Proc. Conf. on Integrated Project Support Environments, Univ. of York, org. by Alvey Directorate and Institution of Electrical Engineers April 1985. Published as : *Integrated project support environments* John A. McDermid (ed.), Peter Peregrinus Ltd, IEEE Software Engineering Series Vol. 1, ISBN:0-863-41050-2, pp. 175-188, 1986.

Mentor - Design and Implementation of the Kernel of a Program Manipulation System ‡

Bernard Lang

INRIA

0.1. INTRODUCTION

Early attempts at building systems for proving, transforming or deriving programs, or even other symbolic manipulation systems, have often bogged down into what has been sometimes called the syntactic quagmire. The development of the semantic parts of these systems was limited or complicated by the inadequacy of the tools available to manipulate whatever representation of programs was being used.

It was then recognized that, in order to build an advanced system dealing with the semantics of programs (and more generally with the semantics of formal systems), one has first to attend the more mundane (and sometimes tedious) task of providing a well engineered collection of syntactic facilities needed to represent and manipulate programs.

The original purpose of the Mentor system was precisely to be such a syntactic facility, with the added purpose that the system should be usable for life-size program manipulations rather than for experiments on small examples. Although the ultimate aim was still to build a semantics based program manipulation tool, this syntactic component turned out to be at the same time sufficiently challenging to become a research project in its own right, and sufficiently powerful to be used as a program development system even without the adjunction of semantic components.

Mentor is not an integrated project support environment (IPSE). Its range of applications goes beyond mere syntax directed editing to a variety of programming support activities (program transformations, transport, preprocessors, documentation, etc.), but it is essentially limited to manipulations of syntactic representations of programs. The systematic integration in Mentor-like systems of tools based on semantics is still the object of ongoing research [1,4,29] and does not seem yet mature enough for an industrial production system. Specific tools (e.g. compilers) could be to some extent integrated on an ad hoc per case basis; we have not attempted this mainly for reasons of manpower and portability. Finally Mentor does not support team oriented activities such as maintenance of a program data base, communications between users or project management.

However, as was initially intended, the concepts and techniques developed for Mentor can provide a sound and uniform basis for all syntactic aspects of an IPSE. The object of this paper is to present these concepts as originally developed for Mentor in its first implementation in Pascal, to discuss implementation issues raised by the use of this language, and finally to present a new implementation in Lisp including only the kernel functionnalities of Mentor, which is intended to be the

‡ The Lisp implementation reported in the second part of this paper has been partly supported by contracts from the Concerto Project of the Centre National d'Etudes des Télécommunications, and by the Commission of the European Communities within the Esprit Project 348 (Generation of Interactive Programming Environments).

syntactic component of an IPSE developed by the Concerto Project of the Centre National d'Etudes des Télécommunications [2].

0.2. THE MENTOR SYSTEM

Mentor is an extensible structured document manipulation system [5,6], intended to serve as a basis for the development of software environments. The main motivation for this approach is the fact that most of the activities of a software project can be viewed as the creation, modification, analysis and maintenance of a variety of related documents: requirements, models of the solution, specifications, code, tests, documentation, user manual, etc... Supporting these activities entails basic choices concerning the following issues:

- the representation of documents,
- the document manipulations primitives,
- the organization of the collection of project related documents,
- the (partial) mechanization of document manipulation activities,
- the design and organisation of tools for the analysis, manipulation or transformation of documents.

0.2.1. Brief history of Mentor.

The implementation of Mentor was started in 1974. The language Pascal was chosen both as the implementation language, and as the object language, i.e. the language of the programs to be manipulated by the system. This choice had the advantage of allowing us to experiment our system on our own programs, and in 1977 we started using Mentor for its own development. In 1981, the first version of the Metal language was designed and Mentor became language independant. Later improvements include an operational full screen interface and extension of the annotation facility.

The system was first developed on a CII 10070 computer, then on a CII IRIS-80, and finally on a CII-HB DPS-8 under Multics. It was tranported at various stages of development on several other machines, and is currently maintained and distributed on DPS-8 under Multics, on VAX-780 under Unix [1] Berkeley 4.2, and on the SM-90 (a Motorola 68000 workstation) under SMX (a version of Unix V7). It is being transported on VAX VMS, on HP 9000, on Data General MV10000 and on the Apollo workstation. It is now running on about 50 sites, of which about 20 are outside France, and it is used in several pilot or Esprit projects.

The current implementation represents approximately 40 000 lines of Pascal code (compiled into about 350 K-bytes of object code), plus several thousands lines written in Mentor's own languages (Mentol and Metal). Mentor itself is used for the source code transformations required by the transports, and the simultaneous maintenance of the distributed implementations.

0.2.2. Representation of documents.

Along with other structure oriented systems [13] Mentor uses a tree structured representation (traditionally called **abstract representation**) of documents instead of the more traditional string representation.

The abstract representation has several advantages:

- It emphasizes the internal structure of documents (especially, but not only, for programs) and supports more effectively their manipulation in terms of this structure. Actually, many traditional tools perform a preliminary syntax analysis to translate documents into a more processable form. This ubiquitous syntax analysis is factored out by using directly the abstract representation.
- It defines a unique standard representation of documents, independant of any concrete physical support and layout, and thus provides a uniform interface between the processors in the

[1] Unix is a well known trademark of Bell Laboratories.

environment. One may even have several concrete representations (for example a program and its flowchart, or a VLSI layout description and its graphic representation) corresponding to a unique abstract one.

- When working *directly* on a common standard representation, the tools may also share the primitives that manipulate it, thus factoring out another component of traditional tool design.
- Tree representations of formal documents (e.g. specifications, programs, proofs) are a long time accepted basis for formal mathematical definitions. Thus this approach is a good step towards the use of the formal techniques, on which advanced software engineering tools will be based in the future.

The abstract representation of a document is a labelled tree. The label of each node corresponds to a syntactic construction of the language (or formalism) used in the document. Figure 13.1 presents a fragment of a Pascal program both in concrete textual form and as an abstract labelled tree.

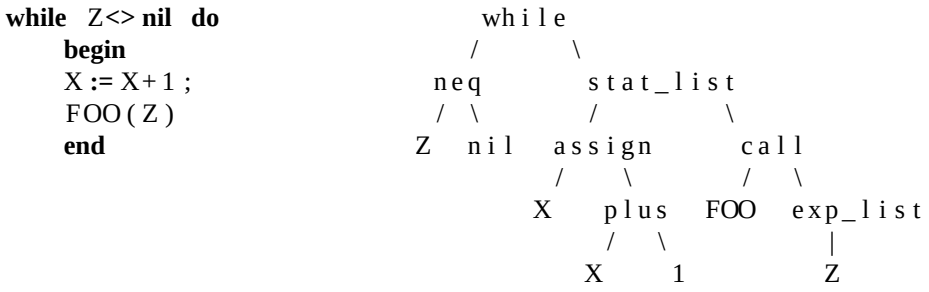

Fig.13.1 Example of abstract representation

Abstract representations are not parse trees. Parse trees often contain structurally irrelevant nodes, and are too dependant on the parsing technique used (because of the constraints it imposes on the defining grammar). The syntactic correctness of an abstract representation of a document in a given language is defined by an **abstract grammar**, which is a tree equivalent of context-free grammars or B.N.F. commonly used to define the concrete syntax of a language. The abstract grammar only reflects the intrinsic structure of the language and is independant of any parsing technique.

An abstract grammar for the language L consists of:

- the definition of the collection of labels (which we call **operators**) allowed in the abstract representation of documents written in L.
- the definition of sets of operators, called **phyla**, corresponding to groups of operators that are allowed in the same context.
- the definition for each operator of a rule specifying its arity (i.e. number of sons) and the phylum associated with each son.

In figure 13.2 we give an example extracted from the abstract grammar of Pascal. The definition of the phylum *STAT* gives the set of operators that may appear on top of a tree to be considered as a statement. The definition of the operator *while* specifies that the sons of a while must be respectively an expression and a statement in that order. List operators may have any number of sons, all belonging to the same phylum.

Fixed arity operators:

```
intcst      -->   {integer}                    (* atomic operator*)
ident       -->   {string}                     (* atomic operator*)
nil         -->                                (* atomic operator*)
neq         -->   EXP          EXP
plus        -->   EXP          EXP
assign      -->   VARIABLE     EXP
call        -->   IDENT        EXP_LIST
while       -->   EXP          STAT
```

List operators:

```
exp_list    -->   EXP          ...
stat_list   -->   STAT         ...
```

Phyla:

```
VARIABLE  ::  ident      meta      unref     index
              dot
EXP       ::  ident      meta      intcst    alfacst
              charcst    nil       hexcst    realcst
              setof      not       uplus     uminus
              unref      hexa      eql       lss
              gtr        neq       leq       geq
              in         intdiv    mod       div
              mult       plus      minus     or
              and        index     dot       format
              call
STAT      ::  meta       goto      repeat    assign
              call       case      while     with
              labstat    stat_list for       if
EXP_LIST  ::  exp_list   meta
```

Fig.13.2 Excerpts from the abstract syntax of Pascal

0.2.3. Combining languages and combining documents.

A language called **Metal** [19] may be used to specify new languages to Mentor. The specification of a language L is a Metal program that has to define at least the abstract grammar of L. A Metal specification may contains other components, the most usual being the specification of the concrete textual syntax of the language, and of the translations between concrete and abstract representation. Work is under way to provide in Metal the means to specify more semantical properties of a language, such as scope or type checking [4].

The Metal specification of a language may be compiled into a collection of tables (or virtual machine code) that are used by Mentor to correctly handle this language. Actually Mentor can handle several languages simultaneously, which is a necessary step towards the maintenance of a project data base.

The organization of a data base of project related documents has not yet been much developed in Mentor. However two mechanisms are available that are intended to facilitate the creation of such a data base. They are called respectively **annotations** and **gates**, and they may be used to combine documents or fragments of documents belonging to different languages, or even to create multi-lingual documents.

Annotations are essentially a means of attaching additional information to a node of the abstract representation. This information may just be comments in the usual sense. It may also be any kind of data that may play a role in some circumstances, but is not an essential component of the document: footnotes, references, assertions, examples, etc... In the new Lisp implementation of the system, annotations may also serve as attributes (in the sense of attributed grammars [28]) used to perform computations on the document.

Gates concern the association of values to the atomic nodes (i.e. nodes without sons) of abstract representation trees. Most atomic nodes, in addition to their label, e.g. *ident* or *intcst*, must have an associated value of an appropriate type, e.g. a string or an integer (see fig.13.2). This value may also be an abstract representation tree, belonging to the same or to some other language.

A detailed discussion of gates and annotations is given in [8]. So far they have only been used to produce single documents integrating components in different languages. Annotations or atom values could also be references to fragments of (or places in) other documents, and could thus serve to establish finely grained relations between documents within a project data base.

0.2.4. Manipulation of documents.

0.2.4.1. The manipulation primitives.

The main purposes of the abstract representation of document is to emphasize their internal structure and thus support their manipulation in terms of this structure. Naturally Mentor provides a collection of primitive actions to explore or modify the abstract representation trees. These actions take as argument **cursors** denoting positions in the trees. New cursors may be created as needed, thus permitting to refer simultaneously to several places in one or several documents.

The main classes of manipulation primitives are:

- *local tree navigation* : up, down, right, left to explore in arbitrary order the branches of the tree; next to explore in prefix order; primitives to move onto annotations or gates values.
- *simple tree editing* : modification, deletion, insertion, exchange of subtrees; modification, deletion or addition of new annotations.
- *pattern matching based actions* : creation of incomplete trees to be used as patterns, pattern matching and instantiation, pattern directed tree searches, pattern directed tree traversal and modification. Tree pattern matching is a powerful technique in all symbolic manipulations, from simple interactive structured editing to much larger applications such as pretty-printer generators or source to source program tranformers.
- *parsing and unparsing primitives* : they perform translations between the concrete textual representation and the abstract tree representation; they are driven by tables generated by the Metal compiler from the specification of the language used.
- *filing primitives* : they store and retrieve abstract representation of documents in the host file system.
- *miscellaneous primitives* : for example to get the label of a node, or the number of sons of a list node.

With respect to filing, let us recall that the abstract representation has been defined here abstractedly, i.e. without any thought about its actual implementation. Since the abstract representation is to be the standard one, its is essential that documents be kept in this abstract form at all times, whether in main memory or on file. Furthermore the abstract representation often resolves ambiguities present in the concrete one (such as the association of a comment with the commented text fragment). Thus two equivalent implementations of the abstract representation are actually provided: one in main memory implemented with pointers for fast manipulations, and one linearized (in polish form) in ascii files. Filing primitives perform the translation between the two forms. For a Pascal program, the linearized abstract representation file is about 40 percent shorter than the equivalent text file.

0.2.4.2. Mentor as a syntax directed editor.

Mentor may be used as a syntax directed editor either on a teletype or in full screen mode on a CRT. Teletype mode is convenient when communicating with the system on a low bandwidth line. A bit-map and mouse version is under development.

In CRT full screen mode, the screen is divided into four windows. The largest one contains the visible (pretty-printed) fragment of document currently edited. This fragment corresponds to a tree denoted by a special system cursor: the visualization cursor. The subtree on which commands are actually activated is displayed in boldface (or underlined characters) within the visible context. It is denoted by an activation cursor. A small window indicates the language and the operator label of the activation subtree. The other two windows are for system messages and diagnostics, and for user input of commands. A fifth window containing help messages may appear when requested by the user. To accomodate window size, it is possible to display abbreviated version of documents, or to scroll the documents in the display window.

New documents may be entered in Mentor either via a multi-entry parser (from a file or from the keyboard) or incrementally with the assistance of a menu system [24]. They may of course be loaded directly in tree form from abstract representation files.

Editing commands are entered with an interactive tree editing language called **Mentol** that can activate the manipulation primitives described above.

0.2.4.3. Creation of new tools.

The Mentol language, though only interpreted, is a full programming language specialized for tree manipulation applications. Its programmability can be used to extend the collection of editing functions, or to develop more complex applications as presented in the next section.

The manipulation primitives are actually Pascal procedures. Thus document manipulation or analysis tools may also be written in Pascal using these primitives, and then linked with the system. Data and control may be exchanged between compiled Pascal code and interpreted Mentol code, allowing one to take advantage of the speed of the former and of the flexibility and interactivity of the latter. As a result, applications written in Pascal may be called interactively during editing sessions.

0.2.5. Practical applications of Mentor.

The primary use or the Mentor system is as an interactive stuctured editor. A variety of languages have already been specified in Metal for this only purpose (e.g. Pascal, Ada, Chill, Metal, Typol, Estelle [16]). The collection of manipulation primitives being limited to purely syntactic manipulation, it is convenient to develop for each language a collection of small or large Mentol programs incorporating some knowledge of the semantics of the language, to assist the user more effectively. Such specialized interactive environments have been developed for Pascal [21], Ada [9], LTR-2 and LTR-3 [34], CPL1.

A special mention is deserved by the language Rapport [22], which has been designed together with its environnement to assist the production of technical reports.

The tools included in these environments may cover many topics:

- *simple navigation and modifications* : for example to find the declaration of an object with a given name, or to automatically create this declaration without actually moving the editing cursor to the declaration part.
- *debugging* : they are based on mechanical insertion (and removal) of debugging statements in the source code.
- *program analysis* : e.g. to produce cross-reference tables, to discover unused segments of programs, or to detect aliasing and potential side-effects [26].
- *program documentation* : automatic adjunction of comments on the basis of analysis results.
- *checking* : tools that ensure the respect of standard semantics (e.g. scope and type discipline), or of local programming norms such as restricted use of the goto statement.

- *static and dynamic measures* : static metrics are easily computed with a structured form or programs [30]. Dynamic measures are obtained by mechanical insertion of measuring statements in the source code.
- *automatic program generation* : by means of a library of program skeletons to be filled in with computer assistance.

A larger application of Mentor is the mechanisation of the tranport of Pascal programs between systems and dialects [7,12]. This is a non trivial task, mainly because of the widely diverging conventions concerning separate compilation.

Mechanical generation of these tools for particular languages from a specification expressed in an extension of Metal is an important topic for current research [10].

Mentor is also often used as a tool to assist language design. A extension of Pascal with a good module and interface facility has been designed under Mentor, and a compiler from the extended language to standard Pascal source code was implemented in Mentol [23].

0.3. IMPLEMENTATIONS ISSUES

The choice of Pascal as the implementation language was in many respects very beneficial. Despite some odd and irksome limitations (on the type of function results for example) the language is reasonably well structured and quite readable. It has become at the same time very popular and standardized enough to allow the port of Mentor on a variety of machines and systems without impairing efficiency. As a well structured language, Pascal was a good experimental subject for the techniques developed in Mentor, and the use of the system for its own development has been an unvaluable source of improvements and new ideas, as well as a benchmark for validating those ideas.

However the development of Mentor in Pascal became increasingly difficult as the system evolved. This was due in part to the natural aging of code in an evolving system, but several problems are traceable to the inadequacy of Pascal for the programming structures required by the technology being developed.

0.3.1. Memory management.

Abstract syntax trees are implemented in main memory by means of records linked by pointers. This implementation is storage costly, but it is very dynamic and allows the fast response required by interactive systems. The records have to be allocated dynamically as trees are created but , as is commonly the case is symbolic manipulation systems, there is no way of knowing when a tree ceases to be usable (i.e. accessible) from within the system. Without some form of garbage collection, accessible free storage is slowly used up, and sessions have to be abandonned after a while for lack of available memory. Practically none of the existing Pascal compilers offers a garbage collector (which it is possible although difficult to implement).

A second problem is related to the possibility of dynamically allocating arrays with a size computed at run time. The code of Mentor is independant of the languages edited and all component processors are driven by language specific tables which are loaded when needed. Efficiency of access usually requires the use of arravs to store these tables. The size of each table varies widely according to the complexity of the concerned language. Since the typing structure of Pascal prevents allocation of arrays whose size is computed at run time, the choice is either to choose a maximum allowed size which is usually wasteful and limits the complexity of processable languages, or to handle storage management explicitely at the (considerable) expense of readability and maintainability.

0.3.2. Type discipline.

The very strict and limited type discipline of Pascal has been a hindrance or a source of inefficiency in other ways. One example is that the system needs identical hash-coded tables of objects of different types, but same physical size (pointers): Pascal requires duplication of the source code to avoid typing conflicts.

More importantly, some structures of the system should be type independant if the system is to be open and extensible [14]. This is for example the case with the annotation mechanism. In Mentor, annotations are always trees belonging to some language. Since annotations are very similar in organization to attributes [28], several Mentor users have requested the possibility of extending them at will to other types and actually use them as attributes. Unfortunately, as with hash-coded tables, their is no simple way to have a unique set of primitives to handle annotations of any type. To some extent the use of variant records, with the discriminant field playing the role of a type tag, may help simulate dynamic typing (i.e. run-time use of values explicitely tagged with their type). Nevertheless, the introduction of any new type requires modification in the source code of the declaration of the variant record type, and the addition of a new case in all subprogram accessing values of that variant record type.

Some of these typing problems could be somewhat alleviated with a more powerful type discipline as in ML [25], but true extensibility of the system by simple addition of new modules without modifications of existing code, requires some form of dynamic typing à la Smalltalk. Of course, a weak alternative is no type discipline at all.

0.3.3. Functional parameters and variables.

Functional or procedural parameters are essential in symbolic applications. In Mentor, for example, one frequently uses tree-walking functions that (roughly) take as argument a tree and a procedure, and call this procedure at every node of the tree. The use of pattern-matching also often requires passing to the matching function an environment-modifying function that can modify the values of variables in the appropriate environment according to the structure of the pattern. A third example is passing the error processing routine as parameter to the system processors in order to have a uniform, though may be user dependant, treatment of error messages.

Although theoretically available in Pascal, functional arguments are usually unreliable or even unavailable in many Pascal compilers, and they were not used in the implementation of Mentor. The tree-walking functions used in Mentor were all written in Mentol.

With respect to the previous section, let us also note that one can simulate dynamic typing in a language having storable typeless pointers and functional values. This is achieved by tagging values with a pointer to a type descriptor containing in its fields the functions (i.e. methods in Smalltalk) associated to the type.

0.3.4. Modularity and separate compilation.

Organization, maintenance or reconfiguration of a large system written in Pascal is rather difficult for lack of a module facility. This is somewhat compensated on most Pascal systems by some form of separate compilation, which may be used to physically separate the logical components of the system. However separate compilation is non standard in Pascal, and it has been diversely implemented. Its necessary use has been a major source of difficulties when transporting Mentor onto new Pascal compiler.

0.3.5. Naming.

Another major maintenance problem was the lack of a good naming facility. Many Pascal implementations do not allow more than 8 to 10 significant characters for identifiers, making it rather difficult to have meaningful but different names for related entities (e.g. the print functions of all types, or the functions giving the language of a tree, an operator, a phylum, etc.). Futhermore, one often wishes to use the same name for entities playing related roles in the program. Standard solutions exist, such as prefixing the name of an object with the name of the module where it is defined (e.g. Ada [33]), or overloading a name with several meanings and then determining from the context the intended meaning of each use of this name. This context may be static as in Ada, or dynamic as in Smalltalk.

0.3.6. Exceptions.

The lack of an exception facility in Pascal has been felt on two accounts:

- exceptions are essential for the robustness (i.e. resistance to failures of all kinds) of a large system,
- exceptions are convenient for the backtrack programming style which is often used in symbolic computation.

Exceptions are available in Ada and Pl/1 in a form oriented towards fail-safe programming, and in Lisp and ML more oriented towards backtrack programming.

0.3.7. Dynamic linking.

Modern software development environments (e.g. Smalltalk, Emacs, Interlisp [11,31,32]) take for granted the possibility of tailoring and extending the environment from inside, that is by adding new code module while the system is running. This dynamic linking facility is usually provided by a language specific environment (it is standard in interpreted languages), but it may also be a characteristic of the operating system (e.g. Multics [27]) and it is not incompatible with compilation (e.g. Le_Lisp [3]).

Dynamic linking is essential to users who want to program specific manipulation routines for immediate use within their session. Its absence in almost all Pascal implementations, together with some of the afore mentionned limitations of Pascal, is one reason for developing Mentol into a programming language. Extensions to Mentor may be programmed either in Mentol or in Pascal, but only Mentol allows dynamic changes.

0.4. THE LISP IMPLEMENTATION

0.4.1. Motivations.

The increasing complexity of the implementation of Mentor, and the limitations of Pascal led us in 1983 to consider starting a new implementation in a different language. Several languages were considered, the main contenders being ML [25], Ada [33], C [20] and some versions of Lisp. The selection was reduced to C and Le_Lisp [3] for several reasons:

- they were readily available and efficiently implemented on many systems,
- they have a very lax type discipline,
- they allow functions and procedures as storable values.

The motivation in the last two points was to have the freedom to choose the most appropriate programming style. The price to be paid is the absence of type checking assistance during program development, particularly in the case of Lisp since limited type checking is available for C [18].

The final deciding factors in favor of Le_Lisp were its memory management with garbage collection, its exception handling primitives, and the existence on top of Le_Lisp of a portable extension for object oriented programming called Ceyx [15].

Augmented with Ceyx, Le_Lisp is a powerful system and our technical choices were validated by the fast realization, the structural simplicity and the conciseness of the new implementation. There were however some drawbacks:

- lack of system assistance with respect to syntax and type checking,
- insecurity and lack of modularity due to the dynamic binding discipline of classical Lisp,
- cost of function calls also due to dynamic binding, [2]
- higher storage costs (than in Pascal or C) due the use of binary lists rather than records (Le_Lisp does have records, but the space overhead required by garbage collection and compaction limits the usefulness of small records).

[2] Strangely enough, classical Lisp with dynamic binding is an inefficient language for functional programming.

- communications between Lisp and other languages are often difficult or limited, especially the call of Lisp functions from other languages.

These remarks are of course specific of Le_Lisp and the situation may be different with other dialects. Le_Lisp itself is being modified to converge towards the Common_Lisp standard which uses a static binding discipline.

Our decision to implement Mentor anew coincided with the start of the pilot project Concerto whose objective is to produce an IPSE prototype for the French PTT. They had arrived independantly to the same choice of implementation language and decided to use this new implementation as the syntactic component of their IPSE.

0.4.2. Organization.

The experience acquired with the Pascal implementation of Mentor has established that its role as a suplier of basic building blocks for the construction of larger tools is as important as its interactive use as an editor. Consequently, it was decided to separate the design of the kernel system from that of the man machine interface. Actually the author has been working exclusively on the design and implementation of this kernel. A man machine interface, including a pretty-printer generator, is being developed independantly within the Concerto project on the basis of the specification of the kernel.

The kernel Lisp implementation is essentially a collection of type, or classes in the Smalltalk sense. The style of programming is essentially object oriented, although types tags are kept at run-time only when necessary so as to reduce space requirements. For each class one must define the representation of values in that class in main memory and in secondary memory, and also define all the functions acting on values of the class. No use is made of the subclassing and inheritance mechanisms available in Ceyx, which was used mainly for its convenient notations and for its record definition facility.

This new implementation extends in several respects the functionalities of the Pascal versions, and especially the pattern matching facilities and the use of gates and annotations. It has a parser based on a Lisp version of Yacc [17] and a Metal compiler. However its abstract representation files are fully compatible with those of the Pascal version when the extensions are not used.

The size of the code is currently on the order of 6500 lines of code (comments included), i.e. less than a third of the corresponding part of the Pascal implementation, although a precise comparison is difficult.

The kernel has been used for a year and a half by about fifteen implementors within the Concerto project for the construction of a complete IPSE. It is a basic component of the Esprit project GIPE (Generation of Interactive Programming Environments).

0.5. CONCLUSION

It is our thesis that a software project is essentially the manipulation of a large number of structured documents. The kernel of any IPSE must contain a powerful symbolic computation facility for efficient processing of these documents in terms of their structure. Such a symbolic computation facility has proved, in our experience, to be a sound common basis on which to build a variety of tools that are to cooperate, and it should provide a uniform methodology covering the whole software life-cycle. Futhermore we expect this approach to benefit from the considerable on going research in symbolic computation.

On the pragmatic side, symbolic manipulation cannot be efficiently implemented in all programming languages. Some basic requirements such as efficient memory management or higher order functions disqualify many traditional languages such as Fortran, Pascal or Ada. This is not to say that symbolic computation is impossible with these languages, it is just a lot more difficult. Futhermore the emergence of efficient portable implementations of suitable languages such as Lisp or ML makes them very good candidates for the realization of industrial IPSE's.

Acknowledgements: The development of Mentor and of the ideas it embodies was conducted by the Croap project at INRIA. The first specification of the Lisp implementation was clarified through several discussions with A. Conchon, V. Donzeau-Gouge, G. Kahn, B. Mélèse and Y. Rouzaud. The members of the Concerto project have provided valuable feedback as first users of this new implementation, which is the joint work of the author and P. Borras.

References.

1. Ambriola, V., Kaiser, G.E., and Ellison, R.J., august 1984, *An Action Routine Model for Aloe,* Report CMU-CS-84-156, Carnegie-Mellon University.
2. André, E., Moreau, B., and Rougeot B., 1984, *Vers un atelier flexible et intégré de logiciel: le projet Concerto,* L'Echo des Recherches, 115, 11-20.
3. Chailloux, J., Devin, M., and Hullot, J.M., 1984, *Le_Lisp, a portable and efficient Lisp system,* Proc. 1984 ACM Symp. on Lisp and Functional Programming, Austin, Texas, 113-122.
4. Despeyroux, T., june 1984, *Executable Specification of Static Semantics,* Lecture Notes in Computer Science, Vol. 185.
5. Donzeau-Gouge, V., Kahn, G., Huet, G., Lang, B., and Levy, J.J., 1975 *A structure oriented program editor:a first step toward computer assisted programming,* Proc. International Computing Symposium, North_Holland.
6. Donzeau-Gouge, V., Kahn, G., Lang, B., Mélèse, B., and Morcos, E., sept. 1983, *Outline of a tool for document manipulation,* Proc. of IFIP 83 conf., Paris, R.E.A. Mason (ed.), North Holland.
7. Donzeau-Gouge, V., Lang, B., and Mélèse, B., march 1984, *Practical Applications of a Syntax Directed Program Manipulation Environment,* Proc. 7th Conference on Software Engineering, Orlando.
8. Donzeau-Gouge, V., Kahn, G. , Lang, B., and Mélèse, B., april 1984 *Document Structure and Modularity in Mentor,* Proc. ACM SIGSOFT/SIGPLAN Soft. Eng. Symp. on Practical Software Development Environments, Pittsburgh.
9. Donzeau-Gouge, V., Lang, B., and Mélèse, B., may 1985, *A tool for Ada Program Manipulations: Mentor-Ada,* Proc. International Ada Conference, Paris.
10. GIPE, 1984. *Generation of Interactive Programming Environments.* Esprit Project Proposal, N.348.
11. Goldberg, A., and Robson, D., 1983, *Smalltalk-80. The Language and its implementation,* Addison-Wesley.
12. Hanout, J.C., Kaiser, C., Lucas, H., and Martin, B., 1983, *Experiences in Programming and Transporting Pascal Systems,* TSI - Technique et Science Informatique, 2, 401-417.
13. Henderson, P.B., Edit., april 1984, Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Pittsburgh.
14. Hewitt, C., april 1985, *The Challenge of Open Systems,* Byte, 10, 223-242.
15. Hullot, J.M., 1983, *Ceyx, a multiformalism programming environment,* Proc. of IFIP 83 conf., Paris, R.E.A. Mason (ed.), North Holland.
16. ISO/TC97/SC21, june 1985, *Estelle - A Formal Description Technique,* ANSI N 42, DP9074.
17. Johnson, S.C., january 1979, *Yacc, Yet Another Compiler-Compiler,* Unix Programmer's Manual, 7th edition, Bell Telephone Laboratories.
18. Johnson, S.C., january 1979, *Lint, a C Program Checker,* Unix Programmer's Manual, 7th edition, Bell Telephone Laboratories.
19. Kahn, G., Lang, B., Mélèse, B., and Morcos, E., 1983, *Metal: a formalism to specify formalisms,* Science of Computer Programming, 3, North Holland, 151-188.

20. Kernighan, B.W., and Ritchie, D.M., 1978, *The C Programming Language*, Prentiss-Hall, Englewood Cliffs, New Jersey.

21. Mélèse, B., 1981, *L'environnement Pascal*, INRIA Technical Report N.5.

22. Mélèse, B., june 1984, *Structured editing - Unstructured editing, Cooperation and complementarity*, Actes du 2e Colloque de Génie Logiciel, Nice.

23. Migot, V., sept. 1983, *Un Pascal modulaire sous Mentor*, Thesis, Université Paris XI, Orsay.

24. Migot, V., *A new User Interface in Mentor: the Menu Mode*, Rapport INRIA (to appear).

25. Milner, R., august 1984, *A proposal for standard ML*, Proc. 1984 ACM Symp. on Lisp and Functional Programming, Austin, Texas, 113-122.

26. Morcos-Oury, E., 1979, *Etude des effets de bord des appels de procédures et de fonctions dans le langage Pascal*, Thesis, Université Paris XI, Orsay.

27. Organick, E.I., 1972, *The Multics system: an examination of its structure*, MIT Press, Cambridge, Mass.

28. Reps, T., january 1982, *Optimal-time Incremental Semantic Analisys for Syntax Directed Editors*, Proc. 9th Annual Symp. on Principles of Programming Languages.

29. Reps, T., august 1982, *Generating Language Based Environments*, Tech. Report 82-514, Cornell University ,Ithaca, NY.

30. Schroeder, A., 1983, *Integrated program measurement and documentation tools*, INRIA Research Report N. 227.

31. Stallman, R.M., june 1979, *EMACS: The extensible, customizable, self-documenting display editor*, MIT, AI Memo 519, Cambridge, Mass..

32. Teitelman, W., october 1978, *Interlisp Reference Manual*, Xerox PARC.

33. U.S. Department of Defense, january 1983, *Reference Manual for the Ada Programming Language*, ANSI/MIL-STD-1815 A.

34. Verove, D., march 1983, *Mentor-LTR: Système de manipulation de programmes LTR-V3*, Technical Report DRET-SEMA.