Programming environments based on structured editors : the Mentor experience

Véronique Donzeau-Gouge, Gérard Huet, Bernard Lang, Gilles Kahn

INRIA

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
BP 105 78150 Le Chesnay
France
Tél 954 9020

Rapports de Recherche

N° 26

PROGRAMMING ENVIRONMENTS BASED ON STRUCTURED EDITORS: THE MENTOR EXPERIENCE

Véronique DONZEAU-GOUGE
Gérard HUET
Gilles KAHN
Bernard LANG

Juillet 1980

PROGRAMMING ENVIRONMENTS BASED ON STRUCTURED EDITORS : THE MENTOR EXPERIENCE

Véronique DONZEAU-GOUGE, Gérard HUET, Gilles KAHN and Bernard LANG

Résumé :

Nous analysons dans cette note l'expérience acquise avec le système de manipulation de programmes MENTOR, en mettant l'accent sur les points suivants :

- les principales décisions prises lors de la conception de MENTOR ;
- notre expérience dans la construction et l'utilisation d'un environnement de programmation PASCAL construit à partir de MENTOR ;
- notre conception de ce que devrait être un environnement de programmation complet.

Abstract :

We discuss in this note our experience with the MENTOR program manipulation system, from the following points of view :

- the main design decisions we made in MENTOR ;
- our experience with building and using a PASCAL programming environment based on MENTOR ;
- our vision of a complete programming environment.

Programming Environments Based on Structured Editors: The MENTOR Experience

Véronique Donzeau-Gouge, Gérard Huet, Gilles Kahn and Bernard Lang

Abstract

We discuss in this note our experience with the MENTOR program manipulation system, from the following points of view:

- The main design decisions we made in MENTOR;
- Our experience with building and using a PASCAL programming environment based on MENTOR;
- Our vision of a complete programming environment.

1. A MENTOR primer

MENTOR is a processor designed to manipulate structured data. This data is represented as operator-operand trees, generally called *abstract syntax trees*. MENTOR is driven by the tree manipulation language MENTOL.

1.1 Abstract Syntax

Abstract syntax trees are structured as sorted algebras; for a given language, we declare a set of *sorts*, and a set of *operators* with sorted operands. Operators may be declared with a fixed arity, or may be associative operators with a variable number of arguments, used to represent lists. We must also specify a parser, which, given a sort, maps a concrete syntax string into the corresponding abstract syntax tree, and some standard inverse mapping, the prettyprinting unparser.

For instance, in MENTOR-PASCAL, typical sorts are *exp*, *stat*, *varbl*, *ident*, *const*, *lexp*, *lstat*. Every meaningful PASCAL construct corresponds to an operator. Typical operators are if, ass, call, lstat, gtr, mult, index, with sorts as follows:

- if: $exp \times stat \times stat \rightarrow stat$.
- ass: $varbl \times exp \rightarrow stat$.
- call: $ident \times lexp \rightarrow stat$.
- lstat: $stat \times stat \times \cdots \times stat \rightarrow lstat$.
- lexp: $exp \times exp \times \cdots \times exp \rightarrow lexp$.
- gtr: $exp \times exp \rightarrow exp$.
- mult: $exp \times exp \rightarrow exp$.
- index: $ident \times iexp \rightarrow exp$.

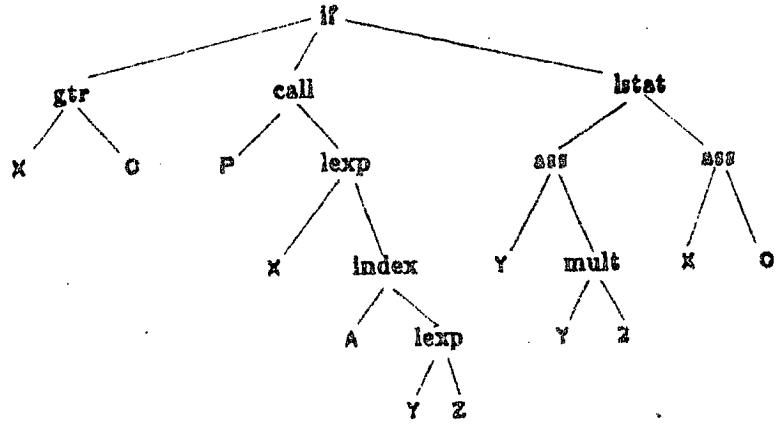
Also, all identifiers and constants are nullary operators, of sort respectively *ident* and *const*. Finally, our sorts are ordered; for instance, $ident \subseteq varbl \subseteq exp$, $const \subseteq exp$ and $lstat \subseteq stat$. In any argument place of sort σ, all operators returning sort $\sigma' \subseteq \sigma$ are authorised.

Example

The following PASCAL program :

```
if X>0 then P(X,A[Y,Z])
else begin
     Y:=Y*2;
     X:=0
     end
```

parses into (and is the unparsing of) the following abstract syntax tree:



1.2 MENTOL, a tree manipulation language

The user communicates with MENTOR through an interpreter for a specialised tree manipulation language, MENTOL. Values in MENTOL are abstract syntax trees (abbreviated ast from now on) and locations in these trees , abbreviated loc. MENTOL commands are themselves ast's in MENTOR-MENTOL. MENTOL variables, called *markers*, may be assigned locs. A loc expression is obtained by composing a base marker with displacement operators such as U,L,R for up,left,right, or Sn, with n an integer, for n-th son. For instance, if marker @TOP marks the top of the above PASCAL tree, @TOP S2 S1 marks the location of identifier P. The *current* marker, @K, may be abbreviated by the empty string for convenience. The MENTOL assignment statement, of the form loc1:loc2, is used to move around in trees, and remember places. For instance, :@TOP S2 S1 would assign the current marker to the location of P in the tree above.

The command loc Pn prints on the console the result of unparsing the ast at loc, down to a level of detail specified by integer n. For instance, @TOP P2 would print:

```
if # then # else ...
```

Note that list nodes are abbreviated by ..., and other nodes by #. The command @TOP P3 would give you some more detail:

```
if X>0 then P(X,A[Y,Z])
else begin
     $;$
     end
```

The standard prettyprinting effected by MENTOR puts PASCAL reserved words in lower case, identifiers in upper case, and indicates the tree structure by indentation. When the level of detail is unspecified, you get a standard abbreviation that in most cases fits in one screen. For instance, @TOP P would produce the text above in full. The reverse operation of P is &, which is an expression denoting the result of parsing a string of characters read on the input device.

An essential feature of MENTOL is pattern-matching. A pattern or *schema*, is any ast containing special terminal nodes called *metavariables*. A schema matches any tree which is an instance of the pattern, replacing metavariables by appropriate subtrees. A given metavariable may appear only once in a given pattern. Metavariables are unparsed as special identifiers, whose name starts with a dollar sign. Schemas may be constructed by commands, or may be input through the parser. When the syntax tables for a given language are loaded, MENTOR constructs a set of predefined schemas, one for each operator. These elementary schemas consist of just the given operator, applied to metavariables; they are accessible through a marker named by the operator. For instance, in MENTOR-PASCAL, we would have:

```
?@GTR P↵          $EXP1>$EXP2
```

The *find* expression loc F pat denotes the first location in the subtree marked by loc which is an instance of the pattern pat (assuming preorder traversal.) If the subtree does not contain any such instance, the special value `fail` is returned. Another find expression, with F replaced by FF, does not limit the search to the subtree marked by loc; that is, the search is continued in preorder beyond loc (this is the search one ordinarily does in the listing of a program, starting from a given point.) When pattern-matching is successful, the markers with same name as the metavariables of the schema are assigned to the corresponding location in the object tree. For instance, with the above example:

```
?@TOP F @GTR P↵
X>0
?@EXP1 P↵
X
```

Let us now explain briefly the main commands and control structures in MENTOL. When a loc expression is used as a command, it abbreviates the operation of assigning to the base marker of loc the result of evaluating loc. For instance, in the example above, a more typical operation would be to use the current marker as follows:

```
?:@TOP;F@GTR;P↵
X>0
```

Note the sequencer semicolon. A sequence of commands is made into a command (abbreviated com) by enclosing it between parentheses (these are not mandatory at top level.) Any command may be iterated n times by postfixing it with integer n. A star means iterate until failure; for instance, U* brings the current marker to the top of the tree it was pointing into. A primitive conditional statement is provided: ?com1,com2 executes com1 if the previous command succeeded, otherwise it executes com2. The command $n exits from n levels of grouping; $-n is the same, but returns failure. There are various other control statements such as a case statement, which we shall not discuss here. The interested reader is referred to MENTOR's manual[2]. Let us now turn to the commands that modify asts.

The change command, loc1 C ast2, replaces the subtree marked by loc1 with the tree ast2. Like in Algol 60, a form of coercion is provided: When the second argument of the change command is some location expression loc2, it denotes a copy of the subtree marked by loc2. Various list manipulation commands, such as insert (I) and delete (D), are provided. loc1 X loc2 exchanges the subtrees at loc1 and loc2 (provided they are disjoint). All these operations maintain the correctness of sorts. Rather than giving an exhaustive list, let us give a few examples:

```
?:@TOP,
?B2 X B3,
?B2 B1 I B3,
?B3 C &,
[STAT]:Z:=0;,          %colon is prompt for parsing;note the sort reminder
?B1 X B2,
ERROR: WRONG SYNTAX TYPE
?P,
if X>0 then
     begin
     Y:=Y*2;
     P(X,A[Y,Z]);
     X:=0
     end else Z:=0
```

Let us finally explain an essential command: eval; E ast returns a copy of the tree ast, in which metavariables are instantiated according to the current environment. The eval command, together with pattern-matching, permits to implement easily program transformations that can be described as tree rewriting systems. For instance, assume we want to transform the operator $>$ in the PASCAL exemple above into the operator $>=$. Assuming the current marker is initially positioned at TOP, the simplest way of doing this in MENTOL is as follows:

```
?F@GTR;CE@GE@;P,
X>=0
?@GE@ P,                          %This works because metavariables match
$EXP1>=$EXP2
```

We have not explained so far how we dealt with comments, and more generally with pragmats and assertions. We have designed a general mechanism that takes comments into account as a particular case of various possible annotations, meaningfully related

to program constructs. The idea is to attach *attributes* to any node of an ast. These attributes are themselves asts in their own language. The loc expressions are extended so as to access the various attributes of a given node and to get back from an ast to the ast node it annotates, if any. For instance, in MENTOR-PASCAL, two attributes are reserved for ordinary comments: the so-called *prefix* and *postfix* comments. These simple comments have a rather poor structure: they may be just lists of lines. We also use comments in PASCAL abstract syntax; for instance, when we optimise some portion of program, we keep the initial version of the construct as a comment. The system is extensible; for instance, we may declare a new abstract syntax for assertions, and annotate various constructs with them, write in MENTOL a verification condition generator that will compute from these assertions, etc.

If MENTOL consisted only in the features mentioned so far, the reader would question our calling it a programming language, and would probably argue that is is nothing more than an editor command language. What makes MENTOL a full-fledged (although not general-purpose) programming language is the possibility to write MENTOL procedures. We permit three kinds of procedure parameters:

a) locs passed by value;
b) locs passed by reference;
c) coms passed by name.

For instance a standard predefined procedure is FORALL, which takes two arguments: a pattern, and a command. For every instance of the pattern, starting from the current marker and with a preorder tree traversal, it executes its second argument. Various utilities procedures are predefined, to generate new identifiers,and provide coercions mechanisms such as between identifiers, strings and comment lines. Finally standard system procedures are provided for file manipulation, interactive help and debugging, etc.

This procedure encapsulation mechanism is essential to MENTOR.. It allows the designer of a programming environment to provide the user with powerful program manipulations in terms of the logical constructs of the specific programming language manipulated. These manipulations can be heavily context dependent, and may use semantic knowledge of the programming constructs, as opposed to the purely structural context-free manipulations of the MENTOL primitives. Finally, it allows to build extensible systems, in which the user constructs and maintains his own environment of procedures.

1.3 A PASCAL Programming Environment Based on MENTOR

MENTOR is a general system to manipulate structured information. However from the start we intended as its main application the realisation of an interactive programming environment in which a programmer may design, implement, document, debug, test, validate, maintain and transport his programs. Furthermore we intended this environment to be realistic enough to help in implementing large software developments, and provide a programming team with tools for specifying a design, enforcing a programming methodology and verifying interfaces. Our intention when we started the project,

at the end of 1974, was to try and bridge the gap between on the one hand existing programming tools such as debugging compilers, and on the other hand the vast amount of theoretical research on semantics of programming languages. At the same time, we did not want to commit ourselves to any currently proposed programming methodology (top-down design, structured programming, etc.) or formalism (first-order assertions, Hoare rules, modal logic), for which a wide consensus did not exist. Rather, we wanted our system to be general enough to accomodate these various formalisms and provide tools to implement the proposed methodologies. We chose to implement a PASCAL environment around the MENTOR system for several reasons. Most importantly, we had chosen PASCAL as our system implementation language, and we wanted to implement first the tools we needed ourselves in our development effort. We bootstrapped as soon as the core of the system was implemented, and this may be one of the most important practical decisions that forced us to focus on pragmatic issues.

The first step in this effort was the design of a structured editor for PASCAL programs, implemented in MENTOR-PASCAL. That is, we wrote a number of MENTOL procedures that are the main user commands to construct and modify PASCAL programs and their documentation. Some of these procedures are used to move in the ast of the program according to higher-level concepts. For instance, FPROC is used to move to the top of a procedure; it first asks the user the name of this procedure. VAR goes to the immediately surrounding variable declaration part, etc. Other MENTOL procedures effect simple program transformations. For instance, LABEL is used to label a statement. It requests the label name from the user, verifies that this label is neither declared nor used in the current environment, declares it, and finally labels the statement pointed to by the current marker. All these manipulations are transparent to the user, as long as no error condition occurs.

We then turned our effort to implementing tools for the normalization and documentation of PASCAL programs. Normalizing programs consists of arranging them in a standard, more readeable form, while preserving semantic equivalence. For instance, declarations may be rearranged so that logically related items be declared in the same area. Various cleaning-up operations are performed, to get rid of unnecessary structures (empty statements, compound statements, etc.) This is especially important when a series of program transformations have been applied mechanically, since often they are easier to program with redundant structure. Of course none of these simplifications should get rid of comments. Automatic documentation consists of generating comments automatically at various standard places in the program, generating scope structures, cross-reference tables, etc. Some of these may involve complicated computations on the program. The basic philosophy is that all generated documentation is itself structured, so that it can be used by further processes. We do not elaborate further on normalization and documentation of programs in MENTOR, and refer the interested reader to [6].

Another area we started to investigate was an approach to debugging by source-level program manipulation. The idea is that, instead of giving you run-time debugging tools that have a more or less satisfactory user interface, we shall provide you with special versions of your source program, with user interfaces built-in. You can compile and run these special versions using your standard production compiler. For instance, a

procedure PROFILE allows you to compute an execution profile of your program as a side-effect of your main computation. We think this area is worthy of more research.

The effort of designing and implementing a bona fide programming environment based on MENTOR-PASCAL is still going on. Rather than listing in painstaking detail all that is available to the user in the current state of the system, let us discuss what is our idea of a satisfactory environment, and what problems we are encountering in its implementation.

The main philosophy of our programming environment is to build specialized interpreters, that help the programmer by doing various computations and rearrangements on his programs. All these interpreters communicate, between themselves as well as with the user, through the abstract syntax of PASCAL and its annotations. The development of a program is conceived as a multi-pass activity, each processor using as assumptions the normalization and computations effected by the previous passes. For instance, the "correction" of a piece of program may be progressively checked/debugged according to the following scenario:

- As soon as the program is input, it is correct as far as its context-free structure is concerned, and this will be inforced by MENTOR's typing mechanism during any further transformation.
- Then a "scoper" processes the program, checking for existence of declarations for the various identifiers used in the program. This pass may be described as "computing the lambda-calculus skeleton" of the program.
- When all names are linked to their proper declaration, it is easy to write a type-checker, that will check for the correct typing of all the programming language operations. This step is conceptually, and indeed in our scenario, implemented as, a non-standard interpretation of the programming language constructs. A complete set of MENTOL procedures for PASCAL scope and type checking has been developed, and used to develop type-preserving manipulations in MENTOR-PASCAL[7].
- At this stage it is natural to check for run-time errors, termination, aliases. Here we need much more semantic information. Most of the checks mentioned are undecidable in general, but easy sufficient conditions are reasonable to implement. These checks can be realized by the combination of specialized data-flow analysis routines and a general symbolic interpreter. A set of MENTOL procedures that check aliasing in PASCAL and its application to proving sufficient conditions for a procedure to be free of side-effects is described in [9].
- The hardest part of program verification remains: checking that the program actually corresponds to what the programmer expected. The traditional approach would be to implement a debugging interpreter, which would execute directly from the abstract syntax and various other structures (symbol tables) constructed by the above processes. A more formal approach would request from the user to state formal specifications, such as first-order assertions, and to check the adequacy of the program with respect to its specifications. For instance, verification conditions may be generated through symbolic execution, and then input to a theorem prover. The formulas generated, as well as the proof trees, would of course be in turn ast's manipulable by MENTOR; the user could therefore monitor the proof with the same tools he is using for manipulating his

programs. This semi-automated approach would alleviate the difficulty of having to implement a completely automatic theorem prover, a task which is still beyond the state of the art. Another rigorous approach would be to process in MENTOR a complete description of the semantics of the programming language, using for instance semantic equations, and use it to translate a program into its denotation. This allows us to get away from the idiosyncrasies of the particular programming language used and limit the proofs to identities between mathematically well-understood concepts. Such a grandiose meta-system could be conceived as essentially combining the capabilities of the SIS[8] and LCF[5] systems within MENTOR.

A similar scenario can easily be designed for program optimization: local optimisations are performed by program transformations, then more global optimizations are effected at the source level after doing the necessary computations by MENTOL procedures. The program is then compiled in an object code which has its own abstract syntax. Final optimizations are performed by transformations on the object code.

The general strategy behind a programming environment as sketched above is to effect successive refinements on the original program, by going from the simpler, better understood tasks, to the more sophisticated and costly verifications. However, only a small fraction of the above ambitious plan has been actually implemented in MENTOR-PASCAL. There are mostly two reasons for this, which are actually complementary aspects of the same phenomenon:

1) Even the easiest and most natural program transformations are hard to implement in a totally safe way in the current state of baroqueness of programming languages. For instance, it is impossible in PASCAL to separate scope-checking from type-checking because of the with construct. The lack of orthogonality of the language makes it a complex and costly process to do but the most trivial program transformations. For instance, replacing tail recursions by gotos in recursive procedures with call-by-value arguments represents about 200 lines of MENTOL procedures. Again, the assumption must be made (and checked) that no with statement occurs.

2) The more mundane transformations have proved to be challenging and interesting research problems. Their careful implementation is often crucial, since many computations involved turn out to be very time consuming. An especially interesting area of applications is the transport of programs. Our largest application so far was to transport MENTOR from its original IRIS 80 implementation to its PDP 10 version. This is performed in a completely mechanical manner by a set of MENTOR procedures. This way any new release of MENTOR can be followed (after a few hours of computation) by a release of a totally compatible PDP 10 release.

The conclusion we draw from this state of affairs is that no really satisfactory programming environment will exist for ugly languages. On the other hand, it is clear that purely applicative languages are not about to be widely accepted; in the real world of programming, complex data structures with sharing and complex control structures and parameter passing mechanisms are the rule rather than the exception. We are not arguing in favor of simplistic toy-programming languages, but the point is rather that the study of program transformations provides interesting guidelines for the design of future programming languages. As might be expected, these design criteria are closely

related to those based on semantic considerations[10]. We have good hope that the state of affairs will improve with the advent of new programming languages whose design will have benefited from programming languages semantics research and experience gained with systems such as MENTOR. A positive step in this direction has been taken with the ADA language development, since the design of the language included a formal semantics definition. It is interesting to note that this formal semantics is based on a MENTOR-compatible abstract syntax definition.

2. The Main Design Decisions in MENTOR

2.1 The abstract syntax

The notion of abstract syntax is familiar to any compiler writer. It is a tree-like representation of the structure of programs. Operators of the abstract syntax are the basic building blocks of the language. We want to strongly emphasize that abstract syntax is NOT parse trees. It is indeed very different conceptually, although our trees can be obtained by collapsing and normalizing parse trees. Here are a few important differences:

1) Lists are represented as one list node, not as binary trees

2) The reserved words of the language occur as node labels, not as leaves

3) Non-terminals of the grammar do not generate nodes. Certain correspond to sorts, others do not appear at all. For instance, an identifier may occur directly as an expression, the intermediate levels of parsing such as simpleexp, factor, term being collapsed.

4) Parentheses are NOT part of the structure, they are generated optionally by the unparser if the context requires it, because of precedence reasons for instance.

Point 3 is particularly important: every node of the abstract syntax leaves a concretely visible mark in the print-out, and this is a big help for the user going up and down the tree. This makes MENTOR significantly different from previous structured editors such as Hansen's, where the user moved around in his program with the help of grammar menus.

Point 4 is important too. For the MENTOR user an exp is an exp. Precedence relations are left for the unparser to worry about. For instance, with the example above:

```
?@TOP F@MULT;S1C&
[EXP]:Y+1;
?U;P
Y:=(Y+1)*2
```

Similarly, the problem of PASCAL's dangling else completely vanishes:

```
?@TOP B2 C&↵                          %we change the then part of an if
[STAT]:IF Y<X THEN X:=Y;↵             %into a conditional statement
?@TOP P↵
if X>0 then
     if Y<X then X:=Y else            %note the extra else generated
else Z:=0
```

MENTOR trees are not LISP trees either. Even for the LISP language, the coding of programs as binary trees with atom leaves is rather remote from the abstract structure of the program. Also points 1 and 2 above apply. Our structure is much richer structurally; for instance, in MENTOR-PASCAL, we have about 100 operators, whereas LISP structures have only one (cons). For these reasons, we consider MENTOR significantly different from say the INTERLISP editor.

So much for the choice of the general formalism of abstract syntax. Of course for each particular language there is a certain degree of freedom in the design of the particular operators and sorts. As we mentioned above, it is crucial that almost every operator add some concrete representation to the unparsing of a piece of program. An important, but not mandatory, requirement is that the unparsing of an operator should not depend too much on the context in which it occurs. This requirement is met by most operators is MENTOR-PASCAL, except that certain nodes are sometimes surrounded by parentheses according to the context. There are mostly two occurrences of this phenomenon:

a) parentheses surrounding list nodes may change with the context. For instance, an lstat is usually unparsed as a compound `begin ··· end`, except when appearing as the loop of a repeat.

b) parentheses may be needed for precedence, or dangling structures such as shown above for the else. Our unparser always generates the minimum number of parentheses needed for a correct parsing. This is the only normalization (besides indentation of course) that is completely automatic and over which the user has no control.

When designing an abstract syntax for a specific language, the following trade-off occurs. Various constructs of the language may be represented by the same concrete strings. Now there is a choice as to whether you want to separate these two constucts as two distinct operators, or if you want to merge them into one. The maximum discrimination has the advantage that your structure will have a finer grain; for instance, you will catch by the find command instances of one construct independently from instances of the other. On the other hand, certain program transformations will be harder, and the user has more constructs to learn. For instance, should parameter declarations use the same construct as variable declarations? As might be expected, referential transparency and orthogonality are important properties for a programming language to possess for a completely satisfactory design of its abstract syntax.

We feel that allowing arbitrary annotations of nodes by abstract syntax trees in specialized languages was an important design decision. This makes our system open ended to various developments, without interfering with the tools already designed: a given interpreter may have access to certain annotations, the others being invisible. For instance, certain annotations are comments for the user to see. Others may be pragmats

for the compiler, specifications in some formal language for use by a verifier, data structures for control flow analysis, original code commenting some optimised sections, example runs, assertions for run-time checks, etc. It is important that there various structures do not interfere with one another and with the program itself.

It may be appropriate to discuss here why we decided to stick to trees, and not go to more complicated graph structures, such as (shared) dags or control flow graphs. The main reason is that we know how to keep the integrity of these context-free structures in an incremental way. For instance, we could imagine keeping the programs correct according to the full PASCAL syntax, including type checking for instance. But, aside from the fact that it would be a lot more costly to maintain all the information needed for checking this correctness during the edition of the program, this would have the additional (and to our opinion insuperable) drawback that it would preclude the development of programs but in the most awkward fashion.

2.2 MENTOL

MENTOL is our tree-manipulation language. The above description of its main commands gives a flavor of MENTOL programming. The salient features of the language are:

a) it is an interactive language, used for editing; but it may also be used to program lenghty batch computations.

b) it is not applicative; MENTOL constructs divide into expressions, that are simply evaluated for their result, and commands, which have various side effects.

c) it is a specialized language, for manipulating trees; it has no pretense of being general-purpose, although it has rudimentary arithmetic capabilities.

d) it has reasonably good user interaction facilities: there are various debugging aids such as a trace package, an interrupt facility, and the user may execute in coroutine with programs, a very handy feature for "controlled" program manipulation.

e) MENTOL has its own abstract syntax. It is therefore possible to edit and develop MENTOL programs under MENTOR. Actually a standard facility exists to go back and forth between a PASCAL editing session and a MENTOL editing environment, in which the (advanced) user may modify his PASCAL manipulation programs.

f) File manipulation primitives are provided. Several formats of files are known to MENTOR: standard text files, that may be input (through parsing) and output (through unparsing). Tree files, that keep asts from one session to the next without the need to reparse. MENTOL files, containing MENTOL procedures, and a special case of which is used as the initialisation file, loading a specific user's editing prelude.

Pattern matching deserves a little discussion. As we already insisted, pattern matching is a fundamental operation in MENTOL. The user may construct any tree pattern, i.e. any ast with metavariables occurring anywhere. However, metavariables may occur in only one occurrence. This condition is required because of the side-effects to the corresponding markers. Note that this is not really a restriction, since a primitive is provided for testing equality of trees. No list metavariables are provided at the moment, because associative pattern matching is a complicated operation (a tree may match a

pattern in more than one way if such list variables are allowed), and because it was never strongly felt as desirable, except probably for orthogonality. But we want to stress the considerable pattern matching capability we have in MENTOR, as opposed say to string searching in a more conventional editor. Anybody who tries to trace uses of identifier I in his program (as opposed to all occurrences of character I, in other identifiers, reserved words, strings and comments!) will understand this point. Furthermore the MENTOL pattern matching is fast, because types are used to focus the search on the relevant part of the trees. For instance, MENTOR knows that in PASCAL statements are disjoint from declarations, and may not occur in expressions. It will therefore focus the search for a statement on a narrow region in the program tree (and of course comments will not get in the way either). We may therefore argue that tree pattern matching is actually faster than string pattern matching. We believe this is one of the main arguments for having typed rather than untyped structures.

It is clear that MENTOL is not the last word in tree-manipulation languages. However, we wanted to acquire a reasonable amount of experience with writing program transformations in MENTOL before drawing definite conclusions about such languages. All in all, MENTOL has well served its purpose: it is easy to learn, it is fairly easy to debug, it is fast enough for editing. However long MENTOL procedures are hard to read, and a compiler is clearly needed for complicated transformations done in batch mode.

2.3 A special word for screen editor fans.

One of the most commonly heard criticisms of MENTOR is that it should be possible to edit programs on your screen in the same way as for instance with the EMACS editor. We do not believe that this would be an easy task, and we do not even think that such a facility is really desirable.

The first point concerns portability. In the initial MENTOR design, we had planned to define a screen as partitioned between several areas, and to have the text under the current marker represented specially on the screen. We went as far as implementing these displays, but then changed our minds, mostly because it was very hard to distribute our system. We reverted to teletype-compatible output. MENTOR can be transported to any machine with any interactive operating system (modulo the PASCAL transport problems, of course). No special terminals are needed, but of course the system will be more pleasant to use if the rate of transmission is higher, so that it is not too costly to have the current marker expression printed often.

The second point concerns the difficulty to maintain two separate representations. Remember, the text printed on your screen is nowhere kept; it is just computed on demand by the prettyprinter. The ability to manipulate screen images would force us to keep the printed text internally, and to try and link it to the corresponding ast. After some modification is effected on the screen, the parser would have to be called in action to validate the changes before updating the tree accordingly. The difficulties may not be insurmountable, but it is not clear that the end result would be worth the effort.

Finally, a major drawback of mixing structured editing and display editing is that the user would have to learn how to use two command languages instead of one. We believe that most users would stick to either mode, but would not like mixing them.

Conclusion

MENTOR has been used for most of its own development and maintenance. Various groups at INRIA use MENTOR as their main programming tool for developing PASCAL programs. MENTOR has been distributed in various research and teaching institutions. In particular, it is being used at Université de Toulouse for teaching programming in PASCAL. Using MENTOR requires some training. It seems that in the average a PASCAL programmer needs about a week to get accustomed to this new world of trees. Past this period, few return to the standard tools.

It is our thesis that using an abstract manipulation system as the core of a programming environment is a good paradigm. However, it is very important that the user may correspond to the system through the concrete syntax he is used to: he should be able to visualize his trees with unparsing, and conversely input his program text with parsing. The abstract syntax manipulation language should have a powerful procedure abstraction mechanism, permitting to extend the system at will with complicated semantic checking, such as data flow analysis and ultimately formal proofs. It is important to be able to manipulate structured annotations, linked to the structure of the program, but conceived as separate entities and not as extensions to the user's programming language syntax. We envision a satisfactory programming environment as unifying under a common set of tools the whole range of a programming team's activity: design, development, documentation, debugging, maintenance and transport. However the long range goal of software reliability will be attainable only when new programming languages, designed along sound semantic principles, will be available.

Acknowledgments

MENTOR was designed and implemented at IRIA by the authors of this paper. Various other people have been involved occasionally in the MENTOR project: V. Chan, J.J. Lévy, B. Mélèse, E. Morcos-Oury, Y. Sugito.

References

1. Donzeau-Gouge V., Huet G., Kahn G., Lang B. and Lévy J.J. *A Structure Oriented Program Editor: a First Step Towards Computer Assisted Programming.* International Computing Symposium 1975, Antibes, France. Also Rapport Laboria 114, Avril 1975, IRIA.

2. Donzeau-Gouge V., Huet G., Kahn G. and Lang B. *The MENTOR User's Manual.* Available from INRIA, Rocquencourt, France.

3. Donzeau-Gouge V., Huet G., Kahn G. and Lang B. *Introduction au systeme MENTOR et a ses applications*. Actes des Journees Francophones sur la Certification du Logiciel, Genève, Janvier 1979.

4. Donzeau-Gouge V., Huet G., Kahn G. and Lang B. *The MENTOR Program Manipulation System*. In preparation.

5. Gordon M., Milner R., and Wadsworth C. *Edinburgh LCF*. Report CSR-11-77, Computer Science Department, Edinburgh University, 1977.

6. Kahn G. *Normalisation et documentation des programmes*. Note technique, Mai 1978, IRIA.

7. Mélèse B. *Manipulation des programmes Pascal au niveau des concepts du langage*. Thèse de 3ème cycle, Université d'Orsay, Juin 1980.

8. Mosses P. *SIS - Semantics Implementation System. Reference Manual and User Guide*. Report DAIMI MD-30, Computer Science Dept., Aarhus University, Aug. 1979.

9. Morcos-Oury E. *Etude des effets de bord des appels de procedure et de fonctions dans le langage PASCAL*. Thèse de 3ème cycle, Université Paris XI, Oct. 1979.

10. Tennent R.D. *Language Design Methods Based on Semantic Principles*. Acta Informatica 8 (1977), 97–112.