CIGALE: A TOOL FOR INTERACTIVE GRAMMAR CONSTRUCTION AND EXPRESSION PARSING

Frédéric VOISIN
L.R.I., Bâtiment 490, Université Paris-Sud, F-91405 Orsay Cedex, France

Communicated by M.C. Gaudel
Received January 1985
Revised November 1985

Abstract. In this paper we present CIGALE, a system for incremental grammar construction and expressions parsing. It is designed to cope with the problem of parsing in environments which require an incremental definition of languages and a great flexibility in syntax. Its main application, parsing in an environment for Abstract Data Types specification, is presented and a comparison with other parsing systems which may be used for such purpose is undertaken.

Introduction

Much work related to programming is devoted to research on Abstract Data Types (ADT) [3, 17]. This approach, which intends to formalize the usual notion of data types, seems to be one way of easing the construction of correct programs, and of enforcing properties such as modularity, reusability and separation between specification and realization.

Most of these works are theoretical and attempt to give a precise description of what an ADT is, to study fundamental concepts such as parameterization [12, 24] or handling of errors or exceptions [3, 4]. But it is now agreed on the fact that specification languages and methods must be accompanied by supporting tools. For example to compare specification languages or to test new ones it is usually necessary to write a lot of ADT specifications with non-trivial complexity or length. An efficient support may be gained by the use of an algebraic specification environment, integrating tools to specify, test and use ADT in programming applications. Examples of such environments are AFFIRM, OBJ or LARCH. The CIGALE [26] system which we present in this paper is part of such an environment, named ASSPEGIQUE [5].

CIGALE deals with ADT considered from a syntactical (parsing) point of view. Its main goal is not only to provide parsing of expressions in this specific context, but also to assist the programmer while (s)he is writing data type specifications. CIGALE tries to free the user from as many syntactical constraints as possible, to let him/her concentrate on the semantical task of designing specifications. From our specific point of view, specifying ADT is considered as defining grammars and
languages. The main capabilities we offer are
- real simplicity for the definition of the operators of an ADT and parsing of
  expressions, by an incremental and interactive construction of these languages;
- handling of types modularity;
- support of specific notions of ADT such as coercion and overloading of operators;
- a flexible and user-oriented way of defining operators.
Section 1 contains an overview of CIGALE and of the problem of parsing in an
ADT environment. Section 2 presents existing methods. In Section 3 and 4 CIGALE
is more fully described and we give a more detailed example.

1. An overview of CIGALE

As mentioned above, CIGALE is part of an algebraic specification environment,
called ASSPEGIQUE. It is important to notice that a parsing system is a significant
part of such an environment, as its capabilities influence those of the whole system:
every feature which is not also handled at this basic level cannot be efficiently
supported by the other modules of the environment.

In the design of ASSPEGIQUE, special care has been focused on the problem
of flexibility for the design of specifications, and on the crucial problem of modularity
and reusability of specifications. As we shall see, these facilities have parsing
counterparts that must be dealt with, and which are unusual w.r.t. those of conven-
tional languages.

Flexibility is related to the ease of writing specifications. This means that the
environment should be adapted to different specification development methods
(bottom-up, top-down or mixed methods . . . ) or even different formalisms. Flexibility
must be present everywhere in the environment, and its main implication for parsing
is to leave a great freedom in the choice of operators, and to offer a convenient way
of using specifications (cf. Section 1.2).

Modularity and reusability are as important for algebraic specifications as they
are for programming if one wants to face fairly large examples. They mainly depend
on the semantic operators available for structuring specifications [7], but also on the
way existing specifications can later be reused in the environment [6]
(cf. Section 1.3).

ADT as grammars and languages: an intuitive approach to ADT is to view each
specification as an enumeration of operators (its signature, or syntactical part) and
the relations describing their mutual interactions, as well as the connection with
operators of already defined specifications (the axioms, or semantical part).

In such a specification, each operator is described by its domain, its range and
its syntax: its own way of combining operands and symbols. The definition of an
operator is thus similar to a production in a context-free grammar. To each
specification corresponds a grammar, and we can define the language associated
with a specification to be the set of all strings composed of valid (w.r.t. the signature)
combinations of operators from the specification or from the languages associated with the required specifications.

1.1. Syntax of the operators

One way of offering simplicity of definition and use, and therefore a first step towards the legibility and correctness of data type specifications, consists in leaving a great flexibility in the choice of the operator's syntax. It should be possible to use all the classical syntaxes (infixed, postfixed, prefixed) for standard operators, as well as to define new ones for other operators, if they are more expressive or less error-prone.

The syntax for operators in CIGALE has been derived from the one proposed by Goguen in the OBJ system [16]: it is composed of symbols and of occurrences of the reserved symbol \( '_' \) (we shall name it \textit{placeholder}) which stands for the position of an operand. All combinations of strings and placeholders are allowed, provided that there are as many name of sorts\(^1\) in the domain of the operator as placeholders in the syntax. Opposite to OBJ, the functional syntax (prefixed, with parentheses and commas) is not supported.

As example, in Fig. 1 we present one of the ways (the most simple and intuitive!) of defining in CIGALE specifications for \textit{integer} and \textit{polynomial} ADT, and some operators for a specification of the (homogeneous and flat) \textit{list} ADT.

Two operators in the \textit{polynomial} specification have a syntax composed of a single placeholder. The first one gives an abbreviation for 0-degree monomials; the second one is used to stop the 'recursion' in the + operator. Such operators (coercions) are specifically discussed in Section 1.4. The \textit{list} example illustrates the possibility to have operators without visible call.

Languages generated by a set of operators \( \Theta \) may be formally defined as follows:

1. Let \( s \) be a sort, the language \( L(\Theta, s) \) is minimally constructed by
   - if \( (a_1 \ldots a_n : s \rightarrow s) \) is in \( \Theta \), then the string \( a_1 \ldots a_n \) is in \( L(\Theta, s) \),
   - if \( (a_1 \ldots a_n : s_1 \rightarrow s_1 \ldots s_p \rightarrow s) \) is in \( \Theta \), then for each \( (x_1, \ldots, x_p) \) in \( L(\Theta, s_1) \times \cdots \times L(\Theta, s_p) \), the string \( a_1 \ldots a_n[x_1/_{-i} \ldots x_p/_{-p}] \) is in \( L(\Theta, s) \), where \( _{-i} \) stands for the \( i \)th occurrence of \( '_' \) in the syntax of the operator, and \( t[u_1/v_1 \ldots u_p/v_p] \) is the string obtained by substituting each term \( u_i \) for the symbol \( v_i \) in \( t \).

2. \( L(\Theta) = \bigcup L(\Theta, s) \), over all sorts \( s \) occurring in the signature of an operator in \( \Theta \).

The AXIOMS parts in the previous specifications give examples of valid sentences.

As one can notice in the \textit{polynomial} and \textit{integer} specifications there may be several operators with the same syntax but different arities. To refer unambiguously to an operator, we shall thus use the couple (syntax, arity) as the \textit{name} of an operator. The parsing problem consists therefore in discovering, given an input, if it represents

\(^1\) Following the ADT literature, we use the word \textit{sort}, rather than \textit{type}, to represent names of domains.
a valid combination (and which combination) of operators. The output may be viewed as a tree whose nodes are the names of the operators involved. The (possible) use of coercions in such a combination is given explicitly in the tree, as coercions are only special cases of operators in the previous definition.

Moreover, if the user is not constrained by CIGALE for the syntactic form of an operator, neither is (s)he limited by restrictions due to parsing considerations. Some parsing methods (LL-like) would refuse, for example, left-recursive rules (e.g. the + given for polynomial), while others (LR-like) would require global properties of the set of operators [1]. With such methods the intuitive way of expressing operators has sometimes to be abandoned, and operators have to be rewritten in a different and more obscure style before to be accepted by the system.

It is our claim that, in the framework we consider, no syntactical constructions must be forbidden, even if they lead to inefficiencies at parsing time . . . .
1.2. Incremental construction of the language

The second constraint of flexibility for the parsing system is related to the ease for testing specifications. The design of specifications is not an easy and straightforward activity: before freezing a specification, one usually wants to compare different 'drafts', with different sets of operators and axioms.

An example of specifications for which the set of operators needed is not easy to find is those of specifications with so-called hidden\(^2\) operators [22]. The need for hidden operators in the design of a specification may not be clear at the beginning!

Another way of modifying the kernel of operators is by a change in the syntax for operators. There are usually several possible syntaxes for a given operator, differing for example by the associativity or precedence they induce.... Axioms may be more or less easy to express, depending on the syntax used for the operator and it is not obvious to find the right syntax at once!

By writing axioms, or by evaluating expressions via a symbolic evaluator, one implicitly uses the language defined by these operators. As the signature of a signature is an evolving object, through the list of operators, or through their syntax, CIGALE allows an incremental and interactive construction of the language. The addition or deletion of operators is always possible and straightforward for the user, while parsing is available at any time to use the language defined.

1.3. Modularity

Another point related to the previous one is the ability to support the modularity, which is characteristic of ADT. We have seen that with each specification is associated a grammar and a language. Grammars are usually defined independently of each other, and a specification database is, in part, a collection of grammars. The languages generated by these grammars may not be disjoint, if there is some sharing of required specifications.

The collections of languages used to specify the dictionary data type could for example be the following:

\[
\begin{align*}
\text{dictionary} &= \text{sorted-array}[\text{word}] \\
\text{sorted-array}[\text{sorted-thing}] \\
\text{integer} \\
\text{word} &= \text{sequence}[\text{char}] \\
\text{sorted-thing} \\
\text{char} \\
\text{sequence}[X] \\
\text{X} \\
&= \text{sequence}[X] \\
\text{boolean}\end{align*}
\]

\(^2\) Operators are hidden, if they must not be visible outside the specification in which they are introduced, but are only used to make the design of axioms easier. Such operators cannot be called explicitly by future users of the specification.
where $\Rightarrow$ means "uses", $\leftarrow$ means "is an instance of... using..." and $\Rightarrow$ means "is parameterized by".

This is an example of a situation in which \textit{a priori} independent specifications are mixed to build a new one. Such mixing of specifications must be a simple step for the user if one wants to promote reusability. Another situation in which grammars have to be added to the current parsing environment occurs with the instanciation of parameterized specifications. In addition to other checks, replacing formal parameters by actual ones calls for the addition of the grammars of these actual parameters (and their required specifications), with possibly some renaming.

The environment (\textsc{ASSPEGIQUE} in our case) must reflect and even induce modularity. The main requirements for our purposes are

- parsing must be oriented towards separate compilation, to allow the modification or enrichment of base specifications (e.g. \texttt{boolean} or \texttt{integer} in the example), without having to regenerate grammars for all that use them;
- it is also useful to introduce the notion of a \textit{current parsing environment}. It contains all the grammars associated with the specifications needed for the work in progress, and the list of operators that have been defined but not yet saved in a specification.

\textsc{CIGALE} gives the possibility of adding to and deleting grammars from this parsing environment.

Both the properties of incrementality and modularity put requirements to the method used for parsing as the complexity of operations to add or delete operators, as well as grammars, to the parsing environment must be considered. All the methods (LR, Coke-Younger-Kasami [1]...) that need some pre-computation on operators before parsing, are rather inadequate in this respect. Some typical computations are for example reduction to a normal form or a systematic detection of ambiguities. Modularity also implies that some checks on operators (unicity of the declaration, detection of possible ambiguities in the language...) cannot be done at definition time but must be postponed until parsing time. Such checks are related only to the current parsing environment.

Finally, most of the restrictions that may be asked, and checked for, when the language is stable, cannot be required from a user who incrementally defines its language.

\subsection*{1.4. Coercion, overloading and ambiguities}

Coercion and overloading are two specific notions usually mentioned in ADT and programming languages. In addition to semantical issues [14, 15], they also have syntactical implications which we describe below.

\textit{Coercions} are associated with the definition of partial ordering over sorts. To define a coercion from $s$ to $t$ is, in short, to accept each term of sort $s$ as a valid operand in every place where a term of sort $t$ is expected. In the PASCAL programming language, such a possibility exists between \texttt{integer} and \texttt{real}. Coercions may also be thought of simply as implicit operators (without any visible call) and they may be used for purely syntactical problems, e.g. as an easy way to stop 'recursive'
definitions. We saw such a use of coercions in the polynomial specification given in Fig. 1. In this case, coercions merely correspond to unit productions in context-free grammars.

An operator is said to be overloaded if there exist several operators with the same syntax but different domains or ranges, as in the classic example of the ‘+’ operators in PASCAL, defined for integers, reals or sets.

In CIGALE, overloading is extended to let operators share not only their syntax but also their domain, the range being the only way to distinguish between them. Its main use is the definition of several constants with the same syntax in different specifications. For example, a typical constant included in the specifications of stacks, files, lists... is the object empty. It is more convenient to let several empty constants (with different ranges) coexist in the system, rather than to constrain the user to name them empty-file, empty-stack or empty-list. Another frequent use of the overloading on the domain is the declaration of variables in axioms. From a parsing point of view these variables appear merely as constant operators! Unicity of the range of a constant would be a very stringent rule, whose verification is difficult to achieve due to modularity. This overloading on the syntax and domain is also supported for operators with a non-empty domain, but its practical use does not seem as wide because of the difficulty for the users themselves in distinguishing between expressions containing such operators.

As mentioned, coercion and overloading are present in modern programming languages and can be handled, in a restricted way, by conventional compilers. Unfortunately they are usually wired in the compiler and cannot be tailored to the user's needs. One exception is the ADA programming language [25] in which overloading may be introduced for literals and procedures or functions. A qualification mechanism must be used to solve ambiguous cases.

The generalization of coercion and overloading in a system leads to inefficiencies at parsing time, due to the ambiguities introduced in the language. In CIGALE, as in similar systems, one may distinguish between two kinds of ambiguities:

- Those due to the syntax of operators; they may correspond, for example, to precedence or associativity problems, or to the use of keywords (here lexical symbols) of the language as identifiers (constant name).
- Type ambiguities introduced by coercion and overloading. Let us consider again our specification of integer and polynomial and the input 0: with the definition given of a language, it may be parsed either as an integer constant, or by applying coercions, as a monomial or even polynomial term. Combining overloading and coercion raises of course similar problems for non-constant operators: there are at least two parsings for the expression 0 + 0, either as an integer or as a polynomial.

We believe there is a difference between syntactical and type ambiguities as the former can usually be solved statically (e.g. by precedence or associativity indications

3 This may also occur because of the simultaneous presence of different instantiations of a parameterized specification (e.g. list[integer] and list[boolean]). The situation may be handled by a renaming, but not always conveniently if each specification is already required by other ones.
An efficient use of these indications is nevertheless more difficult to offer than in other contexts because there may exist overloaded operators with different levels, or levels may interfere with coercions. On the contrary, type ambiguities are introduced on purpose by the user and cannot be solved statically. By putting requirements on the signature of the specifications one may define a notion of most natural parsing of ambiguous input (cf. Section 2.3). This is obviously not possible for operators with overloading on the domain: referring to the examples given, a simple expression such as empty cannot be parsed without the help of the user or without privileging one range! Nevertheless, the context of use frequently raises the ambiguity: with the (non-overloaded) operators:

\[ \text{push}\_\text{on\_} : \text{data stack} \rightarrow \text{stack, append\_to\_} : \text{data file} \rightarrow \text{file and } x : \rightarrow \text{data} \]

parsing of empty in expressions such as push x on empty or append x to empty can be uniquely determined.

Without any restrictions, such situations seem (and may really be) intractable, but most examples are not! Finally the balance is between putting requirements on the signature, with the problems evoked above, or making parsing more complex, and sometimes involving users in the parsing of difficult input. In ASSPEGIQUE and CIGALE, the second option was chosen, mainly because the design of specification is usually an interactive task.

**Note.** As one can notice, part of the properties mentioned above are related to the integration of CIGALE in an interactive environment, and may be shared by every environment in which the language to be recognized is not fixed at the beginning, but is constructed incrementally. Other properties (coercion, overloading and, partly, modularity) are more specific to ADT.

2. Standard parsing methods

Different methods for parsing or systems for generating parsers are described in this section. As these methods are well known [1], we do not describe them in detail and we limit ourselves to their ability to fulfill the requirements introduced in the previous section.

Compiling is now an old topic in computer science and a lot of methods have been developed. We cannot consider all methods and we shall first explain why we think that the examined ones are representative: one way of distinguishing between parsing methods is to classify them according to their scope. Roughly, one obtains two groups: the first one consists of the methods that accept every context-free grammar. They were the first methods designed and they derive productions in a systematic way, sometimes using backtracking to recover from local failures. Their complexity is rather high but they are still of interest when the grammar does not have adequate properties. The second group includes the algorithms designed for
sub-classes of context-free grammars (it can be decomposed into other subgroups . . .). They are now the predominant methods for standard languages, because they balance a limited scope with an optimal time complexity. They are deterministic: they find the next step to be carried out with the help of a few lookahead symbols. The properties of relevant grammars ensure that such a step is unique. Any method will resemble one or the other approach: either it accepts any context-free grammar or it first of all analyses the grammar. The remarks we shall make for the two approaches may thus be applied to other methods. We consider the best algorithm in each of these two groups, and we also recall the main possibilities of the OBJ system.

2.1. Earley’s algorithm

The first algorithm is the one proposed by J. Earley [10, 18]. It is representative for the methods that accept every context-free grammar and it is one of the best in this class (except Valiant’s algorithm for theoretical time complexity).

Given a context-free grammar $G(\Sigma, N, P, S)$ and the input $x_1 \ldots x_n$, the principle of the algorithm is to run through $x_1 \ldots x_n$ and to compute the set of items $I_i$, with $i \leq n$, which satisfies the following property:

Let $A ::= \alpha \beta$ be a production in $P$, $j \leq i$, $[A ::= \alpha \beta, j]^4$ is in $I_i$, iff there exists $\gamma$ and $\delta$ in $(N \cup \Sigma)^*$ such that $S \Rightarrow^* \gamma \alpha \delta$, $\gamma \Rightarrow^* x_1 \ldots x_j$ and $\alpha \Rightarrow^* x_{j+1} \ldots x_i$.

Once all these sets of items have been computed, the input is valid iff a production of the form $[S ::= \alpha, 0]$ is included in $I_n$.

The construction of these sets of items uses a systematic derivation of all productions compatible with the input and the computation, for each $I_i$, of the $I_j$ with $j < i$ (an example of set of items is given below). All combinations of productions which may be useful for deriving a prefix of the input are systematically inserted but in a factorized way (there are maybe several ways to generate a given item in a set of items, but this item appears only once in the set). These ‘parallel’ and exhaustive derivations, which avoid any need to backtracking, may be done in cubic time and quadratic space. Once all the derivations are gained, it is possible to exhibit one parsing tree also in cubic time. The algorithm is self-improving if the grammar has good properties (unambiguity for example), and performances may even become linear.

We give in Fig. 2 an example of sets of items with the (simplified) specifications for integer and polynomial given in Fig. 1. The names of sorts are abbreviated by their initials and productions are translated in BNF.

The characteristics of the algorithm w.r.t. our problem are the following:
- The most interesting aspect is that it accepts every context-free grammar without any restriction, and works with a reasonable worst-case complexity. It is thus possible to use operators with mix-fix syntax and the algorithm may handle

---

4 The dot is a symbol not in $\Sigma$ and is used as a marker.
In addition to these productions we must add three more productions to 'derive' each sort from the start-symbol S. Thus \( P = P \cup \{ S ::= I; S ::= M; S ::= P \} \).

We give the sets of items constructed for the input sentence \( 0 + 0 + 0 \).

The input is valid as we find items \([S ::= I, 0]\), \([S ::= M, 0]\), \([S ::= P, 0]\) in \( I_5 \), corresponding to the possible ranges. Several parsings are also possible for each range: \([S ::= I, 0]\) in \( I_5 \) may come from \([I ::= I + I, 2]\) or \([I ::= I + I, 0]\) in \( I_4 \), corresponding respectively to a right and left-associativity.

---

coercion and overloading, as they appear only as special cases of productions. The method does not need any costly computation on the operators and is thus compatible with an incremental construction of the grammar. Few operations are requested at each addition or deletion of an operator.

Less favourably, although the theoretical time complexity is cubic, this is modulated by a high coefficient; space complexity is also bad and may limit the practical use of the method. Both coefficients are related to the size of the grammar which in our context may be much bigger than the size of the input [18]. It is interesting to recall that the goal of this algorithm is to derive, in a factorized way, all the productions compatible with a prefix of the input! The construction may be improved in several ways: ignoring start-symbols, using some user's indications for precedence (with the problem evoked for overloaded operators) or using type indications. But the algorithm cannot take advantage, during the computation of the sets of items, of the partition of the grammars (modularity) and treats each rule equally, even coercions and overloaded operators. This is, perhaps, the major drawback.

A last remark is that during the 'set of items' construction all parsings for an
ambiguous sentence are systematically prepared. This is obviously costly, although less than for other methods, and may not be really useful.

2.2. LR parsing methods

We now consider an example of the algorithms designed for limited subclasses of the class of context-free languages. More precisely we are interested in the most commonly used: the LALR algorithm. This method is the standard way of compiling conventional languages, as it offers optimal time efficiency, and these languages may be generated by grammars having the required properties. This only needs a long and tedious transformation of the natural grammar for the language. Moreover compiler-generators have been developed for this subclass [19, 21], and parsers can thus be built automatically from the grammar. In fact we consider the use of such tools, rather than the method itself!

The principle of such systems is to 'compile' the productions of a grammar into a set of tables describing all the different contexts that can occur while parsing a correct sentence of the language. The properties of the grammars ensure that the use of these tables and of a few lookahead symbols allow us to find in a deterministic way the next step of the parser. An additional property is that an error is detected as early as possible in a left-to-right parsing.

The reasons for which such systems are inadequate in our context are listed below:
- The main reason is that the grammars we manipulate are usually not included in this subclass (the grammars must at least be unambiguous!). Although some constraints on the grammars may be relaxed by the use of systematic strategies to resolve ambiguities, this way is safe (w.r.t. to the capacity to recognize the whole language) only if carefully controlled by the user. This is possible only if the language is defined once and for all, and if there are only few conflicts. Coercions and overloading are of course rather dangerous in this respect...
- The second point is that the construction of the tables of contexts requires the examination of all the productions simultaneously. This is costly both in time and space (standard ways of reducing the number of states cannot be used here [2, 23]), and therefore tables cannot be re-computed with each modification of the set of operators. Similarly, the modularity cannot be handled efficiently since tables for the global environment cannot be obtained from tables for the individual grammars. The addition of grammars is thus equivalent to a merge of all the grammars in the environment. All the aspects relative to the incremental construction of the language are thus lost.
- The last remark is that it seems necessary to retain some non-determinism to resolve ambiguities in the language. This cannot always be achieved by the use of a fixed number of lookahead symbols.

2.3. The OBJ system

The OBJ system [16], developed by J. Goguen and his team, is the earliest environment for ADT manipulation where, in addition to a theoretical basis, atten-
tion was also focused on the user interface and on a large scale implementation of
the abstract notions of coercion and overloading. Since 1977 Goguen has emphasized
the advantages of such facilities in an ADT environment, despite the overhead it
may involve for the system. Several versions of the OBJ system has been developed
with different particularities, also from the parsing point of view. We consider here
only the latest version: OBJ2 [13] and we restrict ourselves to the parsing aspects.

We mentioned in Section 1 that our syntax for the operators was derived from
the OBJ's one. We thus only give below an example of OBJect:

```
obj LIST[X::TRIV] is  ; a simplified version of 'list of something'
sorts List.
subsorts Elt < List.  ; coercion from Elt to List
op nil: --> Elt.
  op_ _: List List--> List [assoc id nil].
    ; this operation is right-associative and has nil
    has identity element
  op head: List--> Elt.  ; operator with a functional syntax
var L: List.
var E: Elt.
eq head(E L) = e.
```

Concerning the user-interface, OBJ provides facilities to declare precedence and
right-associativity for operators. Other properties may also be defined for operators
and are used by the rewrite-rule engine of OBJ.

To correspond to its operational semantic, OBJ puts requirements on the signature
of the OBJects, in order that a term always has a lowest-parse w.r.t. the order defined
by coercions (see [15] for a formal definition of lowest-parse). This lowest-parse is
the one considered for input with (possible) type ambiguities. In addition, OBJ's
parser may use retraction operators (inverse of coercions); they are related to OBJ's
way of dealing with errors. Concerning syntactical ambiguities, they must be all
removed before an input can be accepted. Parsing is achieved by a recursive-descent
method with backtracking, lower precedence operators being tried first. The system-
atical search for all possible ambiguities should imply that parsing is an expensive
step.

3. CIGALE

In this section we describe the way parsing is achieved in CIGALE. A complete
description may be found in [26].

3.1. The operators

As we provided the way of defining operators in Section 1, we now focus on the
representation chosen for operators. We only recall that it is possible to add or
delete operators at will, as well as to include or remove languages from the parsing environment (a complete description of the way modularity is handled is postponed until Section 3.4). There is no restriction on the syntax of operators, and coercion and overloading (even on the domain) are allowed. The only requirement is that the relation over sorts induced by coercions must be acyclic.

A keypoint for such a system is the internal organization of the operators. It must ease the analysis of ambiguous terms and restrict, as much as possible, the need for backtracking. The internal structure of CIGALE is derived from a classic data structure for systems manipulating the notion of dictionary: the trie. With this structure, operators (productions) with a common prefix share a path in the tree and may be derived 'simultaneously' in a natural way (i.e. without modifying the productions as in an explicit factorization). This helps to start the parsing of a sentence without having to anticipate the production that will be finally used. Overloading will also be dealt with in a natural way, as we shall see below.

The grammar associated with a set of operators is a trie with three kinds of labels:
- terminal symbols which occur in the syntax of the operators;
- a special symbol is associated with each sort and represents the placeholder for an argument of this sort. Placeholders for different sorts are distinguished in the trie;
- the leaves contain all the possible ranges for overloaded operators with the same domain.

We give below the trie for the following set of operators:

\[
0: \rightarrow \text{int}; \quad 0: \rightarrow \text{real}; \quad 1: \rightarrow \text{real};
\]

\[
\text{add}_{- \text{mod}_-}: \text{int int int} \rightarrow \text{int}; \quad \text{add}_{-} : \text{int int} \rightarrow \text{int};
\]

\[
+_{-}: \text{int int} \rightarrow \text{int}; \quad \times_{-}: \text{int int} \rightarrow \text{int}; \quad +_{-}: \text{real real} \rightarrow \text{real};
\]

Terminal symbols are printed in italics, placeholders for operands are represented by the sort they correspond to (in bold); Leaves are enclosed in brackets.

Overloaded operators with the same domain (empty or not) share thus a path from the root to a leaf, and all corresponding ranges may be referred to by this
leaf. Coercions will not be used like the other operators and are stored in a separate graph.

The parsing routines consider the trie to be ordered. The order chosen for the different kinds of labels is globally equivalent to the standard way of resolving conflicts in LALR parsers (but in CIGALE the other alternatives are not suppressed and may be used if the first fails).

- A first choice is to try 'long' rules before shorter ones (see the `add_to_` and `add_to_mod_` operators). This is similar to the resolution of shift-reduce conflicts. Leaves are thus used only after the other ways of following the current path have failed.
- If different ranges are possible for a term and there is no constraint due to type-checking, then an arbitrary range is chosen (reduce-reduce conflict). This occurs mainly when overloaded operators with a same domain are used without disambiguating context.
- A final choice is to privilege nodes labelled by terminal symbols over nodes labelled by placeholders. There is no obvious criterion for choosing between these two kinds of labels as they are equi-probable and none is better than the other. We choose to use first terminal symbols because the cost of such a step is lower. Lexicographical ordering is used between nodes with lexical symbols and the partial order defined by coercions is used for those labelled by placeholders.

Algorithms to add or delete words (i.e. operators) in such a structure are well known and have linear performances w.r.t. the number of symbols in the word. As parsing is directly controlled by the structure of the trie, an incremental definition of the language is possible and really efficient.

3.2. The kernel

Before describing the way parsing is done, we first discuss the prealable choices that we made:

(1) The most critical choice was the one related to ambiguity: the first approach was to offer the user a careful protection against ambiguity. This can be achieved by a systematic detection of all possible ambiguities at the time operators are defined, or, at parsing time, by a systematic search of all parsings for the input as in OBJ. The obvious advantage of this approach is that the system and the user cannot have different understandings of a sentence. The second solution is a looser protection: only one parsing is searched for and the final responsibility is left to the user to accept or reject it. In this last case (s)he may control the parser and obtain the expected term by adding parentheses or indications for type-checking.

For CIGALE, we chose this second solution. A first point to notice is that there may not always exist a lowest parsing for a term, as we do not put restrictions on the specifications. But for most examples this lowest (and in our opinion most natural) parsing exists and will be the one provided by CIGALE. We believe that it is more convenient and efficient to always give only one result rather than an
exhaustive list of the possible parsings, which may be preferable for these few particular cases. In the case of a unique lowest parsing we coincide with OBJ.

Concerning syntactical ambiguities, we mentioned in Section 1 that a systematic detection of ambiguities, at definition time, is not compatible with an incremental construction of the language, nor with the handling of modularity. We believe that an exhaustive search for all parsings is an expensive operation and may not be really useful if most of the ambiguous input can be parsed in a natural way with the help of a few strategies. CIGALE will thus return only one parsing tree.

Given this 'philosophical' choice, an interesting requirement which could be added to the parser is that it be predictive, in order that users might know in advance which parsing tree will be returned for ambiguous sentences. We believe that the combination of this property with good default strategies provides an alternative to an exhaustive search of all parsings. (We shall give some examples in Section 4 to illustrate these ideas.)

(2) Coercions are not considered in the same way as other operators. They will be used only to satisfy type-checking constraints and will never be applied (except on user's request) to the root of a term.

(3) In the current implementation, backtracking is used, but only in a restricted way to keep some efficiency.

The parsing algorithm is mainly a traversal on the trie which represents the current set of operators, with provision for the type-checking. To simplify the description, we only present it as a recognizer rather than a parser. Given an input, it thus returns only true or false, depending on the inclusion of the input in the language. This allows to represent terms by the sort(s) they correspond to rather than to explicitly build them. The current state of the recognizer will be characterized by the sequence which remains to be recognized and the current vertex of the tree. We use the following notations:

- (): the empty sequence,
- \(\alpha \ast s\): the sequence formed by the symbol \(\alpha\) and the subsequence \(s\),
- root: the root of the tree,
- succ\((v)\): the set of successors of vertex \(v\),
- label\((v)\): the label of vertex \(v\). It is either a lexical symbol or a sort corresponding to a placeholder,
- ranges\((v)\), where \(v\) is a leaf, is the set of sorts stored in the leaf. It is not a singleton iff there are overloaded operators with the same domain,
- to distinguish between the different kind of vertices, we use the three predicates is_leaf, is_lexical and is_placeholder,
- \(t \leq t'\), where \(t\) and \(t'\) are two sorts, returns true iff \(t\) may be coerced into \(t'\).
- \(\perp\) represents a local failure.

The algorithm is described by three procedures (see Fig. 3)

- the first one, \(\text{traversal}\), is given the current vertex and a sequence representing the input not yet parsed. It tries to reach a leaf by following a path compatible with a prefix of the sequence. It returns a couple composed by the set of ranges
procedure traversal(v, seq);
if exists v' ~ succ(v) such that is_lex(v') and seq = label(v') * seq'
    result ← traversal(v', seq'); if result ≠ ⊥ return result endif
else if v ≠ root
    result ← fill_in({v' ∈ succ(v)/is_placeholder(v')}, traversal(root, seq))
    if result ≠ ⊥ return result endif
else if exists v' ∈ succ(v) such that is_leaf(v') return (ranges(v'), seq)
else return ⊥ endif
end traversal;

procedure fill_in(V, (T, seq)); V is a set of vertices, T a set of sorts, seq a string.
valid ← {(v1, t1) ∈ V × T/t1 ≤ label(v1)}
while valid ≠ ∅
    choose (v, t) ∈ {t ∈ valid/not exists (v', t') ∈ valid such that label(v') ≤ label(v)}
    result ← traversal(v, seq)
    if result ≠ ⊥ return result else valid ← valid-{(v, t)} endif
endwhile
return(⊥)
end fill_in;

procedure parse(seq); (T, seq') ← traversal(root, seq)
while (T, seq') ≠ ⊥ and seq' ≠ ()
    (T, seq') ← fill_in({v ∈ succ(root)/is_placeholder(v)}, (T, seq'))
endwhile
if (T, seq') ≠ ⊥ return(true) else return(false) endif
end parse;

Fig. 3. The parsing algorithm.

included in this leaf and the remaining input;
- the second one, fill_in, handles the assignment of terms to placeholders and thus
  the type-checking. It is called with three parameters: a set of vertices corresponding
  to the placeholders available, a set of ranges representing the term which must
  fill one of these placeholders, and the remaining input;
- the last one, parse, is called initially with the input to recognize. It runs through
  it and calls the previous procedures until either the input is exhausted or a failure
  is discovered.

In its current version CIGALE returns a syntax tree (with sort indications).

The main characteristics of the algorithm are listed below:
(1) First, as one can notice, the algorithm does not derive all productions sys-
    tematically (it is thus not complete. This point will be discussed in the next section).
For example, when an assignment of a term to a placeholder is done, it is not
cancelled once a leaf on the same path as the placeholder has been reached, even if it leads to a failure later. The most direct example of this is the loop of procedure \texttt{parse}.

The premature detection of operators (reduction) may give rise to the rejection of valid input: with the following operators, ordered in the tree as they are listed:

\begin{verbatim}
  _ _ : item item \rightarrow list;
  _ _ : item list \rightarrow list;
  a : \rightarrow item; b : \rightarrow item;
\end{verbatim}

one can check that \texttt{a b a} would be rejected, due to a premature recognition of \texttt{a b} as a \texttt{list}.

(2) Parsing and type-checking are independent steps: when a term is expected to fill placeholders, type-checking does not affect the choice of appropriate paths in the tree. Type-checking is used only afterwards to validate one of the terms found.

(3) As coercions are not stored in the tree, they cannot be used unless they are required for type-checking. As the partial order defined by coercions is used to order the nodes labelled by placeholders, lower sorts are used first. Coercions are thus minimally applied to operands, in particular for overloaded operators whose domains are related by coercions. For sorts not related by coercions, an arbitrary choice (one of these lowest possible sorts) is made. We reflect this feature by the \texttt{choose} operator in \texttt{fill_in}.

(4) A last point is related to associativity and precedence: if they are not imposed by the productions in the grammar, it follows from the algorithm that all operators are \textit{left-associative} and of equal precedence.

Backtracking is used but only in a restricted way: in the alternative of \texttt{traversal} and in the loop of \texttt{fill_in}.

These restrictions have been introduced to retain practical performances of the system, but are mostly based on the idea that we deal with terms and not only with grammatical productions. The search for an operand being done independently of type problems, parsing is more predictive.

3.3. Other properties

To balance partly these limitations of the parsing routines, two strategies are used to recover from local failures. A first kind of failure is due to a premature assignment of terms to placeholders. In this case the recovery routine will postpone the assignment until it does not lead to a failure in the remainder of the path. It corresponds to 'add brackets' around the right operand. Such a situation is illustrated by the operators below:

\begin{verbatim}
begin_end : inst \rightarrow inst, _ := _ : ident ident \rightarrow inst, x : \rightarrow ident and y : \rightarrow ident
\end{verbatim}

With the main routine \texttt{begin x:= y end} would be refused, as \texttt{x} is not a valid \texttt{inst}. Input is recognised by the recovery routine as \texttt{begin (x:= y) end}.
The second routine handles operators with a right-associative definition (they are not forbidden) as in the example given for the lists in OBJ. It also works by postponing the assignment of terms to placeholders.

**User indications:** To raise ambiguities that may induce the rejection of valid input (even after recoveries), or to switch the parser towards a non-standard result (for example, it may be useful to introduce axioms in which coercions are applied at the root of a term) two kinds of indications may be given. The first one is the use of parentheses. In conventional languages, the possibility of adding parentheses is frozen in the grammar, and they can be used as a way of controlling the parsing only for limited sub-parts of the language, mainly arithmetic expressions. In CIGALE, parentheses may be used in a generalised manner and a bracketed expression may appear everywhere an operand is expected.

The second way of controlling the parser is by using the clause `parse-as` which constrains an expression to be of a given type. Once again, as for the assignment of an operand to a placeholder, this indication is not used to control the steps of parsing but to recompute, using coercions and overloading, the type of the result once it is obtained. Thus, if an input can be parsed without indications, CIGALE returns the same result if the corresponding sort indications are added (recall that only one parsing is returned), but different results are obtained with different indications.

These indications may also be used to speed up parsing, by restricting the scope of the input for which recovery strategies have to be used.

**Completeness & soundness:** The soundness of the parsing functions can be easily checked. But as mentioned above, the restrictions put on backtracking implies that the algorithm, without user indications, is not complete, even after recovery strategies. In [26] classical classes of grammars have been studied to see for which ones the algorithm, without user indications, is complete. It was impossible to obtain a total characterization (it would be equivalent to characterize when the context or recovery strategies are powerful enough to accept every input) but the situation can be pictured by

![Diagram](image-url)
But this characterization is too restrictive as user indications are a means of forcing along the steps of the parser. By their systematic use, CIGALE behaves as a term-checker (in the same way that there are some proof-checkers) and accepts every valid input.

**Performances:** Although limitations have been introduced on backtracking, time complexity for parsing is the one usual for algorithms with backtrack: exponential in the worst case. This result may appear incompatible with an interactive system, but it is balanced by two remarks:
- As indicated in Section 1, time complexity for parsing is not the only criterion used to evaluate the method: we must also consider time for adding and deleting operators and grammars. This step is usually hidden for other methods since it is performed once and for all (or it appears so, when the grammar is finally designed);
- The length of the input is usually small: an axiom in a specification is only a small piece of knowledge about operators and is thus not a complex combination. For example, the biggest axiom included in specifications for a realistic application (part of a telephone switching system [8]) contains about 30 operators with one-third of constants. This remains valid, although less significant, if one considers not only parsing of axioms but also parsing of expressions given to a symbolic evaluator. In this context an exponential method may be a practical one, as the experiments in the ASSPEGIQUE environment proved. Moreover grammars for which this complexity is reached are tricky ones and have ambiguities which are more related to syntactical problems than to type problems. CIGALE has been built with this second kind of ambiguity in mind.

An example is $\Theta = \{1 : s \rightarrow t; 1 : t \rightarrow t \}$. With these operators strings $1 \ldots 1$ are parsed in exponential time. With this grammar, other systems would also take exponential time to display all parsings!

### 3.4. Modularity

The last point we want to describe is the way modularity is handled. We saw that every specification is represented by a trie. Rather than merging all these tries in a unique one, we choose to keep the notion of a grammar attached to a specification and we store them in independent tries (although merging of tries is not a complex algorithm, its practical complexity increases with the size and number of tries). Our solution is to maintain a circular list of tries, each of which can also be accessed directly. This list includes all (and only those) tries corresponding to specifications required by the work on hand and contains a 'scratch-grammar' in which every addition or deletion of operators occurs.

Let us consider a parsing environment required for the specification of polynomial: the grammar that must be included are integer and boolean. The definition of the new operators are stored in the scratch-grammar, which can be saved later as an
independent specification. The parsing environment is the following:

The list specification illustrates the possibility of having grammars which are loaded in the environment (it was required for a previous work) but not included in the parsing environment (but it is no longer needed). A simple inclusion in the circular list of the trie corresponding to list would allow us to use lists of polynomials, list of integers . . .

Graphs representing coercions have to be merged. As their size is very small, this step is not expensive. Some care must also be taken to ensure that all the ranges for overloaded operators with a common domain can be reached, even if they are stored in different tries.

Parsing is made a little bit more complex, as we must deal with a collection of tries, but all other operations are made easier: with this structure, the addition or deletion of grammars can be done easily and in a quasi constant time. It allows us to take advantage of ADT modularity: the search for an operand of a given sort may be started in a preferential way in its privileged grammar: the one which defines this sort. Other grammars may be reached, if required, by the circular list (e.g. for external operators). Operations related to the environment are also eased: display of individual grammars, protection against modification of given grammar . . .

4. A more detailed example

In this section we present a more detailed example of the use of CIGALE. To simplify this example, we present CIGALE as an independent tool, without its usual environment (ASSPEGIQUE). Thus, manipulation of operators and grammars are done at the user level, instead of being managed by ASSPEGIQUE. Add-op, del-op (resp. add-gram, del-gram) are the basic functions to add or delete operators (resp. grammars). visualise displays the grammar used as parameter. parse returns the parsing tree for the input [27]. To simplify this tree, the name of an operator is composed only by its syntax and its range (its domain is implicitly given by the range of its arguments). In the examples, input to CIGALE is given in italic:
First we introduce some operators on integers and reals

\[
\begin{align*}
(add-op \_ + \_ : real real \to real) &\to t \\
(add-op pi : \to real) &\to t \\
(add-op 0: \to integer) &\to t \\
(add-op 1: \to integer) &\to t \\
(add-op x: \to integer) &\to t \\
(add-op \_ + \_ : integer integer \to integer) &\to t
\end{align*}
\]

Let us try some simple examples:

\[
\begin{align*}
(parse 0 + 1) &\to \_ + \_ : integer \\
&/ \ \\
&\_ : integer \\
&\_ : integer
\end{align*}
\]

\[
\begin{align*}
(parse pi + pi) &\to \_ + \_ : real \\
&/ \ \\
&\_ : real \\
&\_ : real
\end{align*}
\]

If we add a coercion from integers to reals:

\[
\begin{align*}
(add-op \_ : integer \to real) &\to t
\end{align*}
\]

Our conventions ensure that \(0 + 1\) remains an integer term (cf. the previous section)

\[
\begin{align*}
(parse 0 + 1) &\to \_ + \_ : integer \\
&/ \ \\
&\_ : integer \\
&\_ : integer
\end{align*}
\]

But coercions are used to satisfy type constraints:

\[
\begin{align*}
(parse 0 + pi) &\to \_ + \_ : real \\
&/ \ \\
&\_ : real \\
&\_ : real
\end{align*}
\]

And coercions may also be used on user request:

\[
\begin{align*}
(parse (parse-as real 0 + 1)) &\to \_ : real \\
&| \\
&\_ + \_ : integer \\
&/ \ \\
&\_ : integer \\
&\_ : integer
\end{align*}
\]

The previous input accepts another syntax tree with the + over real at the root. Both results must be semantically equivalent! Again, only one result is given.
Now we add the grammar for polynomials given in Fig. 1:

\[
\text{(add-gram 'polynomial)} \rightarrow \text{added}
\]

\[
\text{(parse } p + m + 0) \rightarrow \quad \ldots \quad + \ldots : \text{polynomial}
\]

\[
\quad \ldots : \text{polynomial}
\]

\[
\quad \ldots : \text{monomial}
\]

\[
\quad + \ldots : \text{polynomial}
\]

\[
\quad + \ldots : \text{polynomial}
\]

\[
p : \text{polynomial} \quad m : \text{monomial} \quad 0 : \text{integer}
\]

\[
\text{(parse } \{ p + m \} x + 1) \rightarrow \quad + \ldots : \text{integer}
\]

\[
\quad \ldots \{ \ldots \} : \text{integer}
\]

\[
\quad \ldots : \text{polynomial}
\]

\[
x : \text{integer}
\]

\[
p : \text{polynomial} \quad m : \text{monomial}
\]

Parentheses are used to get a non-standard result:

\[
\text{(parse } \{ p + m \} (x + 1)) \rightarrow \quad \ldots \{ \ldots \} : \text{integer}
\]

\[
\quad \ldots : \text{polynomial}
\]

\[
\quad \ldots : \text{integer}
\]

\[
p : \text{polynomial} \quad m : \text{monomial} \quad x : \text{integer} \quad 1 : \text{integer}
\]

Now we add lists (see Fig. 1.):

\[
\text{(add-gram 'list)} \rightarrow \text{added}
\]

\[
x \text{ becomes an overloaded expression (it was a variable in list and integer)}
\]

\[
\text{(parse } x) \rightarrow x : \text{item}
\]

But both results may be reached:

\[
\text{(parse (parse-as integer } x \text{))} \rightarrow x : \text{integer}
\]

And the context may raise the ambiguity:

\[
\text{(parse } x + x) \rightarrow \quad + \ldots : \text{integer}
\]

\[
x : \text{integer} \quad x : \text{integer}
\]

\[
\text{(parse } x x x) \rightarrow \quad \ldots : \text{list}
\]

\[
x : \text{item} \quad \ldots : \text{list}
\]

\[
x : \text{item} \quad \ldots : \text{list}
\]

\[
x : \text{item}
\]
How to deal with list of polynomials and list of integers? We add copies of lists in which operators have been renamed to let polynomial (resp. integer) replace item. These operations should be handled by the environment (ASSPEGIQUE), and are simulated here. \( \text{linteger} \) (resp. \( \text{lpolynomial} \)) stands for \( \text{list[integer]} \) (resp. \( \text{list[polynomial]} \))

\[
\begin{align*}
\text{del-gram } ' \text{list}' & \text{ } \rightarrow \text{ deleted} \\
\text{add-gram } ' \text{lpolynomial}' & \text{ } \rightarrow \text{ added} \\
\text{add-gram } ' \text{linteger}' & \text{ } \rightarrow \text{ added} \\
\text{visualise } ' \text{linteger}' & \text{ } \rightarrow \\
& \text{(op} \_ \text{ : integer } \text{linteger } \rightarrow \text{linteger)} \\
& \text{(op} \_ \text{ : integer } \rightarrow \text{linteger)} \\
& \text{(op empty : } \rightarrow \text{linteger)} \\
& \text{(op} \text{x} : \rightarrow \text{integer)} \\
& \text{(op first(\_): linteger } \rightarrow \text{integer)} \\
\text{visualise } ' \text{lpolynomial}' & \text{ } \rightarrow \\
& \text{(op} \_ \text{ : polynomial } \text{lpolynomial } \rightarrow \text{lpolynomial)} \\
& \text{(op} \_ \text{ : polynomial } \rightarrow \text{lpolynomial)} \\
& \text{(op empty : } \rightarrow \text{lpolynomial)} \\
& \text{(op} \text{x} : \rightarrow \text{lpolynomial)} \\
& \text{(op first(\_): lpolynomial } \rightarrow \text{polynomial)} \\
\end{align*}
\]

Once again, terms are preferably integers rather than monomials or polynomials

\[(\text{parse first } '0 1 0 + 1') \rightarrow \text{first(\_): integer} \]

\[
\begin{align*}
& 0: \text{integer} \quad \_ \_ : \text{linteger} \\
& \quad \_ : \text{linteger} \\
& \quad 1 : \text{integer} \quad _- : \text{linteger} \\
& \quad \_ + _- : \text{integer} \\
& \quad 0: \text{integer} \quad 1: \text{integer} \\
\end{align*}
\]

Now, empty (which belongs to the list specification) is included in both linteger and lpolynomial specifications. The context raises the ambiguities:

\[(\text{parse first } '0 1 \text{ empty}) \rightarrow \text{first(\_): integer} \]

\[
\begin{align*}
& 0: \text{integer} \quad \_ \_ : \text{linteger} \\
& \quad \_ : \text{linteger} \\
& \quad 1 : \text{integer} \quad \text{empty: list[integer]} \\
\end{align*}
\]
Interactive grammar construction and expression parsing

\[
\begin{align*}
(p \text{ parse first } (p \text{ p empty})) \rightarrow \\
\text{first(}_\text{priv}_{\text{priv}}\text{): polynomial} \\
\quad \text{polynomial} \\
\quad \text{lpolynomial} \\
\quad \text{lpolynomial} \\
\quad \text{lpolynomial}
\end{align*}
\]

In the next example a little help is needed

\[
\begin{align*}
\text{parse first } (0 \text{ 1 X** 1 empty}) \rightarrow \text{error on: }> \\
\text{parse first } (0 \text{ 1 X** 0 empty}) \rightarrow \\
\text{first(}_\text{priv}_{\text{priv}}\text{): polynomial} \\
\quad \text{polynomial} \\
\quad \text{lpolynomial} \\
\quad \text{lpolynomial} \\
\quad \text{lpolynomial} \\
\quad \text{lpolynomial} \\
\quad \text{lpolynomial}
\end{align*}
\]

5. Conclusion

The system presented in this paper is written in Franz-LISP and is extensively used in the ASSPEGIQUE environment. The experiments made in this project have reinforced our conviction that great care must be focussed on these parsing problems, to offer a really powerful environment to programmers. A parsing system is a significant part of the kernel module for an ADT specification environment since without an efficient help for manipulating grammars and operators, many of the theoretical aspects of ADT (e.g. coercion, overloading or modularity) cannot be fully used and remain theoretical features.

We believe that there will be an increasing need for systems which let the users choose their syntax and define their language incrementally. Such systems allow to build convenient user interfaces for many experimental tools (object-oriented or PROLOG-like languages, rapid prototyping . . . ), in an easy, natural and fast way. For such experimental tools these properties are maybe the most important ones. 

\[\text{\textsuperscript{5} Cigale has been tested for such purposes: it has been used to build an interpreter for a FP-like language [9].}\]
Acknowledgment

Special thanks are due to G. Guiho and J.C. Raoult for many suggestions during the development and the formalization of CIGALE. M. Bidoit, C. Choppy and S. Kaplan are acknowledged for their fruitful remarks and criticisms. We wish to thank M.C. Gaudel for her support and for detailed comments on drafts of this paper.

This work has been partially supported by A.D.I. Contract No. 639 and the C.N.R.S. (Greco de Programmation).

References