53     (setatt  term_list.rep
                 term.rep))

(54     (setatt  term_list@1.rep
                 (+  term.rep  add_or_sust.rep  term_list@2.rep)))

(55     (setatt  add_or_sust.rep
                 (display  "ADD_OR_SUST")))

(56     (setatt  add_or_sust.rep
                 plus.rep)
        (setatt  plus.rep
                 (display  "+")))

(57     (setatt  add_or_sust.rep
                 minus.rep)
        (setatt  minus.rep
                 (display  "-")))

(58     (setatt  term.rep
                 (display  "TERM")))

(59     (setatt  term.rep
                 factor_list.rep))

(60     (setatt  factor_list.rep
                 (display  "FACTOR_LIST")))

(61     (setatt  factor_list.rep
                 factor.rep))

(62     (setatt  factor_list@1.rep
                 (+  factor.rep  mult_or_div.rep  factor_list@2.rep)))

(63     (setatt  factor.rep
                 (display  "FACTOR")))
```

```
(64     (setatt  factor.rep
                 ident.rep)
        (setatt  ident.rep
                 (display  ident.str))
        (setatt  ident.str
                 (fonction_id)))

(65     (setatt  factor.rep
                 number.rep)
        (setatt  number.rep
                 (display  number.str))
        (setatt  number.str
                 (fonction_num)))

(66     (setatt  factor.rep
                 (+  pleft.rep expression.rep pright.rep))
        (setatt  pleft.rep
                 (display  "("))
        (setatt  pright.rep
                 (display  ")")))

(67     (setatt  mult_or_div.rep
                 (display  "MULT_OR_DIV")))

(68     (setatt  mult_or_div.rep
                 multiply.rep)
        (setatt  multiply.rep
                 (display  "*")))

(69     (setatt  mult_or_div.rep
                 divide.rep)
        (setatt  divide.rep
                 (display  "/")))
)
```

INPL
INSTITUT NATIONAL POLYTECHNIQUE DE LORRAINE

AUTORISATION DE SOUTENANCE DE THESE
DU DOCTORAT DE L'INSTITUT NATIONAL POLYTECHNIQUE DE LORRAINE

VU LES RAPPORTS ETABLIS PAR :
Monsieur PAIR Claude, Professeur CRIN/INPL,
Monsieur LANG Bernard, Chercheur INRIA.

Le Président de l'Institut National Polytechnique de Lorraine,
autorise :

Monsieur CRUZ LARA SILVA Samuel

à soutenir devant l'INSTITUT NATIONAL POLYTECHNIQUE DE LORRAINE, une thèse intitulée :

"GEODE : un système pour la génération d'environnements de programmation intégrés"

en vue de l'obtention du titre de :

DOCTEUR DE L'INSTITUT NATIONAL POLYTECHNIQUE DE LORRAINE

Spécialité : **"Informatique"**

Fait à Vandoeuvre le, **9 Novembre 1988**
Le Président de l'I.N.P.L.

M.GANTOIS

2, avenue de la Forêt de Haye - B.P. 3 - 54501 VANDŒUVRE CEDEX
Téléphone : 83. 57. 48. 48 - Télex : 961 715 F - Télécopie : 83. 57. 49. 55