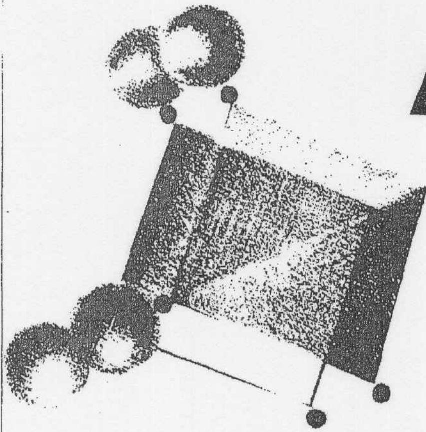
Cépage: A Software Design Tool



By Bertrand Meyer

For many years, software engineers have provided engineers of other fields with advanced design tools that considerably facilitate their jobs, relieve them from tedious tasks, put them in control of the design process, and help them turn out quality products. But the tools used by software designers themselves look quite primitive in comparison: the standard design environment includes a text editor, a compiler, perhaps a debugger, but hardly anything that could be characterized as a design system.

This article presents a system—Cépage—whose aim is to provide software designers with facilities similar to what is known in other application areas as computer-aided design.

Cépage (pronounced *sea-page* or *say-pajj*) is intended to make the full power of computer-aided design available to software developers in practical environments. It frees its users from many of the tedious tasks traditionally associated with the construction of programs and other software-related documents and allows them to concentrate on the really important aspects of software design. The aim is to use Cépage as the basis for a complete software development environment supporting all activities of software design.

Visual and structured

Cépage relies on a simple but powerful idea: to allow visual manipulation of structured documents.

What do we call a structured document? The most obvious example is a program written in some high-level language like Pascal, C, Ada, or FORTRAN, with their hierarchical structure (program subprogram, block, instruction, expression, and so on). Such programs are indeed one of the primary targets of Cépage but by no means the only one. In fact, any document with an interesting enough structure is an equally good candidate for handling by Cépage. This description includes such software documents as specifications, designs, or schedules.

But technical documents whose structure in a given environment is often standardized can also be handled by Cépage. For example, a certain company or project might require that technical reports begin with a header page of a normalized form (author's name, title, abstract, date, approval, etc.), and continue along a formal structure (sections, paragraphs, etc.). The usual way to deal with structured documents on a computer is to use a text editor—in other words, to forget about the structure. All a text editor sees is a flat sequence of characters or lines.

In contrast, a structural editor such as Cépage will use the structure as the basis for all manipulations of the documents. Many benefits may be expected from an intelligent system that knows the specific properties of the objects it handles, rather than treating, say, an Ada program the same way as a meeting report or a user's manual.

Structural editors have existed before; references to some of the most significant projects are shown in the accompanying sidebar. But I think Cépage is outstanding among these tools primarily because of the following features:

- The advanced visual interface through which documents are not only manipulated but actually seen.
- The ease of tailoring the system to any new language, making Cépage a versatile tool adaptable to many different applications.
- The open architecture of the system, which allows it to be interfaced with other tools and makes it a promising candidate as a basis for a complete software development environment.
- The ambition of the entire system design to take structural editing out of the laboratory and make its exciting potential available to practicing programmers in ordinary environments.

Showing the structure

As already mentioned, a structural editor manipulates documents in terms of their structure. This is all good and well, but with interactive systems it is not enough to use adequate representations. One must also present the user with faithful images of these representations.

Many early structural editors were rather inadequate in this respect. They used a primitive, often line-by-line interface. Cépage relies on an advanced interface, modeled on the concepts of direct manipulation pioneered by Smalltalk and analyzed by Ben Shneiderman in "Direct Manipulation: A Step Beyond Programming Languages."

Shneiderman studied many successful interactive systems in such diverse areas as computer-aided instruction, computer-aided design, games, and others. He attributed their success in large part to their ability to provide users with a clear

Other structural editors

Many of the basic ideas for structural editors were contained in W. Hansen's EMILY System.[1] The best known structure editors are Mentor, developed at INRIA,[2,3] Gandalf from Carnegie-Mellon,[4] and the Cornell Program Synthesizer.[5] A more recent tool with graphic facilities is Pecan.[6,7]

References

1. Hansen, Wilfred J. "Creation of Hierarchic Text with a Computer Display." ANL-7818, Argonne, Ill.: Argonne National Laboratory, 1972. [Also as dissertation, Computer Science Dept. Stanford University, Palo Alto, Calif., 1971.]
2. Donzeau-Gouge, Véronique, Gérard Huet, Gilles Kahn, and Bernard Lang. "Environnement de Programmation Mentor: Présent et Avenir." In *Actes des Troisièmes Journées Francophones sur l'Informatique*, Genève, 1981.
3. —. "Programming Environments Based on Structured Editors: The MENTOR Experience." In *Interactive Programming Environments*, edited by David R. Barstow et al., pp. 128-140, New York, N.Y.: McGraw-Hill, 1984.
4. Notkin, David, et al. "Special Issue on the Gandalf Project." *The Journal of Systems and Software* 5, no. 2 (May 1985): 91-178.
5. Teitelbaum, Tim, and Thomas Reps. "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment." *Communications of the ACM* 24 (Sept. 1981): 563-573.
6. Reiss, Steven P. "PECAN: Program Development Systems that Support Multiple Views." In *Proceedings of Seventh International Conference on Software Engineering*, pp. 324-333, Orlando, Fla., Mar. 26-29, 1984.
7. —. "Graphical Program Development with PECAN Program Development System." *SIGPLAN Notices (Proceedings of ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, Pittsburgh, Pa., Apr. 23-25, 1984, Ed. Peter Henderson), vol. 19, no. 5, May 1984, pp. 30-41. [This issue is also *Software Engineering Notes*, vol. 9, no. 3.]

picture of the current progress of the session and the internal state of the system at each step of a session.

This type of system is in contrast to the all-too-common silent movie systems, which are so shy that they do not tell you anything about what's going on. Of course, one should not say too much either; much of the challenge in designing good interactive systems is in finding the right compromise between undue terseness and unwanted verbosity.

So you want to tell the user what's going on. But what do you show? In a text editor the answer is easy: you display some contiguous excerpt of the text being handled. If the editor is full screen, like IBM's SPF or Vi on UNIX, this excerpt is going to fill a screen or window—say 24 lines of contiguous text. Figure 1 shows a typical screen from an editing session with such an editor, where the text being edited is an Ada program.

Unfortunately, the selection of displayed information at any point during the session—a screenful of contiguous text—has very little to do with the underlying structure of the document. Programmers who use text editors for composing programs know this well. They spend much of their time going back and forth from one end of the file to the other, chasing for structurally related but physically remote elements—the infamous scrolling syndrome.

When designing Cépage, we came to the conclusion that such an approach was unacceptable for a structural editor. It would be useless to have a tool relying internally on the document's structure if we were not able to display this structure. What we had to provide was a structural view at varying levels of abstraction.

Again, the driving metaphor is computer-aided design. For example, with a good CAD system for designing electronic systems, you may traverse the structure of your system, choosing to see at each step a graphical representation at the level of the whole system, a subsystem, a wafer, a circuit, a gate, a transistor, etc. For programs the representations are more likely to be textual than graphical (although work has been done on graphical programming techniques), but the same principles apply. For example, a representation of the Ada program in Figure 1 at the top level should be something like that shown in Figure 2.

Here the elements in italics (such as <*5 declarations*>) represent program elements that are present but may not be shown in detail due to lack of space. We say that such elements have been abstracted. By abstracting elements, it is possible to give a clear view of the structure at a certain level in the document. Here we see immediately what procedure *PROCESSOR* is made of.

If one wants to see an abstracted element, say the *procedure_body* of procedure *RESTART*, some of the context will have to be sacrificed. This is achieved in Cépage simply by moving the cursor or mouse to the corresponding place on the screen and requesting a zoom. Since the screen is not large enough to show everything, some details of the enclosing unit will disappear to make space for the procedure body; they may be shown again thanks to the unzooming operation. In any case, the view shown will be a structural one, corresponding to a meaningful syntactic structure rather than an arbitrary grouping of contiguous elements.

In Cépage, the abstraction mechanism is entirely automatic. From user requests the system determines what should be shown and what should be abstracted. To this effect it uses a sophisticated display algorithm that takes into account both the document structure and the available window space.

Excerpts from a session

There is more to say about the principles of Cépage, but since I am claiming so loudly that the system is a visual one, it is better to interrupt my presentation at this point for a small demonstration.

You are using Cépage on an unspecified display. In this article, various font conventions (italics, bold, etc.) are used to distinguish the different types of elements. On an actual terminal, Cépage relies on the facilities provided by the hardware: fonts on a black and white bitmapped screen, different colors on a color display, various levels of highlighting, etc.

Figure 3 shows the picture you might have at a given moment in a Cépage session. Actually, Figures 3-6 do not show

the whole screen but the main window, devoted to the main active document. In practice, the screen contains other windows for such things as session information and help messages. Also, there is usually not just one active document but a catalog of documents. The others are usually copies of parts of the main document that are kept for purposes of partial modification, copy, move, etc.

It appears from Figure 3 that you are working on a program in some Pascal-like, slightly Ada-ish language. The most original feature of this program is that it is not complete. It contains not only true program elements in bold—for example, **label** and **until**—but also things in regular face— like <Instruction> and <Boolean_expression>—which represent as yet unexpanded program parts. (The reader who remembers compiler courses will know these parts as nonterminals.) They are distinguished from the expanded parts by the angle brackets and by the regular face font (or, on a particular terminal, by a different color).

Texts such as the one displayed here are called partially expanded documents (we use the word "document" rather than "program" to emphasize again that the underlying language could describe structured objects other than programs). Partially expanded documents are the basic entity that Cépage handles. Of course, eventually a document should be totally expanded, and Cépage can then generate a textual version of the finished product. In the case of a program, this text may be passed on to a compiler, for example; other kinds of documents might be handed over to a text formatter such as *troff* on UNIX.

There is also an instance of *<Instruction>* in the repeat loop at the beginning of your program body that appears in italics (again, it might be a color) rather than regular face. Similarly, we see *<7 Instructions>* a little below. These are examples of abstracted elements (abbreviations for elements that cannot be shown in detail for lack of space). The expansion of the instruction in the repeat loop might contain as many lines as needed.

Note that when a compound element is abstracted it may be useful to display a comment associated with the element,

A contiguous program fragment

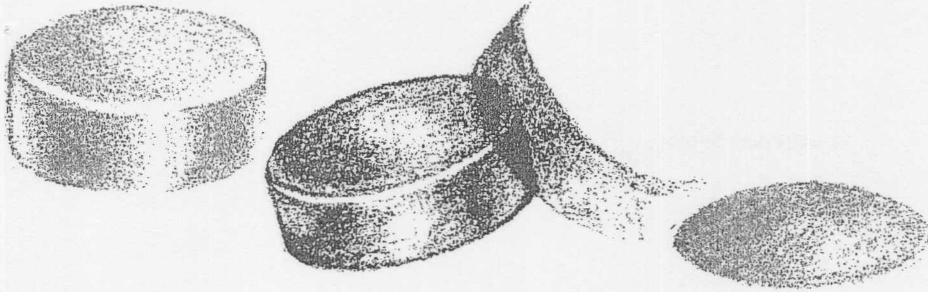
```
REMAINDER: REAL;
NB_USERS: INTEGER := 0;
TERMINALS: constant := 5;
package STOCK is
        LIMIT.T: constant := 1000;
        TABLE: array (1 . . . LIMIT) of INTEGER;
        T_COUNT: INTEGER := 0;
        procedure RESTART;
        procedure BACKUP (F: FILE);
end STOCK;
package body STOCK is
        procedure RESTART is
        begin
                INITIALIZE;
                CHECK_INVENTORY;
                for N in 1 . . . LIMIT loop
                        TABLE (N) := N;
                        T_COUNT := T_COUNT + N;
                end loop
        end;
procedure BACKUP (F: FILE) is
        begin
                TABLE(X) := TABLE (X) + SMALL;
                while ACTIVE loop
                        if NB_USERS > 5 then
```

Figure 1.

A structured program view

```
procedure PROCESSOR is
        <5 declarations>;
        package STOCK is <package_specification> end STOCK;
        package body STOCK is
                <2 declarations>;
                procedure RESTART is <procedure_body> end;
                procedure BACKUP (F: FILE) is <12 instructions> end;
        begin RESTART end STOCK;
        procedure UPDATE (X: INTEGER) is
                <4 declarations>
        begin <7 instructions> end;
        procedure REMOVE (F: FILE) is
        begin <23 instructions> end;
begin — procedure PROCESSOR
        RESTART;
        while ACTIVE loop
                if NB_USERS > 5 then
                        <6 instructions>
                else
                        BACKUP
                end
end PROCESSOR;
```

Figure 2.



rather than just an indication like < *Instruction* >, which is not very informative. It is indeed possible with Cépage to attach a short comment to an element; the comment will be displayed if the element is abstracted. It may be entered either after or before the element has been expanded, allowing for both bottom-up and top-down software construction.

Of course, at some point you may want to see some of the abstracted part. Nothing could be easier: just move the cursor to some position in the < *Instruction* > in italics and press a mouse button (or function key, depending on the terminal). Of course, as you go in you will lose some context, which you may see again by moving out again, using the corresponding option in the menu.

For the moment, however, you are interested instead in developing your program a little more. You have decided to expand the < *Instruction* > that appears just after the *repeat* loop; thus you have brought the cursor (represented by the hand on Figure 1) to the window in which the word "instruction" appears. You look and choose the Expand option, again using whatever selection medium is available: mouse to point in the menu, function key, etc.

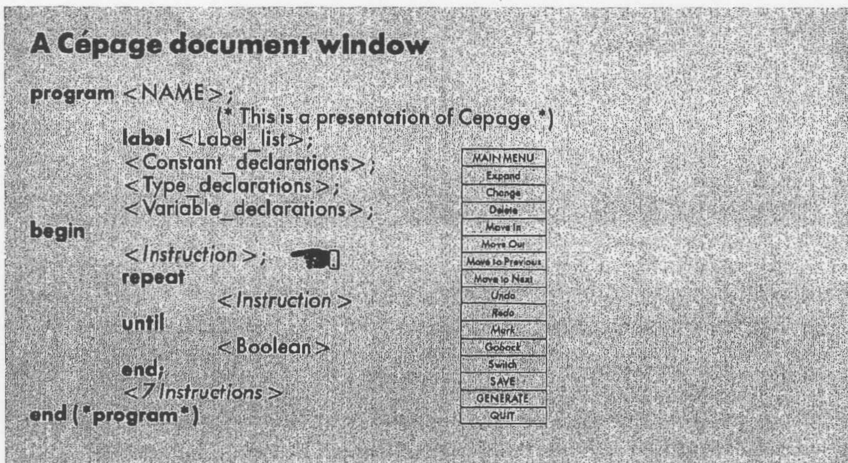


Figure 3.

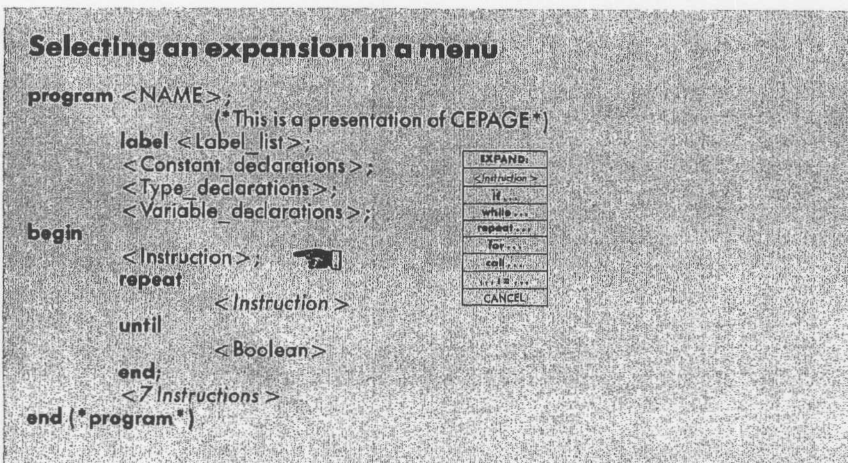


Figure 4.

The basic interaction with Cépage is normally done by show and select. In other words, you indicate a position on the screen and select a function from a menu. S and S is a very productive way of dealing with computers interactively; the effectiveness of this approach is backed by extensive psychological studies. Of course, the S and S principle is at its best when the display and the selection device (mouse, joystick) are adequate.

Once you have said that you wanted to expand a particular instruction, something new will appear on the screen. The text of the document does not change, but a new menu pops up, listing the set of possible instructions in the language at hand (Figure 4).

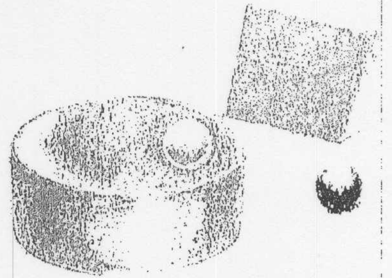
The new menu allows you to select the type of instruction you want. You do not need to type any keywords (such as *if*); you make a choice from the menu and the system will take care of generating the proper syntax for you.

Software developers have more interesting things to do than type. The problem is not so much typing itself as the fact that it detracts your attention from other, more important concerns. (However, you may also type the beginning of the instruction if you prefer to work in this fashion.)

Assume you decide you want a conditional instruction and, with any available selection facility, choose the corresponding item in the menu. The system generates the resulting structure. Figure 5 shows what now appears on the window. The part which previously read < *Instruction* > has been replaced with the syntax for a conditional instruction.

Note that up to now you haven't used an alphabetic keyboard; the mouse has served you well enough. You do not have to use the mouse. You could type phrases, or meaningful beginnings of phrases, if you preferred, or you may work by S and S. Which solution makes more sense depends on your personal taste and on the power of the terminal hardware available.

At some point, for elements such as expressions, it may become tedious to have to describe the structure—you just want to type in the stuff. For the elements of lowest levels, such as identifiers or constants, this is the only possibility anyway since they have no further structure. To enter such elements you just type them



at the place where they appear. They will be parsed immediately by the system.

For example, you may wish to resort to typing when entering the Boolean expression of our newly built conditional instruction, as shown in Figure 6.

It is important to note that the user has a choice at all levels between menu selection and typing. When typing is chosen, an added advantage of structural editors is that you may type just the beginning of a phrase, provided it is long enough to dispel any ambiguity. For example, typing just *while* at a place where an instruction is expected is enough for the system to generate the entire pattern for a *while* loop.

Other features of Cépage resemble those generally available in text editors: delete, copy, move, search, replace, yank (put aside for later use), etc. There is an important difference, however, since with Cépage all such manipulations may only apply to syntactically meaningful parts of the partially expanded document. For example, if we have:

```
if x>0 then
    | b := [c] + 1 |
else
    a := 5;
    call P(x)
end
```

and want to apply an operation such as delete or copy to a part of the document containing *c*, then this part may only be one of the boxes shown. On the other hand, there is no way to replace *else call* by *; goto* since neither pattern corresponds to a syntactic entity.

Language independence

Language independence is one of the main aims of the design of Cépage. Why should we be so interested in preserving adaptability to various languages? This emphasis has several reasons.

The first reason simply comes from an assessment of the situation. Clearly many programming languages are being used today and there is little prospect for unification in the near future. A tool tied to a single language would have a limited practical interest.

On the other hand, entirely different languages are not the only case when you need language adaptability. Very often local programming environments (companies, laboratories, universities, etc.) have specific variants of programming languages (for example, Turbo Pascal, VMS Pascal, and IBM Pascal), which differ in small but significant details. Adapting a tool to such variations should be a simple matter.

Third, methodology-conscious projects are increasingly defining programming standards (such as comment conventions, exclusion of certain constructs, etc.). It may be good to have such standards, and better to control their application, but nothing can beat using a program construction tool where the standards are built-in, having been integrated into the language. A modest example shown by the preceding session was the automatic addition of a specific comment at the end of a main program, but much more interesting conventions can be supported.

Finally, many types of document structures, which may not always be thought of as languages (such as a standard structure for technical reports), may benefit from a versatile tool that will automatically enforce the observation of the structure.

With these remarks in mind, Cépage was designed as a language-independent system where the structure of the language is a parameter. To define a new language for Cépage, a formalism is used: language description language. An LDL document is the description of a certain language, given as a sequence of constructs. Such a construct is shown as follows, and corresponds to the *while* loop of Pascal.

```
construct While_loop
-- "While" loop in Pascal
short "while . . . "
abstract
    aggregate
        test: Expression ;
        body: Instruction
    end
concrete
    format
        "while" test "do" body
    end
end
```

This description comprises the following elements:

- A full name, *While-loop*, used to references the construct.
- A short name, *while* . . . , used for display (for example, in choice menus).

Result of expansion

```
program <NAME>;
                (*This is a presentation of CEPAGE*)
        label <Label_list>;
        <Constant_declarations>;
        <Type_declarations>;
        <Variable_declarations>;
begin
        if <Boolean> then
                <Instruction>
        else
                <Instruction>
        end if;
        repeat
                <Instruction>
        until
                <Boolean>
        end;
        <7 Instructions>
end (*program*)
```

MAIN MENU
Expand
Change
Delete
Move In
Move Out
Move to Previous
Move to Next
Undo
Redo
Mark
Goback
Switch
SAVE
GENERATE
QUIT

Figure 5.

■ An abstract part, giving the structure of *while* loops, which comprises two components: an expression (the test) and an instruction (the body).

■ A concrete part, which gives the external representation of a loop with these components, indicating where the keywords should be placed.

This is a simple but typical example of construct description. Note that there is no need in normal cases to give formatting codes. Indentations, line feeds, etc., are automatically taken into account by the display algorithm of Cépage (but things are not quite so simple in a language with strange formatting requirements, such as FORTRAN, for which the concrete part of some LDL construct descriptions has to be more elaborate).

An added benefit of language adaptability is that Cépage may be used to perform many of the functions of traditional preprocessors that are used to artificially enrich languages.

A typical case for using preprocessors is to add instructions such as *while* loops to FORTRAN, which does not include such a construct. The preprocessor will transform programs with while loops into ordinary FORTRAN programs where the loops are expressed by conditional and *goto* instructions. With Cépage, it is easy to extend the LDL description of FORTRAN with a *while* construct whose concrete part is the actual FORTRAN structure, namely:

```
label IF (<test>) THEN
          <body>
          GOTO label
label END IF
```

In this interactive pattern, the user will select the pseudo-FORTRAN *while* pattern, and the system will generate correct FORTRAN code.

The designers of Cépage felt very strongly that writing a language description should be easy. Using LDL, the description of a language should take from a few hours (when working on a variant of a previously defined language) to at most a week (for an entirely new language). We expect many language descriptions to be obtained by imitation or modification of existing ones.

Of course, LDL descriptions will be done using Cépage. LDL is one of the first languages to be supported by the system.

System structure

To better understand the concepts and applications of Cépage, it is useful to explain some of the techniques underlying its implementation.

The general structure of the system is shown in Figure 7. Cépage was designed and implemented according to object-oriented techniques, and accordingly the structure is best described by explaining the main object classes (abstract data structures) used.

The abstract syntactic forest is a hierarchical data structure that provides an internal representation of the document being manipulated.

The external form is an equivalent representation, suitable for storage on external devices. Remember that Cépage works on partially expanded documents. If a session is interrupted before the expansion is complete, it must be possible to store the temporary state of the document in an appropriate form for later retrieval.

The display form is produced by the Cépage display system by mapping the internal document structure, as represented by the abstract syntactic forest, onto the available window space, choosing the elements that will be abstracted for lack of screen real estate, and performing the necessary formatting and indentation operations. As already mentioned, this process is entirely automatic. The user does not need to intervene but will get an appropriate view at each stage of the computation.

The display form is a list of windows that can be processed by another of our software tools, Winpack (a general multiwindowing screen package).

The text form may be generated for a totally expanded document for possible processing by other tools.

Finally, the grammar graph is the data structure representing the language. This is where Cépage's language adaptability comes in. The system kernel itself knows nothing about any particular language whatsoever; all the information is contained in the grammar graph, reflecting the LDL language description.

Further applications of this technique may involve more than one grammar graph. For example, two windows may be used concurrently to run two instances of Cépage, one for a program design language, the other for a programming language. If the grammar graphs are con-

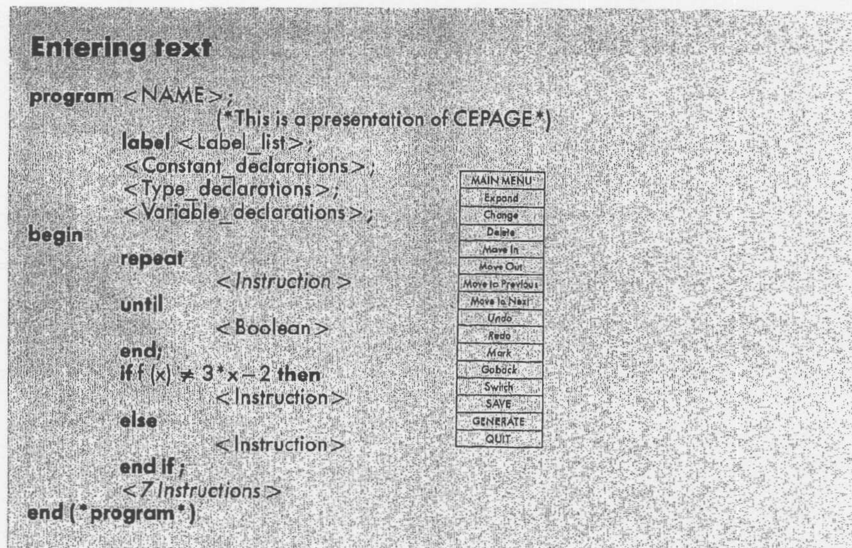
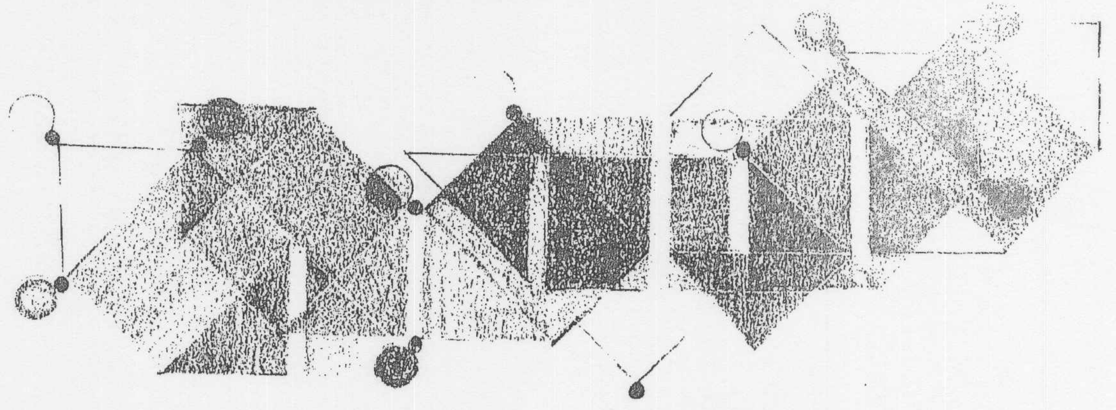


Figure 6.



nected, changes to the former may be immediately reflected in the latter.

Cépage functions

To allow the reader to get a better grasp of the whole scope of Cépage, we now give a systematic list of the functions that are or will be supported by the current version of Cépage.

Moving around. A basic set of functions makes it possible to move around a document in a manner consistent with its hierarchical structure: out (to enclosing context), in (to some subset of the current context), to the next element, and to the previous element.

Marking. The marking commands make it possible to take note of positions in the document while moving around and to come back to them later.

The three marking commands are: *Mark,* which marks the current position in the document; *Back,* which returns to the most recent position to which one has not yet returned; and *Forth,* which cancels the effect of the most recent *Back* command that has not yet been canceled in this fashion.

Expansion. The expansion function makes it possible to expand a previously unexpanded element of the document, as shown earlier. It is executed according to the information contained in the grammar graph.

Cancel and Modify. *Cancel*ing an expansion puts an expanded node back into the unexpanded state. In the case of a choice node, the *Modify* function allows the user to make a new selection; as much as possible of the initial expansion will be carried over to the new one. For example, when transforming an *if. . . then . . . else* instruction into a *while* loop, the Boolean expression and the *then* part of the conditional instruction will be transferred to the loop.

Comment and Explain. The *Comment* command attaches a comment to a node (expanded or not). The *Explain* command changes the display mode so that comments will be displayed with particular emphasis.

Search and Replace. *Search* and *Replace* correspond to traditional editor functions. In Cépage, however, the search pattern and (in the *Replace* case) the replacement are structured elements similar to the document being edited: the editor is called recursively to enable the user to define them (in special windows).

Selection. The selection facility allows the user to make a choice among a set of predefined possibilities and allows the system to determine which item was selected. The way in which the list of choices is displayed and the user makes a selection (pointing with a mouse in a menu, pressing a function key, typing an ordinary key, etc.) depends on the terminal hardware.

Parsing. The parsing function makes it possible to read a text typed in by the user and to build the corresponding syntactic structure (subtree of the abstract syntax tree). The text can be incomplete. The parsing method used in Cépage allows the system to fill in the missing parts if the text typed is incomplete but unambiguous.

Undo and Redo. *Undo* makes it possible to back up to previous states of the editing session by canceling the effects of previously issued commands. *Redo* cancels such a cancellation.

Record and replay. The record and replay facility makes it possible to archive the succession of commands issued during an editing session and to replicate them. It thus allows recovering from a system crash.

Catalog management and copy. Catalog management keeps several documents during an editing session, one of which is the active document, the others constituting the catalog. This function allows the user to select an element of the catalog as the new active document, to copy part of the active document into a new entry of the catalog, or to copy an element of the catalog onto an unexpanded node of the active document.

Delimit. The delimiting function enables the user to define a part of a document to be used as a parameter for a function such as *cancel, copy,* etc. Since the moving around functions are particularly simple to invoke, delimiting is mainly useful for selecting sublists.

Save and restore. The *save/restore* function copies documents (which may be partially or totally expanded) from memory to files and back, using an appropriate external representation.

Library management. Library management maintains data bases of (partially

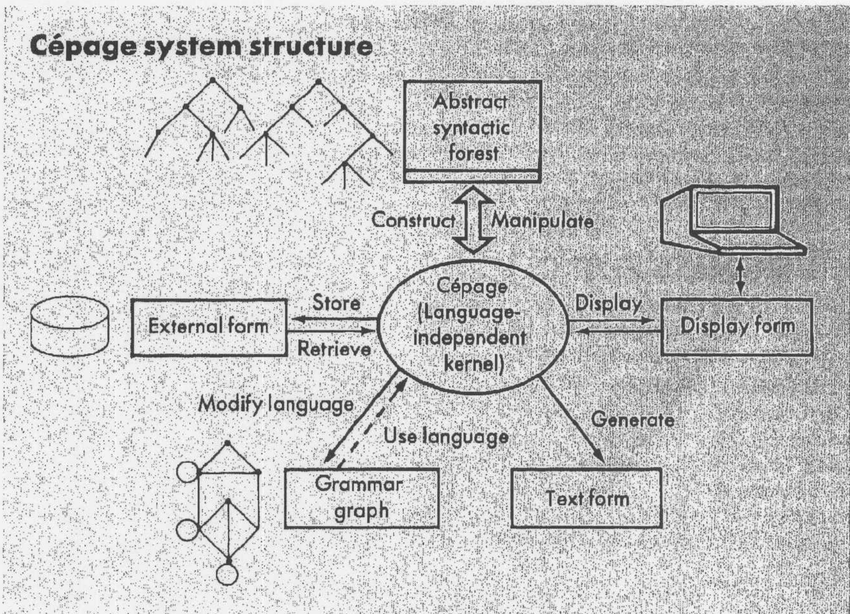


Figure 7.

C DYNAMO

NEW MICROSOFT C $249.95

(NEW! Version 4.0 See offer below. Good till 9/30/86)

C FUNCTION LIBRARIES
POWER WINDOWS
SUPERFONTS FOR C
B-TREE LIBRARY & ISAM
PCNX, UNIX™ *WORK ALIKE*
C-TERP *INTERPRETER FOR C*
POLYTRON *VERSION CONTROL SYSTEM*

™ AT&T

C FUNCTION LIBRARY

BEST YOU CAN GET
325 FUNCTIONS
FULLY TESTED
SUPERB DOCUMENTATION
BETTER FUNCTIONS

Most complete screen handling plus graphics; cursor/keyboard/data entry, 72 string functions with word wrap; status and control; utility/DOS BIOS/time/date functions; printer control & more. Special functions. Functions you NEED but don't have! Full source code. No royalties.

4 disks $129.95

POWER WINDOWS

MOST POWERFUL YET
POP-UP WINDOWS FOR
Menus/Overlays
Help Screens
Messages/Alarms
ZAP ON/OFF SCREEN
FILE-WINDOW MANAGEMENT
COMPLETE CONTROL OF:
Cursor
Attributes
Borders
AUTOMATIC
Horizontal & Vertical Scrolling
Word Wrap
Line Insertion

The most powerful, flexible and easy to use windowing package available! Many types of menus. Highlighting. Move data between files, keyboard, program and windows. Status lines. Change size/location/overlapping. Move/add/delete/cascade windows. Full source code. No royalties.

3 disks $129.95

SUPERFONTS FOR C

SUPER SIZE CHARACTERS
Monochrome adapter
Color/graphics adapter
8 FONT LIBRARIES

Dramatic high impact screens/titles/ messages. Use with or without windows. Great for projection TV. Create own font & image libraries or use ours. Use with windows for special effects or animation. Full source code. No royalties.

Font and Function Library $49.95

High Quality Software Since 1982

NEW!

VERSION CONTROL SYSTEM
FOR PROGRAM DEVELOPMENT
Store/retrieve multiple versions
Release & configuration control
Auto-logging & display of changes
Invaluable audit trail of changes

PVCS gives you complete control of program development, including history & recoverability of all changes. Permits multiple parallel lines of development, multiple versions of your program. The longer the program the more essential PCVS is. Don't develop another program without it. Saves critical time, adds control and flexibility

Single user license $395.00

B-TREE LIBRARY & ISAM DRIVER

POWERFUL DATA MANAGER
Fixed/Variable length records
FAST! EASY TO USE!
16.7 MILLION RECORDS/FILE
16.7 MILLION KEYS/FILE

Fast B-tree indices. Add/remove keys Find first/last/next/any key. Find keys by Boolean selection. Read/ write/delete or add records to file.

Full source. No royalties. $129.95
***MAKE** Utility (Snake) $59.95*

C-TERP

INTERPRETER FOR C
NO COMPROMISE, FULL K&R
BUILT IN SCREEN EDITOR
FAST, FAST COMPILE/LINK
USE EXTERNAL LIBRARIES!
SYMBOLIC DEBUGGING
SINGLE STEPPING
RAVE REVIEWS!

2 disks and manual $299.95

COMBINE AND SAVE!

C LIBRARY plus C WINDOWS
BOTH for only $179.95
+ SUPERFONTS FOR C $199.95
(A $310 VALUE)

C BUSINESS LIBRARY
INCLUDES C FUNCTION LIBRARY, POWER WINDOWS, SUPERFONTS FOR C, B-TREE LIBRARY, ISAM
ALL for $299.95
(A $440 VALUE)

C PRODUCTIVITY LIBRARY
INCLUDES C FUNCTION LIBRARY, POWER WINDOWS, SUPERFONTS FOR C, and C-TERP INTERPRETER
ALL for $419.95
(A $610 VALUE)

C TOTAL LIBRARY
INCLUDES C FUNCTION LIBRARY, POWER WINDOWS, SUPERFONTS FOR C, B-TREE LIBRARY, ISAM and C-TERP C Interpreter
ALL for $499.95
(A $740 VALUE)

SPECIAL OFFER*

(expires 9/30/86)

	$LIST	$SALE	$SAVE
MICROSOFT C COMPILER	450	250	200

	$LIST	$SALE	$SAVE
Version Control System	395	265	130
Virtual Memory Manager	199	99	100
Polytron PowerComm	179	89	90
PolyFORTRAN Tools 1	179	89	90
Poly Cross Ref (C & Assembly)	178	88	90
Poly DeskPlus & Poly Archivist & PolyCryptographer	185	95	90
Polylibrarian II	149	69	80
Polymake	99	59	40
PolyOverlay	99	59	40
OPERATING SYSTEMS *Multi-User Multi-Tasking Networking*			
PCNX™	99	19	80
PCVMS™	99	19	80
O/S TOOL BOX™ (Build your own operating system)	99	19	80

™WENDIN SOFTWARE

*with purchase of any Entelekon combination library listed above. Limit 1 Special Offer item per customer. Offer expires 9/30/86.

THE C DYNAMO FAMILY... YOUR COMPLETE SOURCE FOR C

Entelekon

SINCE 1982

12118 Kimberley, Houston, TX 77024 713-468-4412 VISA-MASTERCARD-CHECK-COD

CIRCLE 25 ON READER SERVICE CARD

or totally expanded) documents, stored under the external representation already mentioned.

Generation. The *generation* function creates textual versions of totally expanded documents.

Interactive language description and modification. The interactive language description and modification function uses Cépage itself to enter and modify LDL descriptions (grammars). It thus relies on a grammar graph obtained from a description of LDL in LDL. This function provides for incremental language modification. In other words, it makes it possible to construct and modify a grammar graph in a stepwise fashion, as the corresponding LDL description is being developed and updated.

Semantic checking. The semantic checking function makes it possible to perform verifications on documents. It is only applicable if the language description includes the definition of the corresponding semantic constraints.

Execution. The *execution* function makes it possible to execute the active document, considered as an executable program. It is only applicable if the language description includes dynamic semantics for each operand type. A partially expanded document may be executed: when execution reaches an unexpanded element, the user is interactively asked to provide the results of the execution. This facility is a first step toward making Cépage into a tool for rapid prototyping and program testing.

Display. The *display* function displays an abstract syntax tree or subtree in a given window area, finding the best representation it can.

Library of primitives. The library of primitives is a set of procedures that enables outside programs to access all of the Cépage functions mentioned and the Cépage data structures. By making these Cépage internals accessible to other software tools, it is planned that Cépage will be used as the kernel of a more complete software environment, in which tools of various kinds (such as static program analysis, complexity analysis, program transformation, testing compiling, rapid prototyping, text processing, etc.) will be able to take advantage of the powerful set of basic data structures and functions provided by Cépage.

Implementation

A prototype version of Cépage was developed at Electricité de France in 1982-83 by the author and Jean-Marc Nerson for an IBM mainframe environment running MVS and TSO, with 3279 color terminals. The current version is an entirely new development, applicable at the moment to any UNIX or XENIX environment. Versions for MS-DOS systems and VAX-VMS are planned.

The development, undertaken by Interactive Software Engineering, Inc., uses a Sumitomo Electric workstation (U-station) running UNIX System V, with a color bit-map display.

The design and implementation were done with an object-oriented language, Eiffel, emphasizing reusability of software through techniques of multiple and repeated inheritance and security through static type-checking. The underlying Dynamem memory management system uses virtual memory and parallel garbage collection. Screen access is through Winpack, a multi-windowing screen management package.

We view Cépage, Eiffel, Winpack, and Dynamem as bricks toward a powerful, integrated software development environment emphasizing the efficient production of high-quality software.

The next step

This article has emphasized three aspects of Cépage:

- Editor—the system for creating and modifying documents at the source language level
- Program development system, with facilities for program checking, testing, and rapid prototyping
- Basis for a programming environment.

These goals are short-term. To conclude with a more futuristic view, we will now present a more remote but very promising application of this system in solving the problem of software reusability. We may call the Cépage solution pattern-based interactive program generation.

Most of the software written today is of a repetitive nature. A number of basic program patterns (counting, searching, sorting, comparing, exchanging, assigning, creating . . .) exist on which programmers compose endless variations. Most of this work is done at the lowest reasonable level, that of common languages.

The use of shared, standard components is not—despite a few exceptions such as libraries of numerical software—commonplace. This situation stems in part from the fact that each new situation may be slightly different from the ones encountered previously. For example, even though most search routines share a general organization (go to the beginning of the table, loop until either the required element has been found or the subset of the table in which it may appear has been exhausted, report "found" or "absent"), the representation details will considerably vary from one case to the next.

It is not easy to construct software components that provide a suitable answer to the problem of reusability. Consider the simple problem of providing users of a computing center with a tool for sorting arrays. Assume the algorithm chosen is Quicksort, which is well explained in computer science textbooks, so you don't have to worry about this aspect of the question. A particular problem instance is characterized by how elements will be compared and how they will be exchanged. The solutions open to a software toolsmith are the following:

A. Provide procedures for the most frequently occurring cases, for example, increasing and decreasing sort of integer, real, etc., arrays.

B. Provide a single procedure (or operating system command) with many parameters or options.

C. Provide a single procedure with two procedure parameters corresponding to the comparison criterion and the exchange mechanism.

D. Have a sorting procedure skeleton and manually create a tailor-made version for each user who requests it, filling in his or her particular sorting criterion and exchange mechanism, with the help of a text editor.

Whitesmiths, Ltd. Has The Compiler You Want On The Machine You Use.

For over seven years Whitesmiths, Ltd. has focused its efforts solely on developing and supporting a family of quality systems software. Today, Whitesmiths is the only company offering compatible C and Pascal native and cross compilers for the full spectrum of computers on the market—from the IBM PC to the IBM 370, from the DEC Micro-11 to the VAX 8600, and all of the most popular processors in between.

As a forerunner in the development of C and Pascal compilers, Whitesmiths has played a major role in defining and refining the standards for ANSI C and /usr/group libraries.

The result is a product line built from the ground up to provide a uniform environment for the professional applications developer. Identical source code across all machine architectures; support for ROM-based programs; a uniform run-time environment; and the ability to mix code in assembler and other high level languages are just a few of the many features that comprise these superior compilers.

If you need a C or Pascal compiler for your machine, give Whitesmiths a call at **1-800-225-1030.** Chances are, we have what you want.

Whitesmiths, Ltd.

97 Lowell Road, Concord, MA 01742 • (617) 369-8499 / Telex 750246

INTERNATIONAL DISTRIBUTORS: **FRANCE**, COSMIC S.A.R.L., 52 Quai des Carrieres, 94220 Charenton Le Pont, Paris, (14) 378-8357 • **GERMANY**, GEI, Gesellschaft fuer Elektronische, Informationsverarbeitung MBH, Pascalstrasse 14, D-5100 Aachen, 02408/13-0 • **JAPAN**, Advanced Data Controls Corp., Nihon Seimei Otsuka Bldg., #13-4, Kita Otsuka 1-Chome, Toshima-ku, Tokyo 170, (03) 576-5351 • **SWEDEN**, Unisoft AB, Fiskhamnsgatan 10, S-14155 Goteborg, (31) 125810 • **UNITED KINGDOM**, Real Time Systems Ltd., P.O. Box 70, Douglas, Isle of Man, (624) 26021.

CIRCLE 95 ON READER SERVICE CARD

Solution A is too partial. In many cases, the users will want to sort an array of pointers, leaving the elements themselves in place, or use only part of the elements as keys, so it is unlikely that many actual cases will be covered by the library routines.

In solution B, the options may cover a larger number of cases, but the tool will require coding many options and thus using a reference manual, a cumbersome and error-prone process.

Solution C will work but with great inefficiency since the procedures passed as parameters will be called repeatedly in the inner loops of the sorting program. The overhead, which is typically a factor of 10, will be unacceptable in many cases.

Solution D, using an editor to generate tailor-made versions, is tedious and error-prone.

Solution E implies learning the conventions of the particular macro-processor on hand, which may be at odds with those of the programming language used, even if they were designed by the same group. For example, on UNIX, the macro-processor embedded in the C compiler and the M4 macro-processor have different conventions regarding parentheses, commas, reserved words, etc. Futhermore, macro-processing is not interactive—the user must first provide actual arguments in the adequate formalism, then wait for the macro-processor to generate a text for inclusion in the program.

Structural editing may provide a better solution. A simple idea is to apply the notion of abstract syntax. In the same way as a *while* loop was described as consisting of two components, a test and a body, a sorting program may be defined as:

```
sorting = comparison; exchange
```

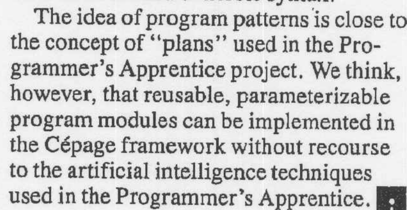
One can thus envision an extension of the editing process in which the user interactively describes the comparison criterion and exchange mechanism to be used in a particular instance, and the system automatically produce the displayable form:

```
if c then A else B end
```

from the description of the language, the user providing only *c*, *A*, and *B*. The only difference is that the amount of text generated by the system will be proportionally larger in the case of program generation.

We believe that such interactive, pattern-directed program generation is possible in the Cépage framework. The basic mechanisms are already present. In particular, since the language is a modifiable parameter, it is possible to extend the basic constructs such as conditional and loop with libraries of program patterns such as search, sort, or even payroll. Such patterns will be defined in the same way as basic language constructs—by their abstract and concrete syntax.

The idea of program patterns is close to the concept of "plans" used in the Programmer's Apprentice project. We think, however, that reusable, parameterizable program modules can be implemented in the Cépage framework without recourse to the artificial intelligence techniques used in the Programmer's Apprentice. ■



References

Card, Stuart K., Thomas P. Moran, and Allen Newell. *The Psychology of Human-Computer Interaction.* Hillsdale, N.J.: Lawrence Erlbaum Associates, 1983.

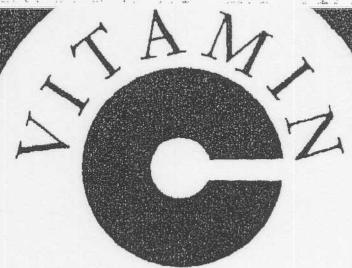
Meyer, Bertrand. *Eiffel: A Language for Software Engineering.* Technical Report TRCS85-19. Santa Barbara: Univ. of California, Nov. 1985.

Shneiderman, Ben. "Direct Manipulation: A Step Beyond Programming Languages." *Computer (IEEE)* 16 (Aug. 1983): 57-69.

Waters, Richard C. "The Programmer's Apprentice: Knowledge-based Program Editing." In *Interactive Programming Environments*, edited by David R. Barstow et al., pp. 464-468. New York: McGraw-Hill, 1984. Originally in *IEEE Transactions on Software Engineering* SE-8 (Jan. 1982).

Bertrand Meyer is a visiting associate professor at the University of California at Santa Barbara and president of Santa Barbara-based Interactive Software Engineering Inc. He was previously division head at Electricité de France.

Artwork: Anabella Gonzalez



It's good for your system.

FULLY Integrated, Data Entry Windows!

- Complete input formatting
- Unlimited Validation
- Full attribute control
- Multiple virtual windows
- Fully automatic, collision proof overlay and restore
- Print to & scroll background windows
- Animated window "zoom"
- Move, grow, shrink, hide, or show any window
- "Loop function" allows processing while awaiting input

AND MUCH MORE!

$149.95

Includes 100% source, tutorial, reference manual, examples, and sample programs. *Specify Microsoft, Lattice v2 or v3, Computer Innovations, Aztec, DeSmet, or Mark Williams. Ask about Unix.*

100% Money Back Guarantee

NOW ... VCScreen!

Our new interactive screen "painter" actually lets you draw your data entry windows! Define fields, text, boxes & borders. Move them around. Change attributes. Then the touch of a button generates C source code calls to the Vitamin C routines!

$99.95

Requires Vitamin C Library above. For IBM & compatibles.

For Orders Or Information, (214) 245-6090

Creative Programming Consultants, Inc. Box 112097 Carrollton, TX 75011-2097

Include $3 ground, $6 air, $15 overnight shipping, $25 if outside USA. Texans add 6⅛% tax. All funds must be in U.S. dollars drawn on a U.S. bank.

creative PROGRAMMING