CENTAUR : the system

Patrick Borras, Dominique Clement, Thierry Despeyroux, Janet Incerpi, Gilles Kahn, Bernard Lang, Valérie Pascual

INRIA

UNITÉ DE RECHERCHE
INRIA-SOPHIA ANTIPOLIS

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tel. (1) 39 63 55 11

Rapports de Recherche

N° 777

CENTAUR : the system

Patrick BORRAS
Dominique CLEMENT
Thierry DESPEYROUX
Janet INCERPI
Gilles KAHN
Bernard LANG
Valérie PASCUAL

DECEMBRE 1987

CENTAUR: the system

Patrick Borras, Dominique Clément, Thierry Despeyroux,
Janet Incerpi, Gilles Kahn, Bernard Lang, Valérie Pascual

Abstract

This paper describes the organization of the CENTAUR system and its main components. The system is a generic interactive environment. When given the formal specification of a particular programming language – including syntax and semantics – it produces a language specific environment. This resulting environment includes a structure editor, an interpreter/debugger and other tools, all of which have graphic man-machine interfaces.

CENTAUR is made of three parts: a database component, that provides standardized representation and access to formal objects and their persistent storage; a logical engine that is used to execute formal specifications; an object-oriented man-machine interface that gives easy access to the system's functions.

CENTAUR is essentially written in Lisp (Le_Lisp). The logical engine is Prolog (Mu-Prolog). The man-machine interface is built on top of the virtual graphics facility of Le_Lisp, itself primarily implemented on top of X-Windows.

CENTAUR: le système

Résumé

Cet article décrit l'organisation du système CENTAUR et ses composants principaux. Ce système est un environnement interactif générique. Lorsqu'on lui fournit la spécification formelle d'un langage de programmation donné –spécification sémantique aussi bien que spécification syntaxique– il produit un environnement spécifique pour ce langage. L'environnement ainsi construit contient un éditeur structuré, un interprète/metteur-au-point et d'autres outils, le tout muni d'une interface graphique.

CENTAUR comprend trois parties: un composant de type base de données, qui fournit une représentation et des moyens d'accès standard aux objets et à leur sauvegardes; une machine logique qui sert à exécuter les spécifications formelles; une interface homme-machine orientée objet qui permet un accès agréable aux fonctionnalités du système.

CENTAUR est pour l'essentiel implanté en Lisp (Le_Lisp). La machine logique est Prolog (Mu-Prolog). L'interface homme-machine est construite à partir du terminal graphique virtuel de Le_Lisp, lui-même implanté au-dessus du protocole X-Windows.

CENTAUR: the system

P. Borras[1], D. Clément[2], Th. Despeyroux,
J. Incerpi, G. Kahn, B. Lang[3], V. Pascual

INRIA, Sophia–Antipolis
06565 Valbonne CEDEX, FRANCE†

This paper describes the organization of the CENTAUR system and its main components. The system is a generic interactive environment. When given the formal specification of a particular programming language – including syntax and semantics – it produces a language specific environment. This resulting environment includes a structure editor, an interpreter/debugger and other tools, all of which have graphic man-machine interfaces.

CENTAUR is made of three parts: a database component, that provides standardized representation and access to formal objects and their persistent storage; a logical engine that is used to execute formal specifications; an object-oriented man-machine interface that gives easy access to the system's functions.

CENTAUR is essentially written in Lisp (Le_Lisp). The logical engine is Prolog (Mu-Prolog). The man-machine interface is built on top of the virtual graphics facility of Le_Lisp, itself primarily implemented on top of X-Windows.

1. Introduction

The CENTAUR system is a generic interactive environment. When provided with the description of a particular programming language – including its syntax and semantics – it produces a language specific environment. The resulting environment comprises a structure editor, an interpreter/debugger and other tools, all of which have graphic man-machine interfaces. We believe that designing new languages – whether large or small, specialized or general – will be a very common activity in the future. Hence it makes sense to build such a system.

In the design of CENTAUR, we are faced with at least three challenges:

i) Is it possible to use as the description of a programming language a *formal specification* in the sense used by researchers in programming language semantics?

ii) Is it possible to obtain on that basis an efficient system, both in terms of speed and in terms of memory space?

iii) Is it possible to create a man-machine interface that is comparable in convenience to what can currently be found in dedicated commercial systems, say on a MacIntosh?

---

[1] *SEMA, Paris*
[2] *SEMA, Sophia-Antipolis*
[3] *INRIA, Rocquencourt*
† This research is partially supported under ESPRIT, Project 348.

This paper describes the CENTAUR system. The conclusion presents preliminary answers to the questions above.

1.1. *Outline of the CENTAUR architecture*

The architecture of the Centaur environment generator is based on a decomposition into three classes of functions:

- a *kernel* for the representation and manipulation of structured objects,
- a *specification level* to support both the syntactic and the semantic aspects of languages
- a *user interface* that takes care of all interactive communication between the system and its users.

The kernel plays a central part in this architecture because it is intended to be used, directly or indirectly, by all other components of the system. Every structured object is represented within the system as an abstract syntax tree that is manipulated with primitives of the kernel.

1.2. *The kernel*

The purpose of CENTAUR's kernel is to support symbolic manipulation of structured documents. By symbolic manipulation we mean destructive manipulation, typically used while editing documents, as well as evaluation, more common in semantic processing. The kernel of the CENTAUR system is made of two specialized components:

- an abstract machine that handles syntactic aspects, called the Virtual Tree Processor or VTP [21],
- a logical machine that handles semantic aspects, i.e. evaluations of all kinds. At this stage the logical machine is a Prolog interpreter.

These two components are intended to be used as co-routines. They are connected through an interface that contains two classes of primitives:

- *control* primitives to call Prolog from within the Virtual Tree Processor and to call the Virtual Tree Processor from within Prolog,
- *transformation* primitives to perform coercions between Virtual Tree Processor trees and Prolog terms.

1.3. *The specification level*

To generate an interactive programming environment for a given language one must provide specifications for its concrete syntax, abstract syntax, and semantics. These specifications are themselves written in appropriate formalisms.

1.3.1. *Specifying syntax*

The specification of syntax contains the components that are necessary for deriving a structure-oriented editor. After compilation of this specification, one obtains a scanner, a multi-entry parser, a pretty-printer, and abstract syntax tables. The scanner and parser are used to transform a textual representation into a structured representation. The unparser, or pretty-printer, is used to transform a structured representation to a textual form. The abstract syntax tables are used to check the validity of *editing* operations.

The specifications of concrete and abstract syntax, together with their relationship, are presently written in METAL [20], a formalism developed for the MENTOR system [10] [11]. A METAL specification is a collection of grammar rules, with annotations that specify what

abstract syntax trees should be synthesized. Pretty-printing of abstract trees is defined in the PPML formalism [25]. A PPML specification is a collection of unparsing rules associated with abstract syntax patterns.

In the future we hope to merge all syntactic aspects of language definition within a single formalism [15].

1.3.2. *Specifying semantics*

To derive systematically a language-specific interactive programming environment it is necessary to go beyond syntax and introduce semantic specifications. For example:

- from a specification of static semantics, one wants to derive a type checker and context-dependent interactions,
- from a specification of dynamic semantics, one wants to derive an interpreter and a symbolic debugger.

The semantics of a programming language may be specified with a variety of formalisms. We have experimented with a formalism based on Natural Deduction that we call Natural Semantics [19]. A semantic specification is written in TYPOL [6], the computer version of Natural Semantics. It is then type-checked and compiled into an executable form. At the present time Typol specifications are compiled into Prolog clauses.

1.4. *Principles of the man-machine interface*

The functions of CENTAUR are accessed in two non-exclusive ways: through direct function calls and through the man-machine interface. The design of the interface presupposes that the user is interacting with CENTAUR *via* a modern workstation, using a keyboard, a mouse, and a high resolution screen. Most of the time, the user makes his intentions known using the mouse. When a mouse button is depressed, the location of the mouse determines what graphic object the user is talking to and a finite automaton takes control of the mouse until the button is released.

A variety of graphic objects are used in the man-machine interface: menus offering various options, buttons to control execution, pseudo-terminals, etc. More specific to CENTAUR are the *views* where abstract syntax trees are displayed[1]. Structure editing as well as text editing are available in a CENTAUR view. Within a view, a subexpression is emphasized: it is printed in boldface. This subexpression is the subject of most commands. The simplest gesture with the mouse is to point at an expression to emphasize it.

The user is normally manipulating several formalisms at the same time. Each formalism has its own structure and its own specific environment. In fact, in a single CENTAUR view more than one formalism can be present: typically annotations are interspersed within the text of a program. The operations that are permitted and the menus that pop up all depend on the formalism of the currently emphasized expression.

Furthermore, the various objects that appear on the screen are usually not independent. For example, one view may display a source program, and another view the result of translating this program into an abstract machine code; or two views may look at the same abstract syntax tree with different pretty-printers. A network of such dependencies is maintained automatically to assist the user in keeping a consistent screen. This *logical control* of the interface relies for the time being on the Prolog component of CENTAUR.

[1] Trees are displayed as text, graphic representations have not yet been considered.

1.5. *Implementation Issues*

CENTAUR is implemented in Lisp and Prolog. It runs on standard modern workstations. It is also possible to run CENTAUR on a centralized time-shared computer and retain a high quality graphics interface because the man-machine interface is built on top of X-Windows [12].

1.5.1. *Using Lisp*

The design and implementation of CENTAUR are based on our experience with the design, implementation, and actual use of a first generation system called MENTOR [10] that was implemented in Pascal. The software technology used for environment generators is clearly based on symbolic computation. Traditional languages such as Fortran, Pascal, or Ada are not well suited for implementing such applications.

The selection of an appropriate implementation language hinges around the following issues:

- automatic memory management (garbage collection),
- type discipline (strict vs. lax),
- functional parameters and variables,
- modularity and separate compilation,
- exceptions (for program robustness and for backtrack programming),
- dynamic linking of program modules.

Several of these issues are well solved in a functional programming language like ML [22]. An efficient implementation of ML was unfortunately not available at the start of the project. Furthermore it was not clear whether such a high-level language with a very rigorous typing structure would allow the *efficient* use of sophisticated programming techniques that had not been built into the language in advance.

After some experimentation with C it was decided to use Lisp as a flexible *low-level language* for the implementation for the VTP. The Lisp dialect actually chosen is Le_Lisp [3]. This dialect of Lisp was selected for its availability on a large variety of machines and the ease with which it is ported on new hardware. Furthermore the Le_Lisp compilers produce efficient code.

1.5.2. *Using Prolog*

Early experiments had shown that Prolog was a convenient target language for the compilation of semantic specifications [7]. These experiments were confirmed repeatedly during the project development. In the context of an interactive programming environment mostly implemented in Lisp, using Prolog as an inference system implies that Prolog may be used and accessed from within Lisp, and in turn, that Lisp may be called from within Prolog. Then Prolog is available as a coroutine to Lisp.

An immediate advantage of this strategy is the possibility of experimenting with different Prolog engines independently of the rest of the kernel. This has already proven useful in switching from a first Prolog interpreter, C-Prolog of Edinburgh University [27], to a more powerful Prolog system, the Mu-Prolog of Melbourne University [24]. In the near future we hope to change over smoothly to *compiled* Prolog, obtaining a substantially faster system.

This approach, however, requires coercions between different internal representations. For example a Virtual Tree Processor abstract tree must be transformed into a Prolog term. Using directly the primitives for constructing Prolog terms (rather than the Prolog reader) the cost

of this coercion, both in time and in memory space, is roughly that of a copy. One may wonder whether a more integrated approach, where every object has a unique representation that is used by all processors, would not be superior. Without entering this debate, we are convinced that even with a tighter integration it is not possible to avoid transformations, because the proper choice of representation of structured objects depends on what manipulations have to be performed on these objects. This need for multiple representations seems to be inherent in the manipulation of symbolic objects.

From Lisp one uses Prolog in a straightforward manner. Goals are sent to Prolog via a buffer, the Prolog mailbox, with a call to the *mailtoprolog* procedure. Then to solve this goal, one calls a *resolve* that:

- fetches a goal in the Prolog mailbox
- silently resolves the goal
- then returns to its caller. Later calls will find Prolog in the exact state where it was left. Hence Prolog and Lisp behave as two co-routines. Evaluable predicates may be written in Lisp. This is in particular how Prolog is given access to CENTAUR's man-machine interface.

1.5.3. *Using a modern workstation*

The development and effective use of CENTAUR demand, in its current form, a computer system with a large memory (8 megabytes), reasonable computing power, and a graphic interface. Modern workstations provide the appropriate hardware. Code written in Le_Lisp is perfectly portable but graphic interfaces are far from standardized. To isolate the programmer from system variations, Le_Lisp includes a virtual multiwindow graphic terminal. CENTAUR was initially developed using software from Brown University [26] to implement the Le_Lisp graphic interface. A new binding of the interface has been implemented on top of X-Windows, a graphics software package from MIT [12]. In this new incarnation, CENTAUR may run on one computer while it is accessed via a distinct graphic workstation. Using X-Windows should make CENTAUR available on most workstations, with the unfortunate drawback that X-Windows is not entirely stabilized today.

2. The database

The first component of the kernel is the Virtual Tree Processor or VTP. The VTP defines and implements efficiently a protocol supporting creation, manipulation, and persistent storage of abstract syntax trees.

2.1. *Structure of the VTP*

As a protocol, the VTP is a collection of abstract data types. Certain functions that are not logically necessary (for example iterators) are included to obtain a far more efficient implementation. The protocol is *open* in that it provides a method for creating new types of objects to enrich the VTP.

The specification of the various data types yields a natural modularity to the VTP. Another level of modularity results from following an object-oriented approach. Many objects provided by the VTP carry with them their own dynamic types. These types are organized hierarchically. For example, a generic pretty-printer is defined for all abstract syntax trees. When a new formalism is defined, the abstract syntax trees pertaining to this formalism are pretty-printed with the generic pretty-printer. As specific rules are added for the new formalism they supersede the default rules used by the generic pretty-printer. The main

advantage of this architecture is to allow *extending the system* without any alteration of existing code, as long as the type hierarchy is extended.

2.2. *Main concepts of the VTP*

The abstract concepts defined and implemented in the VTP are motivated by a number of earlier experiments on syntax-directed document manipulation, either with our MENTOR system or with other systems essentially built on the same syntactic base.

The VTP is organized around very few concepts. A *formalism* embodies the notion of an abstract syntax. A *tree* belongs to one formalism, but it may have subtrees in other formalisms, for example using the *gate* mechanism. *Contexts* materialize the idea of one or several locations in a tree. *Annotations* may be hooked to abstract syntax trees to retain some information without modifying their structure.

2.2.1. *Constructing a formalism*

An abstract syntax is represented in the VTP by an object in the class *formalism*. A formalism is made of objects in the classes *operator* and *phylum*, where a phylum is a set of operators. Primitives are available to create a new formalism and populate it with new operators and phyla. When creating an operator, its arity and the phyla of its operands must be provided. Operators come in three kinds: operators with a fixed arity (pure constructors), list operators where the list may not be empty (associative constructors), and list operators where the list may be empty (there exists also a neutral element). It is possible to iterate over all operators and over all phyla in a formalism, and over all operators in a phylum.

When all phyla and and all operators of a formalism have been given, the formalism is declared complete. Then a number of global calculations are performed that result in faster tree traversals and more compact tree representations in persistent storage. The formalism so constructed has a persistent representation: it may be saved in a file and read in at a later time.

The natural way to create a new formalism is to use a specification language. A compiler for the specification language will then use the primitives provided by the VTP to create the formalism. For example, the language METAL is currently used to specify simultaneously the concrete and abstract syntax of a language. It is possible to experiment with other syntax specification languages (e.g. SDF [15]) without changes to the rest of the system because all other components of CENTAUR are only interfaced with VTP objects.

2.2.2. *Trees*

From an operator and an adequate number of subtrees one can create a new tree. The tree is then said to belong to the formalism of its top operator. The VTP provides primitives to navigate in a tree (such as *down*, *left*, *right*, *up*, etc.) and to modify the tree: replacing a subtree by another one, inserting and deleting elements in lists. All modifications *fail* if they are incorrect with respect to the abstract syntax. Syntax checking is strictly local and very fast. It is possible to iterate efficiently over all subtrees of a tree. Finally, trees have a representation in persistent storage.

The VTP defines a *gate* mechanism to associate values to the leaves of an abstract syntax tree. For example, identifiers or integer literals are zero-ary operators: their value is obtained in traversing a gate. In this way, the value associated to a leaf may be *any* VTP object, in particular an abstract syntax tree in a different formalism.

Typically, trees will be constructed by a parser and modified by interactive editing. The VTP defines the protocol that any parser must use to create trees, but it is of course not committed to any particular kind of parser. Likewise it provides the primitives to build an interactive structure editor but it does not enforce a specific mode of interaction.

2.2.3. *Contexts*

An essential aspect of the VTP is that it supports *imperative* manipulation of formal documents. Thus the abstract representation of a document is not a recursive structure that is built and processed in a purely applicative way. Rather it is a kind of database to explore and modify. As a result it is frequently necessary to keep references to places of interest in the manipulated structure. Consider a very frequent operation during interactive editing, memorizing a location in an abstract syntax tree. One possibility is to use as designator a value of class *tree*. Using this value as argument to the appropriate VTP primitive, it is possible to move to adjacent nodes (father, sons, siblings), or to modify the tree locally (by insertion, deletion, replacement, etc.). However a modification of the tree may have unexpected (or unwanted) side-effects. Assume that the *tree* value P were also used to designate a location within a larger tree T. After a modification of T that changes the subtree located where P is (by deletion or replacement), the designator P still denotes the old subtree. Hence it is no longer connected to the original tree T. Thus the designation of the location within T is lost, which is usually not the desired effect in an editing environment.

This example shows that pointing at a sub-object within a structured object can denote:

a. the sub-object that is placed at the designated location,

b. or the location of that sub-object within the larger object.

We call *contexts* values referring to locations (b) rather than sub-objects. A context value is thus a place within a mutable object, where a component of that mutable object may be stored [1]. Several classes of contexts are defined in the VTP. The classes *subtree*, *annotation*, and *gate* correspond to the mechanisms used in constructing trees. The class *sublist* designates several consecutive locations in a list.

In the current implementation of the VTP, contexts do not have a persistent representation. A more abstract notion of context that will denote an arbitrary set of disjoint locations will be added to the VTP in the future. It will have a persistent representation so that, for example, the state of an editing session may be saved.

2.2.4. *Annotations*

Annotations are a more formal and structured view of the old concept of *"property lists"*. Property lists were introduced in Lisp. The idea was to attach to atoms a list of *independently* managed tagged values, where each tag or *"property"* characterizes the role of the value.

In the VTP, the role of an annotation must declared by first defining a *decor*. For example, one can define a decor called "pragma" that can be used to attach string annotations to nodes of abstract syntax trees representing Pascal programs. So the definition of a decor includes its name (e.g. pragma), the type of the annotation itself (e.g. string) and the type of objects that may be decorated by annotations in this decor (e.g. Pascal programs).

Annotations were originally introduced as a means to handle *comments* in a structured way. From various applications came the need of a more general mechanism to decorate abstract representations. Using annotations for document organization and manipulation is discussed at length in [10]. One important application is to use annotations to store attributes, in the precise sense of attribute grammars [28].

In the VTP, the role of decors and annotations has been extended to yield a general structuring mechanism. It is possible to define decors and annotations for other classes of objects than trees. For a given class of objects, new decors may be dynamically created, and then annotations corresponding to that new decor may be attached to the objects in the class. So annotations are essentially a way to define dynamically and independently optional new

fields or properties. In the VTP, annotations are used as a modularity device to *extend* the functionality of classes without interacting with their already existing semantics.

2.3. *Discussion*

The VTP is a stable and trustworthy component of CENTAUR. Further work on the VTP will take place in several directions: improving error handling, adding new functionalities, improving the overall efficiency.

New functionalities are added sparingly. As indicated above, a more abstract notion of context will be added to the VTP. To keep track of the various trees constructed during a session, a notion of VTP *variable* distinct from Lisp variables will be developed as well. In another area, the mechanism of annotations needs extension, for example so that all annotations in a given decor might be accessed without traversing the annotated tree.

Efficiency is improved on a demand basis. The performance analysis of the VTP is generally difficult, because various processors use the VTP in fairly different ways. Inefficiencies often point to an inadequate choice of the primitive functions for some class. This has motivated adding several iterators to the VTP. Functions performing translations between computational and persistent storage must be improved to be faster and to produce even more compact representations.

3. Using the database

This section presents three components of CENTAUR, with a special emphasis on the way they are using the VTP. The first component is the METAL compiler, that creates a new formalism and a parser/constructor for it. The second component is the PPML compiler, is a tool that generates a pretty-printer from a symbolic description of pretty-printing rules. The generated parser and pretty-printer are both subroutines of a third component, a simple structure oriented editor for abstract syntax trees.

3.1. *The METAL compiler*

METAL is a language used in specifying new formalisms. The METAL specification of a new formalism is made of three parts:

(1) a definition of the *abstract* syntax of the new formalism in terms of *operators* and *phyla*. The operators label the nodes of the abstract trees that represent syntactically valid structures of the defined formalism. An operator is defined by its arity and by the type of its sons. The phyla are non-empty sets of operators. They are used to define the type of the sons of each operator.

(2) a definition of the *concrete* syntax of the new formalism in terms of BNF rules. These rules are used to create a *parser* for the defined language,

(3) a definition of the *tree generator* of the new formalism in terms of *tree building functions*. These functions specify what tree corresponds to each syntactic component of the language, i.e. they connect concrete syntax rules and abstract syntax. A tree building function is associated to each production rule in the grammar and it is invoked whenever the rule is used to parse.

The compilation of a METAL specification creates a formalism (in the precise sense of the VTP), a concrete syntax parser, and a Lisp program that will construct abstract syntax trees, using the VTP primitives, as a program text is being parsed. In fact each component of a METAL specification is compiled with a specialized compiler and it is possible to invoke these compilers separately if desired.

3.1.1. *Abstract syntax compiler*

To define a new language in METAL one has to specify at least an abstract syntax. Then this definition is compiled with the *abstract syntax compiler.* This compiler constructs the internal representation of an abstract syntax, i.e. a VTP object in the class *formalism.* Then it is possible to save the new formalism on external storage, currently as a table.

3.1.2. *Concrete syntax compiler and tree construction*

To compile the concrete syntax part of a METAL program it is necessary to have compiled the abstract syntax part first. Indeed the defined formalism is one of the arguments needed to call the concrete syntax compiler. At the current stage, METAL concrete syntax rules are compiled into rules in a format suitable for input to YACC. An action number is uniformly associated to each tree building function. Additionally the compiler produces a list of all keywords used within the concrete rules. This list is useful in constructing a lexical analyzer using LEX.

To complete the compilation of a METAL specification it is necessary to compile the tree building functions that specify the connection between the abstract syntax and the concrete syntax of a formalism. This compilation again can take place only after the formalism has been created. Tree building functions are compiled into Lisp code that uses standard VTP primitives.

When parsing a program in a language F, for each reduction performed by the parser the associated semantic action is executed and a tree is constructed. In fact this transformation from concrete form to abstract form is achieved through an intermediate data structure. The output of the parser is a list of action numbers[1], where each action number correspond to a semantic action. Then the output of the parser is used by the abstract tree generator to execute the adequate Lisp code that was generated during the compilation of the tree building component of METAL.

3.1.3. *Discussion*

The METAL compiler has been exercised on several large languages, for example Pascal, Ada, and C. It is a satisfactory component of CENTAUR but several major improvements are necessary:

- METAL is too verbose: there is no star notation for lists, routine cases deserve a more compact notation.
- METAL should include lexical analysis.
- A better parser generator than YACC should be used: hard work on syntax rules would be spared, the METAL specification would be simpler.
- The METAL compiler should be more incremental: changing one rule or a few rules should involve little processing.
- The METAL environment should contain more consistency checks.

Another formalism to specify formalisms, called Syntax Definition Formalism, SDF for short, has been designed and implemented by our partners from CWI in Amsterdam [15]. This new component should be integrated with CENTAUR within a short time. In SDF concrete and abstract syntax are defined simultaneously and there are no restrictions on what context-free grammar may be used. Furthermore, the scanners and parsers are built incrementally [16][17]. As a consequence, generating an environment for a new language will be far more comfortable.

[1] It is not a list of production numbers because several productions may in fact invoke the same action.

3.2. *The Ppml compiler*

A special purpose formalism, PPML [25], is used to associate concrete layouts to abstract structures. The formalism aims at specifying in a compact but readable fashion:

- the desired layout for an abstract syntax tree,
- alternate layouts and how to switch from one to another if the page is not wide enough,
- how the required level of detail affects what is being shown,
- what subtree is intended when the user points at a given token on the screen.

Like the METAL compiler, the PPML compiler has been experimented on large languages such as Pascal, Ada, and C.

3.2.1. *Pretty-printing rules*

Roughly speaking, a PPML specification is a list of pretty-printing *rules* of the form

$$pattern \longrightarrow format$$

where the lefthand side of the rule is an abstract syntax tree containing variables, i.e. a pattern, and the righthand side is a formatting specification. The rules corresponding to more specific patterns must be listed before those concerning more general situations. PPML specifications can be organized in chapters, with local constant declarations. A PPML specification imports a language's abstract syntax. To restrict the applicability of rules, all phyla of the language are known, and other special-purpose phyla may be added.

A pretty-printing pattern is either a variable or an arbitrary abstract syntax tree that may contain (non-repeated) variables. A standard first order pattern-matching mechanism is used to select a rule matching the top of a given tree. When a rule is selected, every variable in the pattern is associated to the corresponding subtree. Then on the right hand side of the rule, each occurrence of a variable denotes the result of a recursive call of the pretty printing rules on the subtree. For atomic operators the type of their values is used to call, in an object oriented manner, the adequate pretty-printing method. A special mechanism allows iterating on lists. Variables in the pattern may be restricted to belong to some phylum for the rule to be applicable.

Since pretty-printing may be context-dependent, one can give a name to a set of pretty-printing rules and request that a variable should be pretty-printed using another set of rules.

3.2.2. *Pretty-printing formats*

A box language [5] is used to describe the concrete layout of patterns, and its definition is very similar the definition of a tree structure: a box is either an atomic box or a compound box.

- an *atomic* box is used to represent an individual object and to wrap an imaginary rectangle around this object. In particular an atomic box is used to represent the lexical elements of a language,
- a *compound* box combines several boxes, on the basis of their imaginary surrounding rectangles. It has itself an imaginary surrounding rectangle. Note that, as in typesetting, it is not certain that all pixels fit within the rectangle, which is used as a scaffolding for these constructions.

The combination of the elements of a compound box depends on the *type* of the box and on *separators*. The type of a compound box specifies the position of the components of that box. The set of combinators is not fixed, but the following ones are used often:

1) *h* to lay sub-rectangles side by side horizontally,

2) *v* to pile sub-rectangles with their left side vertically aligned,

3) *hv* to perform as an "h" box, but with automatic folding when the horizontal composition does not fit into the width of the output device,

4) *hov* to align sub-rectangles horizontally if possible, but switch to a vertical composition if the horizontal composition does not fit. This box acts either like an "h" box or like a "v" box.

To define a compound box it is also useful to express *indentation*, for vertical composition, and *spacing* either horizontally or vertically. In the current pretty-printing formalism this is achieved with *separators*. Separators are either global (to a chapter or a set of rules) or local to a rule. Local separators, located between two sub-boxes in a box, are used to override a global separator but only between these two sub-boxes.

A typical PPML rule is shown on Figure 1.

$$\text{program}(\textit{decls}, \textit{stms}) \longrightarrow [< \text{v} > \textbf{begin} < \text{v tab}, 0 > [< \text{v} > \textit{decls} \quad \textit{stms}] \quad \textbf{end}]$$

Figure 1. A typical PPML rule

The lefthand side matches a program and introduces two variables, for the declarations and the statements in the program. The righthand side specifies a vertical box with three components: the keyword **begin**, a middle box, and the keyword **end**. The middle box is indented by tab blank characters, and it is itself a vertical box stacking declarations above statements.

3.2.3. *The generated code*

PPML specifications are type-checked and then translated to Lisp code. The type-checker verifies that no rule is useless (i.e. hidden by a more general rule) and that the specification is consistent with the language definition. It does not check that certain language constructs are omitted; indeed it is convenient to develop a pretty-printer incrementally. When no rule is given for a construct, the system uses by default a generic pretty-printer.

The PPML compiler generates code for all rules and then collects code fragments involving the same top operator. The collected code can then be used in an object-oriented fashion. The code contains calls to an abstract formatting machine so that it is easy to mix directives for pretty printing several languages simultaneously. This is very useful because annotations are almost certainly in a different formalism.

The pretty-printer generates automatically a data structure that is used in locating the mouse and emphasizing the current expression. Once this data structure is created, it is not necessary to call the pretty-printer again merely to select a different subexpression in the document. So pointing in the tree is very fast and, above all, the time needed to select the current expression does not depend on the size of the document nor on the size of the window.

3.2.4. *Discussion*

The Lisp code generated by the PPML compiler is itself compiled, resulting in an efficient interactive display of tree structures. Further optimizations may still improve the speed of the formatting machine.

A good environment for PPML has been generated with CENTAUR. The PPML compiler is incremental: when a rule is modified, only this rule and the rules concerning the same operator are recompiled. Experiments have indicated that a draft version of a PPML specification can be obtained from the specification of syntax. Hence the meta-user can really focus on layout, preferred line breaking, and appearance at all levels of detail. Extremely good results can be obtained in a short time.

Several improvements to PPML and its compiler are not yet implemented:

- Tabular layout is sometimes requested. This implies a two-pass algorithm over some subtrees to calculate the maximum width of a column.
- To pretty-print differently a specific subexpression, one would like to annotate it with a PPML rule, to override locally the general pretty-printing rules.
- The algorithm that chooses the final layout needs, in some cases, to have additional look-ahead.
- A more pleasant way of defining new formatting combinators is needed.

A more significant demand is to have identifiers printed with a font that depends on their type. This implies complete overloading resolution and type-checking at pretty-printing time and points to a specification in the formalism used for static semantics, TYPOL, rather than in PPML. Given the performance of the system today, it seems feasible to accomodate all the improvements above and still keep a good interactive response time.

3.3. *The prototype editor*

The construction of a smooth, natural structure-oriented editor, where structural and textual manipulation are blended effortlessly is one of the major challenges of CENTAUR. But to get a sense of direction, an initial prototype has been constructed first. The main functionalities of the prototype are reviewed now.

3.3.1. *Displaying trees*

To observe trees, one creates views in which the trees are pretty printed. The pretty-printer creates an image that depends on the width and height of the view, and various pretty-printing parameters such as the desired level of detail. If a given tree does not fit completely in the view, the view can be moved over the tree. Several views may look at the same tree at different locations. Typically one wants to create a view that will *focus* on a subtree to keep that subtree on the screen while one peruses other parts of the tree. Modifications performed in one view are of course reflected in all views that look at the same tree.

A view uses two VTP contexts. The first context designates what location in the tree is shown in the view. The second context designates the *current subexpression* of interest. The current subexpression is the primary *subject* of editing commands and it is usually printed in boldface. When the rightmost mouse button is depressed, a language-dependent menu pops, based on the language of the current expression. Different views on the same tree have *a priori* independent current subexpressions. But another layer in the interface, the logical control, may constrain two contexts to remain equal or related in some other fashion.

3.3.2. *Designation and navigation*

The normal way of designating a location in a document is to use the mouse. The pretty-printer builds an auxiliary data structure so that when the leftmost button of the mouse is depressed (usually with the index finger) the corresponding subtree (or rather *context*) is located very quickly. The former current subexpression returns to normal and the new current expression is emphasized. If the user drags the mouse, while the button is depressed, then

the new current expression is the smallest context containing the subtree that was designated initially, when the mouse button was first depressed, and the subtree designated by the current position of the mouse cursor. Dragging is a convenient way to designate sublists. It is also useful in correcting an initial designation that was slightly off the mark.

A pop-up menu provides more sophisticated navigation primitives, primarily for long-distance moves. Simple moves that are guided only by the syntactic structure of the language are programmed in Lisp using the pattern matching facility of the VTP. More complex moves are derived from a semantic specification: they rely on the logical machinery of CENTAUR.

3.3.3. *Editing*

Editing functions are ordinarily invoked with the mouse. Some users find control keys faster, so all functions can be bound to keys as well. One may want to type in new text for a subexpression or if it is faster (and safer) to replace it by an existing subexpression. One may want to insert or delete elements in lists. All these operations are valid only if they respect the abstract syntax. By convention, the *current expression* is the subject of commands. When a tree is replaced or deleted, it is moved to the *clipboard*. Commands that need two arguments (e.g. *insert*) take the current expression in the clipboard as second argument. After performing the required action, each command decides what is the new current expression. After a change or an insertion, the current expression denotes the *newly introduced* subtree. When the current expression denotes a list element that has vanished, the next element in the list is the convenient choice for resetting. When the last element of the list is removed, then one resets the current expression to the new last element.

All simple commands are simple combinations of VTP primitives. Using contexts is particularly useful to manipulate sublists in manner that is cogent with other subtrees. Textual input is parsed and the corresponding tree is constructed using VTP primitives. When parsing fails, the textual input is still available for textual editing and reparsing. In the current prototype, text is located in a separate input window but it will be overlayed with the *view* in the future.

It is necessary to add language-dependent editing commands. The commands that rely solely on syntax are programmed in Lisp using the primitives of the VTP. More sophisticated commands must use the logical machinery of CENTAUR.

3.3.4. *Discussion*

The VTP contains adequate primitives to construct a structure-oriented editor. For example, adding a menu-driven mode of input, in the style of the Cornell Program Synthesizer [30], is the matter of one or two pages of code. When additional objects are needed, the VTP may be extended easily. The real difficulty is in designing a man-machine interface that is convenient, intuitive and fast.

4. The logical machinery

A method for specifying the semantic aspects of languages is included in CENTAUR, so that the system is not restricted to manipulations that are based solely on syntax. This section describes the TYPOL language and its compiler, and it shows how it is possible to take advantage of semantic descriptions in an interactive environment.

4.1. *Specifying Semantics with TYPOL*

TYPOL is an implementation of Natural Semantics [19]. A TYPOL program is roughly a collection of axioms and rules of inference. The next paragraphs present quickly the principal structures in TYPOL.

4.1.1. *Rules*

A semantic definition is an unordered collection of rules. A rule has basically two parts, a numerator and a denominator. *Variables* may occur both in the numerator and the denominator of a rule. These variables allow a rule to be instantiated.

The numerator of a rule is again an unordered collection of formulae, the *premises* of the rule. Intuitively, if all premises hold, then the denominator, a single formula, holds. Formulae are of two kinds: *sequents* and *conditions*. The conclusion of a rule is necessarily a sequent. On the numerator, sequents are distinguished from conditions, that are placed slightly to the right of the inference rule. Conditions convey in general a restriction on the applicability of the rule: a variable may not occur free somewhere, a value must satisfy some predicate, some relation must hold between two variables. Conditions are boolean predicates built from atomic conditions with the help of logical connectives. One may wish to axiomatize atomic conditions, for example in a separate set of rules.

A sequent contains two predicates separated by the turnstile symbol $\vdash$. Predicates come in several forms, indicated by various infix symbols. These infix symbols carry no reserved meaning, they just help us in memorizing what is being defined. The first argument of the predicate on the right is called the *subject* of the sequent. By extension, the subject of a rule is the subject of its conclusion.

A rule that contains no sequent on the numerator is called an *axiom*. Thus an axiom may be constrained by a condition. A typical TYPOL rule is shown on Figure 2.

$$\frac{s \vdash \text{ID} \Rightarrow x \qquad s \vdash \text{EXP} \Rightarrow v \qquad s \overset{\text{update}}{\vdash} x, v \Rightarrow s_1}{s \vdash \text{ID} := \text{EXP} \Rightarrow s_1}$$

Figure 2. A typical TYPOL rule

The subject of this rule is the assignment ID := EXP, and indeed the rule explains how to execute an assignment statement.

In a single semantic definition, sequents may have several forms depending on the syntactic nature of their subject. For example, in a typical Algol-like language, there are declarations, statements and expressions. The static semantics will contain sequents of the form: $\rho_1 \vdash \text{DECL} : \rho_2$, $\rho \vdash \text{STM}$, and $\rho \vdash \text{EXP} : \tau$. The various forms of sequents participating in the same semantic definition are called *judgements*.

Some structure must be introduced in a collection of rules, if only to separate different semantic concerns. For example, in static semantics, one wishes to distinguish *structural* rules that state what is a consistent use of types from management of *scope* and the properties of *type values*. To this end, rules may be grouped into sets, with a given name. Sets of rules collect together rules or, recursively, rule sets. When one wishes to refer to a sequent that is axiomatized in a set of rules other than the textually enclosing one, the name of the set is indicated as a superscript of the sequent's turnstile.

4.1.2. *Assigning types to rules*

Semantics tells us facts about the constructs of a language. These constructs taken together form the abstract syntax of the language. Each construct has arguments and results belonging to syntactic categories, and some syntactic categories may be included in others. The declaration **use** L indicates that a definition is concerned with language L. When specifying a translation, two languages are involved and it is necessary to import two abstract

syntax definitions. When semantic rules are analyzed mechanically, for example in the TYPOL compiler, the various abstract syntax constructors are recognized.

Abstract syntax terms occur in most rules. They have to be valid terms w.r.t. their abstract syntax. Every such term is typed with a syntactic category (such as L.*expression*, or L.*statement*, or L.*declaration*). A language L includes all of its syntactic categories, and it is possible for two languages to share a given syntactic category. For example PASCAL and MODULA can share the category *expression*.

The scope of variables is limited to the rule where they appear. Nevertheless, it is necessary to follow certain naming conventions to make a definition readable. For example, we want to assert that variables called ρ, possibly with indices or diacritical signs, are environments. For this we use variable declarations. The scope of such declarations is the set of rules where they appear. It is not necessary to declare all variables, because often their type may be inferred. Variable declarations and abstract syntax definitions serve then in typing sequents.

4.2. *The TYPOL compiler*

A TYPOL specification must be compiled before it can be used in CENTAUR. The compiler includes a type-checker and a code generator. Both have been written in TYPOL itself.

4.2.1. *Generating Prolog code*

Given a semantic specification, we want to use the computer to solve various equations on sequents. For example, given state s_1 and program E, is there a state s_2 s.t. $s_1 \vdash \mathrm{E} \Rightarrow s_2$ holds? Or given an initial type-environment ρ_0, is it possible for expression E to be given a type τ such that $\rho_0 \vdash \mathrm{E} : \tau$ holds? Typical *unknowns* are type values, states, or generated code, however environments or program fragments may also be unknown.

To turn semantic definitions into executable code, there are probably many approaches. The TYPOL compiler transforms rules into Prolog code, taking advantage of the similarity between Prolog variables and variables in inference rules. Roughly speaking, the conclusion of a rule maps to a clause head, and the premises to the clause body. Distinct judgements map to distinct Prolog predicates. Conditions, although written to the right of rules, are placed ahead of the rule body.

An equation is turned into a Prolog goal. Since pure Prolog attacks goals in a left to right manner, proofs of premises will also be attempted from left to right. This is not always reasonable, so we need to use a version of Prolog that may postpone attacking goals until certain variables are instantiated. Conditions should be evaluated as soon as possible to avoid building useless proof-trees. In CENTAUR, we use the delaying facilities of Mu-Prolog [23] to achieve that effect.

4.2.2. *Actions*

It is useful to attach actions to rules, in a manner that is reminiscent of the way actions are associated to grammar-rules in YACC [18]. Actions are triggered only after an inference rule is considered applicable. An action needs to access the rule's variable bindings, but it does not interfere with the deduction process. In particular, an action cannot introduce new bindings and a rule cannot fail because an action failed.

Typically, actions are used to emit messages and to perform a variety of side effects. It is important to understand that searching for a proof may involve backtracking, so that if an inference has been used in a computation at some point, it does not necessarily participate in the final proof.

In terms of style, actions should be used with parsimony. For example, when specifying a translation, it is mandatory to *axiomatize it* rather than have actions generate output code. On the other hand, in the context of type-checking, it is more appropriate to have actions filter error messages, rather than introduce strange type values to handle various erroneous situations.

Actions are normally written in Lisp and it is convenient to have the possibility of changing dynamically the action associated to a TYPOL rule.

4.3. *Using semantic specifications in CENTAUR*

4.3.1. *Operating the logical engine*

The TYPOL compiler generates Prolog code. To use this code from CENTAUR, a goal must be constructed and mailed to the Prolog engine. Goals usually have an abstract syntax tree as argument. Since trees are VTP objects in CENTAUR, it is necessary to coerce tree objects to Prolog terms. A generic routine is included in the VTP to perform this coercion quickly. The routine uses directly the primitives of the Prolog implementation that construct Prolog terms. Conversely, the result of a computation, for example in a translation, may be an abstract syntax tree represented as a Prolog term. A second generic routine converts such terms into VTP objects, using directly primitives of the Prolog implementation to traverse terms and the primitives of the VTP to construct the corresponding VTP object. During these coercions, Lisp and the Prolog implementation communicate via a common storage area to speed up parameter passing and to avoid unnecessary *cons*ing.

When the arguments for the goal are ready, the goal is "mailed" to Prolog for resolution. Code generated by the TYPOL compiler before and after the code of inference rules givethe ability to control precisely the Prolog execution from CENTAUR. A simple automaton, the *semantics manager*, receives on the one hand indications of what the Prolog engine is doing (trying to use a rule, succeeding or failing with the rule, backtracking on a subgoal), and on the other hand instructions, usually via the man machine interface, on how to proceed (step, continue till breakpoint, step back). Such instructions are translated into a succession of elementary replies of the semantics manager to the Prolog engine: *continue* to let Prolog execution go on normally, *fail* to force Prolog to fail and backtrack, *retry* to have Prolog go forward again, *abort* to abort Prolog execution, and *skip* to proceed nonstop until the rule that is being tried succeeds or fails.

The semantics manager has access to the complete environment of the current rule: the rule itself, its subject, and all variables of the rule. For example it may check whether a breakpoint has been set on the subject, or on the rule itself; it may test the value of a variable, prompt the user, and then finally decide how to proceed.

4.3.2. *An example*

The description of the environment built for a very small language, ASPLE [9], illustrates the operation of the logical engine. First the METAL and PPML description of ASPLE had to be constructed and compiled. Then a type-checker and a front-end processor written in TYPOL were added. Both can be invoked by selecting a menu item. The front-end processor constructs a new program in a so-called *deep* abstract syntax where all overloading has been eliminated and type information has been moved to places where it is needed at run-time, in this case only input-output statements. The dynamic semantics of ASPLE is defined on the deep abstract syntax trees. The execution of the dynamic semantics yields an interpreter. The semantics manager controls the interpreter. The user is given two classes of options:

1. It is possible to follow graphically the subject of the next rule, to display the semantic rules as they are being used, and to display the values of all variables in the program.

2. It is possible to control the inference mechanism, through the semantics manager. Breakpoints can be attached as annotations to the source code. Execution will stop when the subject of a rule carries such an annotation. Analogously, breakpoints may be attached to *semantic rules*. When an attempt is made to use such a rule, computation will halt. The user may step one inference at a time or run to the next breakpoint. It is possible to execute the next goal so that in fact an arbitrary granularity in the execution is achievable in a simple way.

All these operations can be *undone* with equal ease: stepping back, undoing the last goal, racing back to the previous breakpoint. Breakpoints may be removed or ignored.

4.3.3. *Transforming TYPOL programs*

In the example above, the TYPOL specification was directly compiled and executed. If one is ready to analyze and transform semantic specifications, very interesting and useful behaviors can be obtained.

A first example concerns type-checking. Executing directly the rules that characterize well typed programs results only in a "pure" type-checker, that either succeeds or fails. A more useful tool will emit appropriate diagnostics and proceed with type-checking beyond the first error. Such a tool is obtained by mechanical transformation of a TYPOL type-checking specification [13]. All causes of failure are examined and replaced by operations that always succeed but emit error messages. To handle errors in declarations, an idea from polymorphic type-checking is used: variables are declared of an unknown symbolic type that is refined as more information is collected. Further transformations lead to an incremental type-checker. A by-product of this analysis is a program that leads from an identifier to its declaration.

Another tranformation is very useful in the context of a *translation* specified in TYPOL. From the specification of the translation, one can deduce mechanically a *program* correlating tree addresses in the source code and tree addresses in the target code. Using this correlation, one may execute the target program and animate the source. When the correlation might be recomputed over and over, for example in tight loops, it is reasonable to use a *cache*.

4.4. *Discussion*

A good TYPOL environment has been generated with CENTAUR and many experiments have been carried out. The TYPOL type-checker is very useful. When writing semantics in TYPOL, one can focus on the true difficulties of the language to describe. The TYPOL debugger is an essential component of the system, because following Prolog execution is rather more difficult than following machine code. In all fairness, no truly large languages have been worked on yet: more understanding could be gained, at this stage, from small difficult examples rather than from sheer accumulation of rules.

The interpreters derived from the formal specifications are sufficiently fast for an interactive use, and using a Prolog compiler [31] will make them even better in this regard. The ability to execute in reverse is very pleasant and instructive. But it indicates that the generated interpreters –even though structure-sharing minimizes memory waste – take up much space. Methods to control the use of storage, and in general to supervise TYPOL execution are under investigation. As we begin to tackle parallel languages, such control becomes essential.

TYPOL has been mostly used thus far to generate interpreters/debuggers and translators, but novel uses are being considered. For example, a partial evaluator may be deduced from a semantic specification [14].

Compiling into Prolog is very convenient, but it is not the only possible strategy. If TYPOL rules are analysed in greater detail, functional subsets may be discovered. TYPOL can then be compiled to Lisp, which will not affect speed as much as storage use.

5. The man-machine interface

In the previous sections we have presented two major components of the CENTAUR Environment Generator, the data-base kernel and the logical machine, together with their interfaces and specification formalisms. Of course, a third essential component of CENTAUR is the user interface. At a very general level the user interface of a system contains all the components for the user to modify the state of the system and for the system to inform the user about its state. The state of the CENTAUR environment is a collection of structured objects that the user and/or specific processors can operate on. Simple operations involve only one object: this is the case for example when editing. But often several objects in different formalisms are involved, for example when translating from source into code. This section describes the architecture of the CENTAUR user interface and presents its main aspects.

5.1. *Design Goals*

As an interactive environment the CENTAUR system must be able to receive its input from the user, through a keyboard and a mouse, and to return its output to the user, through various devices. But components of CENTAUR may also be very useful when integrated in some specialized tools, such as compilers. In these tools one needs only a collection of primitives to access the various structures of the system, rather than a man-machine interface. The first design goal is therefore to separate very carefully the man-machine interface from the system itself. This implies an object-oriented architecture.

Our second design goal is to keep the man-machine interface open-ended in two directions.

- A rich set of graphic objects has to be available to begin with, but any language-specific environment may require entirely new graphic objects. Even a final user may want to define his own, that make sense for his problem. Hence the man-machine interface should contain primitives and tools to construct new graphic objects.
- Certain generic functionalities are available in all environments. Other operations are specific to a particular language, or to a particular user. The design of the man-machine interface must make it easy to choose graphic objects and connect them to the desired functionalities.

5.2. *Graphic Objects*

The CENTAUR interface provides a library of standard graphic objects such as buttons, menus, pop-up menus, menubars, scrollbars, etc. But it also provides a *framework* to define specialized graphic objects. For example, a special graphic object called a *view* is used to display the abstract syntax trees manipulated by CENTAUR. A view usually has a title bar, a scrollbar, a resize corner; in its main area – the view proper – the three buttons of the mouse have a distinct meaning; the menu that pops up when the rightmost button is depressed depends on the language of the current expression; a special data structure helps in finding what subexpression the user means when the leftmost button is depressed; finally certain keys of the keyboard are bound in a different way in this area.

To define graphical objects we specify *separately* their appearance and their behavior. A set of Le_Lisp primitives, the AIDA package [8] is used to describe the geometry of the objects. These include basic constructors such as string, point, rectangle, box, etc., and geometric combinators such as row, column, overlay, etc. Compound objects are built using these primitives. To obtain a graphical object, one merely associates a window to such an object. It is necessary to associate a behavior to some graphical objects. This is the case with a button, a menu, a menubar, a scrollbar, etc. These objects are seen as reactive systems that change state in response to input events (from the mouse and keyboard for example) and also generate output events. The behavior of all objects is written in ESTEREL [2], a special purpose language for real-time applications. The ESTEREL compiler generates a Lisp

description of a finite automaton. The automaton is hooked in a systematic fashion to input and output events. Graphic objects are presented in more detail in [4].

The graphical interface of the CENTAUR system is built on top of the Le_Lisp virtual window system. This underlying system provides primitives for creating and modifying windows and subwindows together with primitives to handle both keyboard and mouse inputs. The purpose of the virtual Le_Lisp window system is to isolate the Lisp programmer from differences at the level of the window system used in the operating system. Indeed, several bindings of the virtual Le_Lisp windows exist, with various kind of window managers (X-Windows [12], SunView [29], BWE [26], etc.).

5.3. *Functionalities*

Any program using the kernel of CENTAUR can be made available to the man-machine interface. Hence in this section, we describe only generic operations that are readily available for other applications.

A first collection of operations concerns *navigation* in the object, or in a data base of objects. For all local navigation, the mouse is used as described earlier. To navigate through large documents, several mechanisms are available:

- An integer controls the level of detail for what is shown in the view. This integer is adjusted with trill buttons.
- A *focus* operation creates a new view on the current expression. This new view shows it automatically in much greater detail. Of course, this mechanism can be iterated. The current expressions of all the windows created in this way are *independent*, so that it is very easy to keep on the screen fragments of a document that are far apart.
- A view can be moved forward and backward over a document.
- New views on other objects, for example other modules, can be opened at will.

For long distance moves, the mouse is insufficient. The navigation menu provides associative searching in both directions, and it is extended with any navigation primitive that makes sense for semantic reasons: go to the surrounding procedure, go to the declaration of that identifier; or in rule based formalisms such as METAL, PPML, or TYPOL, find the rule that handles a particular construct.

Other manipulations are destructive, such as editing. Here we want to provide for both the copy-cut-paste style of editing of a structured object as well as *in-place* textual editing. The copy and cut operations require selecting subexpressions while the paste operation requires positioning in the structured object. For in-place text editing a fragment of the structured object is transformed into a text string. This functionality is still under development and in the current prototype, a separate text window pops-up. There the text is edited *à la Emacs*.

The basic functionalities are implemented in a straightforward manner on top of the VTP. This includes selecting, cutting, pasting of subexpressions, as well as reading and saving structures. All operations maintain at least the structural consistency of an object. That is, it is not possible to replace a subexpression with another of an incompatible syntactic type.

Some operations require that the structure first be coerced into another form. This is necessary in the case of textual editing. The structured object must first be transformed into a string and after editing the string must be analyzed to reconstruct a structured object. When an operation is derived from a semantic specification, it will be performed by the logical engine, i.e. under the control of Prolog. This means a coercion from VTP trees to Prolog terms and conversely. It is important to keep the time spent in these coercions under strict control.

5.4. *User Interface*

From the user's standpoint, the interface is a multi-window system that is menu-driven. Through this system the user can manipulate various types of structured objects. To look at an object one opens a "view" on the object. It is possible and useful to have several views on the same object. Each view has its own emphasized subexpression, and possibly its own pretty-printer for the object. Views are obviously aware of the type of the object they contain, as different menus pop-up.

For manipulating the objects in views there are menus, both pop-up and pulldown, as well as menubars. These graphic objects give access to the standard generic manipulations (e.g., cut and paste, reading and storing objects, elementary navigation, etc.). But the commands available depend on the type of the current expression, in the current view. Menus are dynamically extended for different environments. Likewise keyboard bindings may be a little different in each view. Dialog boxes allow the system to query the user for input and allow the user to parameterize and control the system. Various kinds of dialog boxes appear on the screen, using several types of graphic objects: buttons, browsers, text-editable areas.

When one has several views looking at the same object clearly a modification done in one view is reflected in all the views. A difficulty arises when the user makes a destructive modification in one view while another view was looking solely at a part of the subexpression that was modified. In this last view the display must be adjusted to show some valid part of the object. To keep a consistent screen across such manipulations, a "look-at" relationship is maintained between views and structured objects.

5.5. *Discussion*

The performance of the graphical objects is good; that is, response to mouse and keyboard input is reasonably fast and one can work with the system. Once all the basic objects are in place it may be necessary to optimize the interface with the underlying window manager. Further development of graphic objects and primitives is needed, in particular to improve the appearance of the objects themselves and to provide a more symbolic way to specify their construction.

Basic operations (functionalities) are acceptable but must be made faster. This requires optimizations in the VTP (for example to access persistent objects) and a less costly interface with parsers.

In another area, it may seem surprising that CENTAUR has no command language. In fact, we use Le_Lisp as a command language, and a special window called TopWindow containing a Lisp top level is always present on the screen. The primitives of the VTP and the framework for graphic objects are available at all times. For debugging purposes, direct access to a Prolog top level can also be obtained. There are two reasons for postponing the design of a command language:

- First, it is necessary to gain experience with the system before designing such a language
- Second, what seems to be lacking is a language to express queries, in the sense of data-base queries, rather than a conventional command language.

Operations that involve relations between structured objects and windows seem a little slow. However, further experiments are needed before addressing performance issues. Maintaining screen consistency and maintaining a network of relationships between objects can take many forms. For example, the user may want to specify relations to be maintained between two views, such as always displaying the same current subexpression. As well, implicit relations exist between objects such as a source program, its code, and the translator that was used to compile the source into the code. Thus a general mechanism must be provided for allowing this type of screen/state control. This mechanism relies on knowing which views

look at which objects and checking what relations must be maintained. At this time, the logical engine of CENTAUR seems well adapted to build an experimental system for this *logical* control of the interface.

6. Conclusion

The construction of CENTAUR is not complete. Very few selected users have had actual experience with the system. Still, it is sensible to examine the system and see how it meets the challenges presented in the introduction.

i) Is it possible to use as the description of a programming language a formal specification in the sense used by researchers in programming language semantics?

On the basis of our experience with many sample languages, extracting useful behaviours from a formal definition is in fact *surprisingly easy*. As an example, handling reverse execution involves a minute programming effort. More work is necessary in the *analysis* of formal definitions, a very promising research area. Also, practically all our examples are sequential programming languages and experiments with parallel languages are just beginning.

ii) Is it possible to obtain on that basis a efficient system, both in terms of speed and in terms of memory space?

The answer to this question is subjective: what is efficient for one user may be considered inacceptable in another context. The main source of skepticism about the efficiency of CENTAUR stems from the use of Prolog as a logical engine. First, experience has shown that Prolog used as a coroutine in CENTAUR is a remarkably flexible tool to experiment with and, in fact, often suggests novel approaches. Second, provided it is used only when appropriate, even an interpreted Prolog is *fast* – i.e. comparable to compiled Lisp. However, it is not very easy to control the use of space in CENTAUR and much effort in this direction is necessary in the future.

iii) Is it possible to create a man-machine interface that is comparable in convenience to what can currently be found in dedicated commercial systems?

It would be preposterous to answer affirmatively today. In fact, users of CENTAUR will be the judges. But two remarks can be made. First, building a good man-machine interface is an extremely interesting and rewarding task: the software architecture is non-trivial, the human factors questions are challenging, the logical control of the interface is novel. Second, the literature in this area seems to be lagging behind the commercial achievements. As a result, it is easy to make errors and to underestimate the manpower needed to obtain satisfactory results.

Acknowledgments

Beside the authors of this paper, several people were instrumental in the design and implementation of CENTAUR. L. Rideau-Gallot worked on the virtual graphics interface of Le_Lisp, first with BWE then with X-Windows Version 10.4. Thanks to her, CENTAUR has a modern appearance and good graphics performance. J. Despeyroux and L. Hascoët, intensive and unconventional users, suggested many improvements. F. Montagnac organized a good software development environment and gave critical assistance in all Unix related matters. M. Devin of ILOG was willing to discuss AIDA at length; his work helped establish the man-machine interface of CENTAUR on solid ground.

Throughout the development of CENTAUR, we have had numerous discussions with the CWI group led by Paul Klint, to prepare for the integration of both efforts in a single system.

REFERENCES

[1] BANCILHON F., P. BUNEMAN, *Proceedings of a Workshop on Database Programming Languages*, Roscoff, France, September 1987. To appear.

[2] BERRY G., P. COURONNÉ, G. GONTHIER, "Synchronous Programming of reactive systems: an introduction to ESTEREL", in *Proceedings of the First France-Japan Symposium on Artificial Intelligence and Computer Science*, Tokyo October 1986, North-Holland.

[3] CHAILLOUX J., M. DEVIN, J-M. HULLOT, "Le_Lisp, a portable and efficient Lisp system", *Proceedings ACM Symposium on Lisp and Functional Programming*, Austin, Texas, 1984.

[4] CLEMENT D., INCERPI J., "Graphical Objects: Geometry, Graphics, and Behavior", *in preparation.*

[5] COUTAZ J. " The Box, a layout abstraction for User Interface Toolkits", CMU Report CMU-CS-84-167, Pittsburgh PA, 1984.

[6] DESPEYROUX T., "Executable Specification of Static Semantics", in *Semantics of Data Types*, Lecture Notes in Computer Science, Vol. 173, Springer-Verlag, June 1984.

[7] DESPEYROUX T., "Spécifications sémantiques dans le système MENTOR", Thèse, Université Paris XI, 1983.

[8] DEVIN M. ET AL., "Aida: environnement de développement d'applications", ILOG, Paris, 1987.

[9] DONZEAU-GOUGE V., G. KAHN, B. LANG, "A complete machine-checked definition of a simple programming language using denotational semantics", IRIA Research Report 330, October 1978.

[10] DONZEAU-GOUGE V., G. KAHN, G. HUET, B. LANG, J-J. LÉVY, "Programming environments based on Structured Editors: the MENTOR experience", in *Interactive Programming Environments*, D.R. Barstow, H.E. Shrobe and E. Sandewall (Eds.), McGraw-Hill, 1984.

[11] DONZEAU-GOUGE V., G. KAHN, B. LANG, B. MÉLÈSE, E. MORCOS, "Outline of a tool for document manipulation", *Proceedings of IFIP Congress 1983*, Paris, North-Holland.

[12] GETTYS J., R. NEWMAN, R.S. SCHEIFLER, "Xlib – C Language X Interface, Protocol Version 11", MIT project Athena, February 1987.

[13] HASCOET L., "Transformations automatiques de spécifications sémantiques. Application: un vérificateur de types incrémental" Thèse, Université de Nice, March 1987.

[14] HASCOET L., "Partial evaluation with Inference Rules", September 1987. To appear in New Generation Computing.

[15] HEERING J., P.R.H. HENDRIKS, P. KLINT, J. REKERS, "SDF Reference Manual", Centre for Mathematics and Computer Science, Amsterdam, 1987.

[16] HEERING J., P. KLINT, J. REKERS, "Incremental generation of lexical scanners", Centre for Mathematics and Computer Science, Amsterdam, 1987.

[17] HEERING J., P. KLINT, J. REKERS, "Incremental generation of parsers", Centre for Mathematics and Computer Science, Amsterdam, 1987.

[18] JOHNSON S.C. "Yacc: Yet Another Compiler Compiler", Unix Programmer's Manual, 7th Edition, Bell Telephone Laboratories, January 1979.

[19] KAHN G., "Natural Semantics", *Proceedings of STACS 1987*, Lecture Notes in Computer Science, Vol. 247, Springer-Verlag, March 1987.

[20] KAHN G., B. LANG, B. MÉLÈSE, "METAL: a formalism to specify formalisms", Science of Computer Programming, Vol. 3, pp. 151-188, North-Holland, 1983.

[21] LANG B. "The Virtual Tree Processor" in *Generation of Interactive Programming Environments*, Intermediate report, J. Heering, J. Sidi, A. Verhoog (Eds.), CWI Report CS-R8620, Amsterdam, May 1986.

[22] MILNER R. "A proposal for Standard ML", *Proceedings of the ACM Symposium on Lisp and Functional Programming*, Austin, Texas, August 1984.

[23] NAISH L., Negation and Control in Prolog, Lecture Notes in Computer Science, Vol. 238, 1986.

[24] NAISH L., "The MU-Prolog 3.2 Reference Manual", Technical Report 85/11, Department of Computer Science, University of Melbourne, October 1985.

[25] MORCOS-CHOUNET E., A. CONCHON, "PPML, A general formalism to specify pretty-printing", *Proceedings of IFIP Congress 1986*, Dublin, North-Holland.

[26] PATO J.N., S.P. REISS, M.H. BROWN, "The Brown Workstation Environment", Brown University CS-84-03, October 1983.

[27] PEREIRA F., D. WARREN, D. BOWEN, L. BYRD, L. PEREIRA, "C-Prolog User's Manual, version 1.5", SRI International, Menlo Park, June 1985

[28] REPS T., "Generating Language Based Environments", Ph.D. Thesis, Tech-Report 82-514, Cornell University, Ithaca NY, August 1982.

[29] *Sun View Programmer's Guide*. Sun Microsystems, Inc. Mountain View, California, February 1986.

[30] TEITELBAUM T., T. REPS, "The Cornell Program Synthesizer: a syntax-directed Programming Environment", Communications of the ACM, vol. 24 (9), September 1981

[31] THOM J.A., J. ZOBEL "The NU-Prolog 1.1 Reference Manual", Technical Report 86/10, Department of Computer Science, University of Melbourne, October 1985.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique