

AN INTEGRATED, INTERACTIVE AND INCREMENTAL
PROGRAMMING ENVIRONMENT FOR THE
DEVELOPMENT OF COMPLEX SYSTEMS

Harald WERTZ

LITP / CNRS
2 Place Jussieu
75005 Paris

&

Université Paris 8
Dépt. Informatique
2 rue de la Liberté
93526 St.-Denis Cédex 02

France

Abstract : We are currently implementing a system to help experienced programmers during the development, implementation and debugging of their programs. This system, built on top of a screen oriented structural editor, offers possibilities to attach descriptors to every portion of the program and to maintain - simultaneously - different versions of the program being written, including tentative hypothetical versions. It comprises a mechanism to maintain minimal consistency between modified parts of code, the non-modified parts of code and the attached descriptors, as well as an evaluation module functioning in different modes : *normal evaluation*, *symbolic evaluation* and *checking evaluation*.

The standard programming aids, such as indexors, pretty printers, trace packages, undo- and history-facilities are generalized to handle the descriptors and unfinished programs as well.

1.0 INTRODUCTION

Constructing large and complex programs is rarely a well defined, orderly process proceeding from abstract specification to design and finally to implementation. It is a tedious, time consuming and complex activity, based often on some method of trial and error or successive refinement, while constructing, testing, then rejecting or modifying different algorithms, heuristics and representations used thus far.

Often even, the initial (formal or informal) specification of parts of the developing system will be modified as a result of new insights gained during the process of interactive design and implementation.

Special problems may arise when - as is usually the case in production environments - the system is developed by a group of several programmers, each one perhaps modifying (or wanting to modify) parts of the others' work.

In actual programming environments (PE) no on-line possibility exists to keep track of the developmental decisions, to access previous or *future* (1) versions of the system and to switch back and forth between different *viewpoints* of it. In addition, on-line possibilities to help to evaluate the *range of influence* of intended modifications do only rarely exist.

The construction of highly complex software systems not only creates problems during the development, maintenance, extension or transportation phase, it also creates crucial problems for the *comprehensive* interactive reading or annotation of it.

What is needed is a PE permitting the programmer to keep track of all his/her ideas, intentions and decisions underlying every part of his/her program, as well as a possibility to access them at every moment of his/her interaction with it. The programmer needs to have a possibility to create tentative versions of parts of the program, without destroying or otherwise fundamentally changing the structure of the already existing program. S/he should be able - after an incorrect decision - to go back to a previous version of the code, while keeping the actual version as an annotation of it, perhaps even with commentaries stating the reason for abandoning this line of development. A possibility should be offered to work simultaneously on different, independent versions of parts of the program, leaving open the *guided* merging of parts of those different tentative versions into a new and more elaborated one.

In addition, during the development of the program, the programmer needs a possibility to execute *unfinished* programs interactively - without being confronted with already anticipated 'fatal error' messages.

The PE I am proposing addresses all those issues: it is able to keep track - through an annotation facility - of all the modifications; it

(1) Since the programmer has the possibility to work on different versions of his program, he may well, when working on an 'old' version, be interested to consult some more recent one. The notion of 'future' is relative.

enables - through a multiple evaluation function - the simultaneous use of multiple versions of a program; it accepts temporarily limited solutions, and it permits the execution of unfinished programs and useful annotation of every part of the developing program.

These possibilities are obtained by means of three major changes in our LISP interpreter :

- a new representation of a program as a tree of *all* its versions until now constructed, along with possible attachments to every point in the tree
- a sophisticated, structure oriented editor through which all interaction with the LISP system takes place and which works on this tree representation of programs
- a special evaluator, able to work in different, programmable modes and contexts.

With this system, the programmer can, at every moment, interactively control the evolution of his/her software architecture.

In the following sections of this paper, I will, after exposing some preliminary design decisions, describe the significant aspects of the proposed PE, as well as the possibilities it offers. Finally, I will rise several as yet missing - but necessary - features and propose directions for further development.

2.0 PRELIMINARY DESIGN DECISIONS

In this somewhat technical, short paragraph I will describe some of the implementation details necessary to understand the feasibility of the proposed interactive programming environment.

Since we wanted our programming environment to integrate the - normally separated - activities of editing, executing and annotating of programs, the usual representation of a program as either a string of text (for the editor) or a execution adapted representation of code had to be unified in a single representation, combining code, text, commentaries and so on. I achieve this in representing the program as a tree, growing through the development process. It is then the task of the different modules working on this tree, to change the representation, that is the point of view one can have of this tree.

In LISP, as soon as one is confronted with the project to change the representation of a program from a linked list of code to a tree of all different versions of the program thus far constructed, enriched by all kind of possible descriptions (see Fig. 1), the use of the classical linked list is no longer adequate : there is no possible place to attach descriptors, nor is there any convenient possibility to have more than one successor to a node (at least as long as one is concerned about cost-effectiveness and does not want to pay too high a price in space (1)).

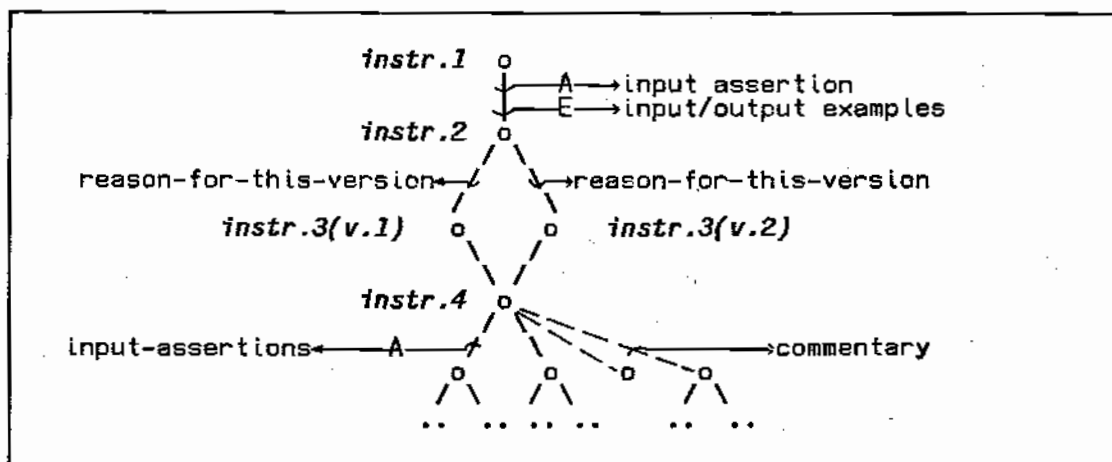
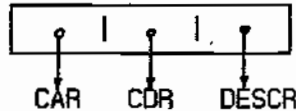


Figure 1 : part of a program-tree with attached descriptors

I have decided to maintain - simultaneously - two versions of LISP : one (the *production version*), which corresponds to the normal version of LISP

(1) to change the representation of a list in a tree, while continuously using linked lists, would double the number of CONS-cells.

and is used outside the PE for programs whose development may be considered as completely finished, and a second one (the *environment version*), where I don't work on simple lists, but on *triples* (as in TREET [Haines 65]) of a CAR, a CDR and a DESCR (called CSR in TREET),



where the CAR and CDR parts of the triple correspond to the standard CAR and CDR fields, while the DESCR field itself is divided in a field for the *descriptors* of the CAR and another one for those of the CDR of the actual triple. The DESCR field will contain the *attachments* to the different parts of the program.

Since the descriptors may contain modifiers for the evaluation process, such as other *versions* of the associated CAR and CDR, I have rewritten all the standard LISP functions in such a way as to be able to handle this new structures.

Except for the special case of commentaries attached to the code, the *descriptor* field can only be written using the editor. This implies that I also maintain two READ functions, one to read in normal S-expression and to convert them to this internal triple-representation (and which, as a side effect, converts commentaries to machine internal representation thus, by the way, making them accessible for use during run-time) and one for reading in S-expressions already in the form of triples.

The PRINT function is parametrized to print only *this* level (or point of view) which is actually under consideration. So, if the programmer considers version one of his program, all the other versions appear to be *hidden*.

The new EVAL function is described in paragraph 4, being a fundamental component of the PE.

The other representational change concerns variables: since one must assure dynamic access to previous values of each variable, they are represented as individual stacks (cf. dream-paper [Steele 79] and the design of the new HP 32-bit processor).

Even if these design decisions seem to be rather space inefficient, let me remark that these representations are only in use during interaction with the PE. Once the development and implementation of a program is finished, it is handled by a normal LISP interpreter or compiler.

3.0 THE EDITOR

All interaction between the programmer and the PE is monitored by the editor, which is an integrated part of the language. Externally no distinction is made between editing commands and execution commands, and at every moment full access to all the available commands is assured, enabling the user as well to execute function calls in the midst of an editing session as, during *normal* interaction with LISP, to edit the functions or data structures which need modification.

3.1 Internal Organisation of the Editor

The editor is directly working on the internal representation of the edited objects; in this way, the objects are continuously available for the normal evaluation process.

The core of the editor is similar to the Interlisp editor [Teitelman 78] or to EF [Greussay 78], with the additional feature that the visualisation of the edited objects is display oriented, meaning that in stead of successively printing out the newly accessed structures, a unique screen image of them is constantly updated. This isn't but a slight improvement over the classical structure oriented LISP editors, except that it permits a *pattern-oriented* editing as exposed in paragraph 3.3.

A much more important extension is that each edited modification *adds* new code, commentary or other information to the already existing in-core representation of it. An editing command *never* - except on very special occasions - deletes anything: a new *current* version is created, and the old one is kept on *historical* attachments. This means that a program is a tree of *all* previous versions of it, or - said differently - a program is a set of successive *viewpoints* of it. The current version is just a special way to travel down the tree.

Let us take a (very) simple example. Suppose the programmer has written a function, called FOO, to implement the factorial program:

```
(DE FOO (N)
  (IF (= N 0) 1 (* N (SELF (FOO (1- N))))))
```

and that for some reason s/he has modified this function so that it computes the sum of all integers up to N:

```
(DE FOO (N)
  (IF (= N 0) 0 (+ N (SELF (FOO (1- N))))))
```

Internally this function will then be represented as :

```
FOO
↳ (LAMBDA (N)
  (IF (= N 0) [(vs0 . 1)] [(vs0 . *)] N (SELF (FOO (1- N))))
    [(vs1 . 0)] [(vs1 . +)]
```

where *vs0* and *vs1* are just the version numbers. An absence of a version number indicates that the corresponding part of program is valid for all versions. To read the *current* program, it is sufficient to travel along the tree in following but the highest *vs-n* where ever different versions are encountered. At the same time this representation also permits access to any previous version of the program, it is sufficient for that to consider the corresponding version number as the highest possible path.

If for some reason the programmer wants to modify a previous version of the program - by this way creating a new intermediate version - the editor generates sub-versions to already existing ones. So, for example, if s/he re-edits version zero (*vs0*) of the program FOO and changes the recursion exit test to

```
(= N 1)
```

the internal version will be :

```
FOO
↳ (LAMBDA (N)
  (IF (= N [(vs0.0 . 0)] [(vs0 . 1)]
      [(vs0.1 . 1)] [(vs1 . 0)]
      [(vs0 . *)] N (SELF (FOO (1- N))))
    [(vs1 . +)]
```

Let me remark that the editor keeps the date of the changes, modifications or adds also in its internal representation, and that it is possible to communicate to the system some *reasons* for changes, which then are also saved in the internal representation, so, permitting a future reader to follow the design decisions which led to the current version of the program.

The new branches created at each modification automatically inherit most

of the attachments and (minimally) indicators are set saying that they refer to a previous version and possibly need some updating. A further research will be to construct some inference mechanisms which could (at least partly) automatically generate those updatings.

A psychologically important point in all interactive PE's (cf. [Weinberg 71] and [Shneiderman 80]) is not to overload the perceptive capacities of the user. That is why, during the editing sessions, all versions except the one under consideration, and all attachments are *invisible* or *hidden*, and may only selectively be rendered visible through special commands. This design decision makes it possible to edit a program exclusively on an arbitrary level of attachments, without being overloaded by the display of all the additional information contained in the program tree.

Actually, this important notion of *minimally necessary information* exists already in rudimentary form in the standard structured LISP editors [Teitelman 78, Greussay 78, Adda 81] in the form of the *star convention*, which I keep in our display oriented editor. This characteristic also enlarges the notion of an editor: it is not any more considered as a mere tool to write and modify programs, but also as a crucial instrument for *interactively reading* already existing programs. It permits to read a program in concentrating on selective viewpoints (versions, attachments) only. So, a program may be read only on the level of global commentaries, to get an overview of the working of the system, or on the level of logical assertions, or whatever point of view may be of interest to the programmer or reader. The standard reading is what corresponds to the 'normal' definition of the functions or data-structures.

3.2 The Attachments

The system recognizes, and the editor has a set of special commands to access and modify, some reserved standard attachments, built-in a priori. Those are :

examples

To every function and, more generally, to every portion of code may be attached a set of *concrete* examples. Those may, in the first place, be considered as a special case of commentary, helping the casual reader to understand the use of the associated program fragments. It is also, following an idea of Patrick Greussay, a way to control - during execution - the 'correctness' of the part of code associated. This is especially of interest when the program has been altered but is continuously supposed to fulfill the same tasks. The *examples* are, by the system, considered as minimal *contracts* for the associated code.

commentaries

A standard deficiency of LISP is that its READ function ignores commentaries. I have modified the S-expression reader of LISP, so that commentaries are not lost any more for inspection once the program is loaded, in attaching them to the code which they precede.

index

Indexes are attached to function- and variable names. They are constructed automatically during the editing of the functions and may be supplied manually for functions whose implementation is left open.

meanings

The programmer has the possibility to attach special commentaries to his functions, expressing the 'meaning' of it. Those commentaries may be printed out by special commands (they constitute one point of view of the associated function). They also serve other utilities, such as the trace-package, the pretty-printer, and even the evaluator when it encounters unfinished parts of the program.

assertions

To every part of the program may be attached special predicates describing the restrictions which must be valid at entry to this point of the program. During execution, the evaluator automatically checks their satisfaction and interrupts execution if some discrepancies appear.

version

As already mentioned, modifications of the program generate new branches in the program tree: the version attachments. They contain special subfields for the *date* of the modification, its *reason* (commentaries given by the programmer) and - if the program is developed by more than one person - the *person* actually carrying out those modifications.

Two additional specialized attachments are planned: one for *symbolic descriptions*, generated automatically by Daniel Goossens' CAN system [Goossens 79] or by Patrick Greussay's RAINBOW system [Greussay 80], and one for *propositions* of improved versions, generated automatically by my own system PHENARETE [Wertz 82].

The user has the possibility to introduce new attachments and - optionnally - he can make them known to the system in defining special procedures for which the editor, the evaluator and other parts of the PE are automatically checking their presence. Those procedures may be considered analogues to the user defined pretty-print properties of more classical LISP PE's.

3.3 Using the Editor

The editor divides the screen in 3 different zones: one where the edited structure is displayed, one for giving a menu of special commands, and one for the command line. Figure 2 shows the screen at a given moment of interaction displaying a partly constructed function, a menu window showing commands to handle attachments, and in the command line a request to write this partly defined function into a file.

<pre>(DE FOO (N) (IF ; ASSERT: (> N 0) ; (= N 0) 1 <if-false-statements>) <instr>)</pre>	<pre>^A attach assetion ^E attach examples ^R attach reasons</pre>
<pre>^X^W filename: TEST.VLI</pre>	

Fig. 2: organisation of the display

In Figure 3, we have displayed part of a sample editing session. Note that at every interaction with the editor the pattern of the expected instruction is displayed, helping as well the novice programmer with his/her initial syntactical problems (no closing paranthesis has ever to be typed!) as indicating to the experienced programmer continuously at which point s/he is inside the program. The cursor moves automatically to the next syntactic unit as soon as the typing of the previous unit is finished. All attachments appear as commentary when displayed, preceded by special symbols indicating its type.

```
<function><arguments> ; the user typed an opening paranthesis,
                        ; cursor position is displayed by ~
(DE <ftn-name><vars><instrs>) ; the user typed DE
(DE FOO (<var1><vars>)<instrs>) ; user typed FOO and "("
(DE FOO (N <var2><vars>) <instrs>) ; a ^L indicates "exit of level"
(DE FOO (N)<function><arguments> <instrs>) ; user typed "("
(DE FOO (N) ; user typed IF, newline
  (IF <test><if-true><if-false-statements>) ; and indentation is
  <instrs>) ; echoed by the machine
(DE FOO (N) ; user wants to attach
  (IF ; ASSERT:~. ; an assertion and typed
    <test><if-true><if-false-statements>) ; a ^A
  <instrs>)
...
...
...

```

Figure 3: sample session with the editor

When the program gets larger than what can be displayed on the screen, the editor automatically changes to the *star convention*. In this way, there is always assured that the entire program *structure* is displayed, so that the user is never lost as to the point s/he is inside this structure.

There exist naturally more complex commands, some of them are listed below :

write saves the actual state of the program tree by writing it in a file

read retrieves a program tree from a file and places the cursor at the beginning of the program or at the first syntactic unit not already filled out

find moves the cursor to a node which matches a given pattern

attach removes a subtree from the program tree and 'attaches' it to the cursor

detach inserts the previously attached subtree at the position of the cursor

More sophisticated editing commands exist to move the cursor, to display subtrees or attachments, or to 'delete' them. All commands are syntactically checked, so after finishing editing the user is assured to have a syntactically correct program. (This checking corresponds roughly to the *surface reading* of the PHENARETE system [Wertz 82].)

4.0 THE EVALUATOR

In an integrated and interactive PE, such as the one I propose, the evaluator has to be designed in such a way as to be able to take into account all the possibilities offered by the PE. This implies that there have to be special features to handle the *program tree*, since it is no more confronted with a *linear* version of a program, but with *multiple* versions of it, enriched with special descriptors - possibly directing the evaluation process itself. It has to be able to handle all the special cases related to the debugging and development phase, such as conditional evaluation, interruption on *events*, *inverse* evaluation and evaluation of unfinished code. In this paragraph I will describe the differences between normal and PE dependant evaluators.

4.1 Internal Organisation of the Evaluator

The most important characteristic of our evaluator is that I have generalized the notion of *context*: I distinguish between syntactic context, the context determining which one of the multiple versions of the program is to be executed, and dynamic context determining the *state* of evaluation, mainly determined by the stack frames.

4.1.1 syntactic contexts

Since the evaluator is working on *program trees* rather than on linear programs, a call to the evaluator has to determine which path in the tree corresponds to the version actually needed. This may be defined globally, in setting the version number - which by default will be the latest, i.e. the highest one - or dynamically during internal calls with as additional argument the version to consider.

I also intend to include special descriptors which would - depending on the dynamic context - determine the syntactic one. This will be especially useful if the user intends to use different versions of his/her program during different execution modes (trace, step-by-step, etc) and implements an elegant way for *table driven* program development, analogues to *table driven* programming.

4.1.2 dynamic contexts

The dynamic contexts determine the *mode* of evaluation at any given moment. These modes may be determined explicitly, for example by invoking the *stepper*, trace package or undo-facility, or implicitly, for example: when the evaluator encounters a call to an *undefined* function, the *normal*

mode of evaluation is suspended and a special *careful* mode is entered.

Besides the user programmed modes (a facility already necessary for incrementally extending the system itself) I actually distinguish the following *modes* :

normal mode : the normal evaluation of a given version of a program,

trace mode : this mode, as its name indicates, is on when tracing the execution of a function. It determines the *grain size* - indicating at which level of detail, or for which predetermined events the trace has to be carried out.

stepping mode : as the trace mode, this mode determines the *grain size* of the steps. After each step it enters a break point during which the programmer has access to the entire system.

event mode : in event mode, the evaluator is looking for special computational events to happen, such as the assignment of a special value to a variable, the call of a function with some special arguments, or any other predetermined computational event. If such an event occurs, the evaluator interrupts normal execution and either enters a break point or evaluates a user given function associated to this event. Any number of events may coexist.

careful mode : when the evaluator has to compute the value of a function call to an until now undefined function, or the value of an unbound variable, normal execution is suspended, the arguments of the function call and the name of the function or the name of the variable and the available index information are displayed, and the user is asked how to proceed. He can supply manually some value for the variable or define the missing function, and then proceed in the normal evaluation, or he can tell the machine to switch to symbolic mode, or he can enter debugging mode.

symbolic mode : whenever the evaluator is in this mode, instead of computing values it computes symbolic expressions describing (reflecting) the computation. I intend, in a further development of the system, to hand the symbolic evaluation over to Goossens' system CAN [Goossens 79].

inverse mode : inverse mode corresponds, basically to the 'classical' undo facility of Interlisp [Teitelman 78]. Two possibilities exist : either one undoes in a step by step manner - in unwinding the control stack - or one undoes until the occurrence of a special event.

Naturally, the different modes may be combined in different ways. So, the user can, for example, tracing the 'undo'-ing until a given event, combining inverse, trace and event mode, but it is still an open problem to determine which combinations are possible and which are not. Also, I am looking for possibilities to make the evaluator switch autonomously between *real* and *symbolic* execution whenever necessary.

The guiding principle of the evaluator is to continue evaluation as long as possible, without ever punishing the programmer with any kind of 'fatal' errors.

There exist an additional mode : *debugging mode*. In this mode, listed separately since independent of the evaluator, the display screen is divided in several regions : one for displaying the run-time stack, one for editing functions and one for displaying variable values. The user can, in this mode, travel through the stack, visualising the functions pointed at whenever s/he wants (and edit them), and re-enter evaluation at any point he chooses.

In some cases, such as for example when the user has crucially modified a function of which a call is somewhere below on the stack, the normal restarting of the evaluation can not be guaranteed to be error free.

4.2 Automatic Checking

As already mentioned in paragraph 3.2, the user has the possibility to attach examples and logical input assertions to every part of code. The examples are input/output pairs which the associated code should be able to handle correctly. They are considered as minimal *contracts* of this part of the program. Whenever the evaluator encounters such an example attachment not already verified, it suspends normal execution and executes the portion of code on the input examples given. Then it compares (with the EQUAL function) the computed output for those examples with the output supplied, continues - after marking the example as verified - normal execution if all the input/output pairs are satisfied, otherwise it enters a breakpoint, informs the user of the discrepancies encountered and waits for further instructions.

Naturally, this kind of checking can not be considered as program verification, but it gives a minimal security as to the correctness of the associated code : it ensures the user that the program works *at least* for the examples supplied.

At every entry to the evaluator, it checks also for the presence of an *assertion* attachment, indicating the class of values which are accepted as possible input by this part of code. In Figure 2, the assertion indicates for example that the variable N should have a value greater than zero. If such an attachment is present, the evaluator applies the given predicate (which may be of arbitrary complexity) to the input values and proceeds evaluation if the predicate yields true, otherwise it interrupts evaluation at the point this piece of code (or function) was called and enters in *careful* mode.

Let us remark that nothing excludes - a priori - the combination of a sophisticated program verifier to the PE. To this end it would be sufficient to define a new class of attachments containing the logical descriptions of the program's behaviour and to associate to those attachments a special *event* mode invoking the verifier. What I am describing here in this paper is not a final version of a PE, but the basic PE on top of which much more sophisticated and personalized user aids may be incrementally built on. It is the basis for an incrementally growing PE.

5.0 MAINTAINING CONSISTENCY OF DESCRIPTORS

Given that, in our PE, a program is a complex tree combining code and descriptions, and that I want the system to automatically check its correctness, one of the main problems emerging is to keep those descriptors consistent with the code, without overcharging the programmer, during every modification of some aspect of the program tree, with an enormous burdon of checking its consistency manually. The ideal would be that the system automatically maintains the consistency, but this implies the construction of a program understanding system completely beyond the actual scope.

The solution adapted so far is :

1. to propagate automatically every modification
2. to try to automatically construct new formal descriptors which could be compared to the already existing ones.

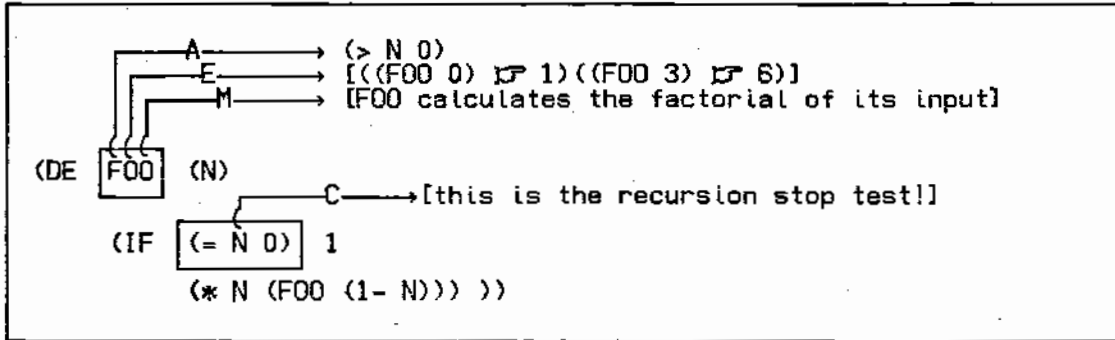
5.1 Propagation of Modifications

Let me repeat : in my representation schema, every part of a program may be represented from a multitude of different viewpoints : code, meaning, commentary, examples, symbolic descriptions, etc. To be of any *practical* use, a minimal consistency between different viewpoints of the same object must be assured.

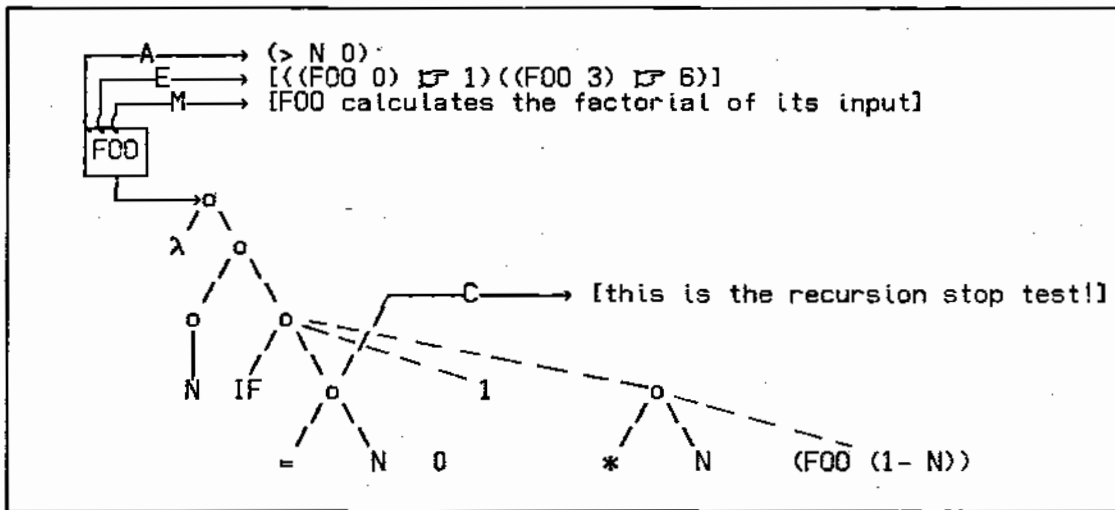
I take as granted - and it is left to the responsibility of the programmer - that the initial version *is* in itself consistent (with the exception of the *example*- and *assertion*-attachments, which are checked by the evaluator). Whenever the user modifies part of his code or its description however, this consistency can not be assured any more. What the system does in such cases, is to keep the previous version as it is, but to travel backand forth in the program tree, associating to all dependant parts a conditional attachment referring to the modified code. This propagation technique permits to indicate during future displays of those parts, the fact that some parts have been altered and that the correctness of the old attachments can't be assured.

Special editor commands exist to delete those conditional attachments, or to replace them by new ones.

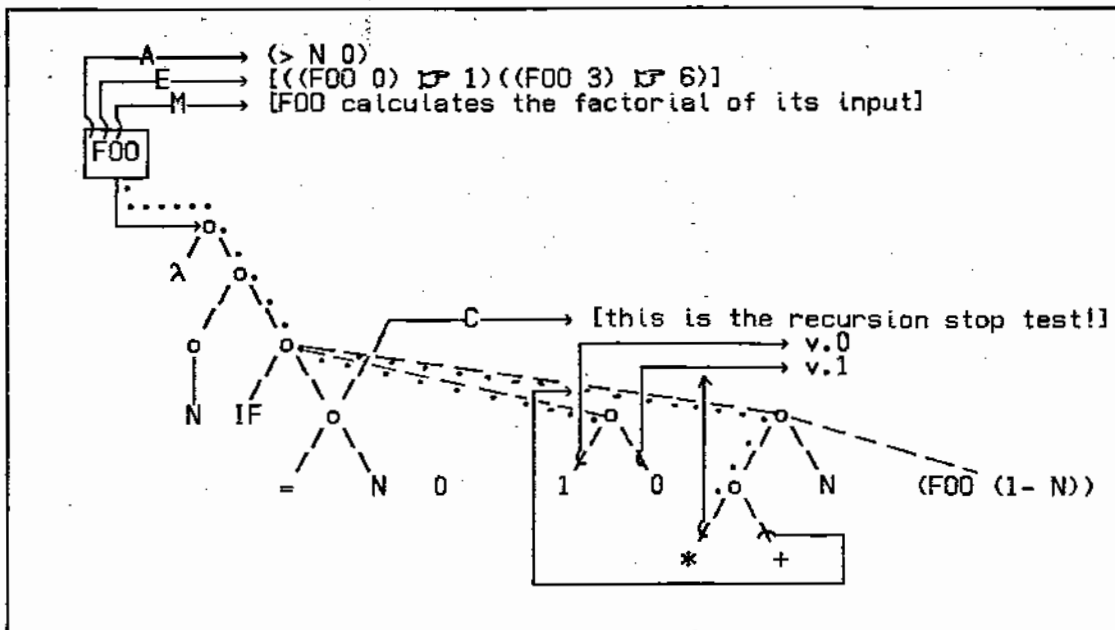
In order to be more concrete, let me go back to the example of paragraph 3.1, the function FOO, and let us suppose the initial version having some descriptors attached, such that we have the following state :



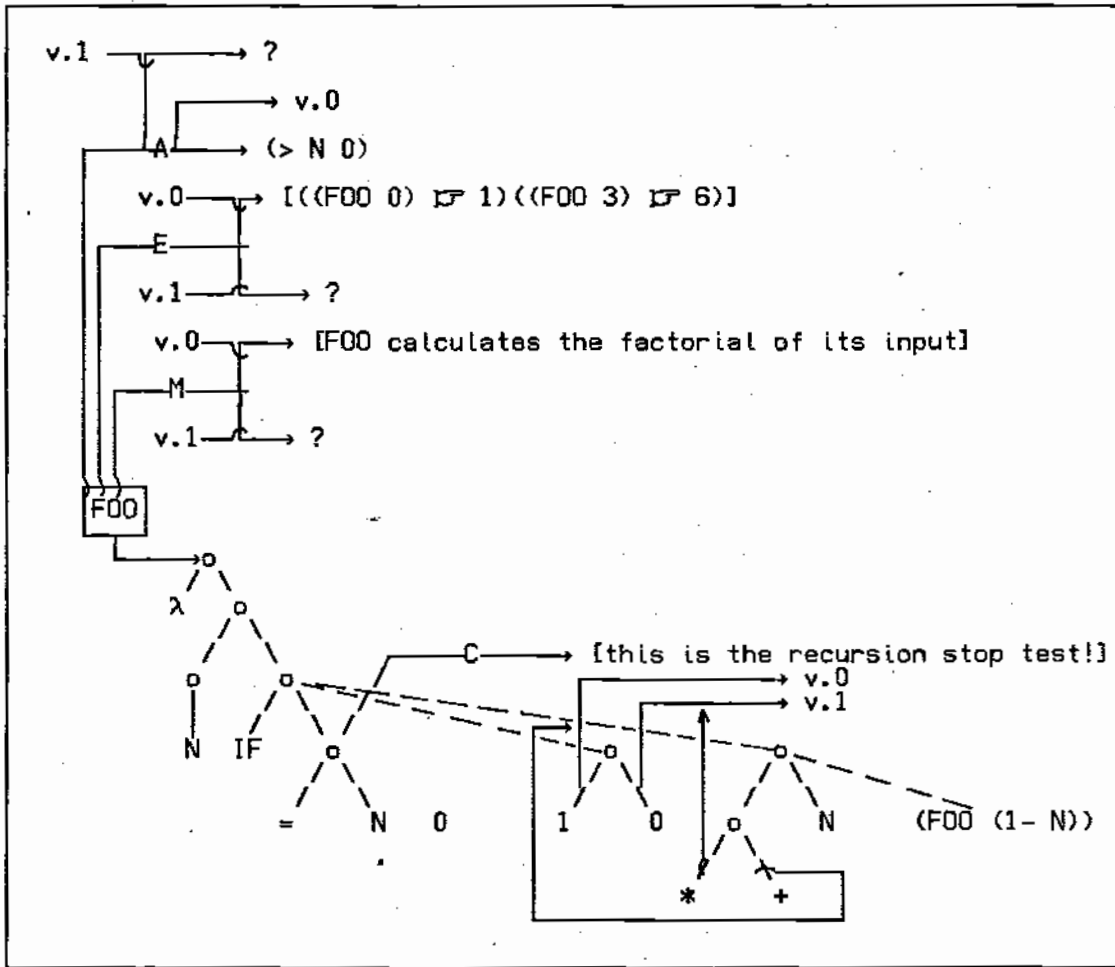
or, if we represent it in a tree like structure (simplified for the purpose here) :



After the modification of paragraph 3.1, we have initially :



Travelling 'back and forth' in the program tree is displayed here by the dotted line. Only those attachments are marked conditionally which are encountered on this route. So, we find after the propagation of the modifications the following modified tree :



In this example only endpoints of the tree were modified, so the only updating necessary was upwards in the tree. The reader shouldn't have any problem imagining a modification of a non-terminal node which would also imply some downward updating.

The ?-filler may be replaced afterwards with new specifications or, with a simple editor command, deleted. In this latter case the entire restriction on version zero or version one (in our example) is dropped, and the attachment is once more valid for the entire subtree.

Note that, if the function has an index attachment, the updating is continued in the calling functions. There exists an editing command permitting to *trave* through the newly created conditional attachments, permitting to get an immediate impression of the *range* of influence of the

modification, and permitting to correct - if necessary - the influenced parts. Needless to say, that the last modification, during an editing session, can be un-done : the tree takes back its previous structure if this is the case.

5.2 Automatic Construction of Descriptors

The initial design of our PE has a limited possibility of automatically constructing descriptors (attachments).

During the editing of a function an index-program is automatically invoked, attaching to the actual version a description of what functions are called (specifying the actually valid version of them), and indicating which variables in it are local, which global (bound and unbound). It equally updates the index attachments of the currently valid version of the called function, indicating there the complementary information, i.e. that they are called by this version of the function. Also, the property lists of all the variables used inside the function are updated with corresponding information.

As previously mentioned, in the paragraph on the evaluator, those informations may be used by it for the symbolic evaluation. Nevertheless, it is mainly intended for the interactive reading of the program and for future debugging of called functions : indications are given to the user to check that the modifications are consistent with the different calling sequences.

In a planned further development of the PE, constant use of the CAN [Goossens 79] and PHENARETE system [Wertz 82] are planned : each of this systems continuously constructing new attachments for every edited version. CAN will construct symbolic contracts, while PHENARETE will construct alternative versions of the edited functions. For a precise description of the capacities of CAN and PHENARETE the reader is referred to [Goossens 81] and [Wertz 79].

6.0 CONCLUSIONS

In this paper a programming environment being designed and implemented by the author in VLISP [Chailloux 80, Greussay 78] has been presented. The first version will be experimental; only intense use of it will ultimately expose its strengths and weaknesses and determine the next version.

The proposed system gives programming support to a single programmer or a group of programmers working on a single, non partitioned program. It is intended for use by experienced AI-programmers as well as by novice programmers [cf my work with naive programmers [Wertz 81]] and should, for this reason, combine high level support, such as automatic updating of descriptions and consistency checking, with low level support, such as automatic syntax checking during editing. It is centered around an interactive, structure oriented, language dependant editor, with special possibilities for handling multiple viewpoints of programs as well as supporting the simultaneous existence of multiple versions of a program.

The design is made to permit a maximum of flexibility and of ease of extension : every standard feature of the system can be adjoined by user defined similar features.

The proposed PE intends to integrate - until now dispersed - subsystems in a unified system of interacting modules. Much of its possibilities are invoked automatically, without the user even being aware of it, such as the fact that modifications on one level of descriptions of the program are automatically propagated to all the other dependant levels.

The programs are - internally - represented as program trees, possibly containing multiple versions and multiple viewpoints of it. During interaction with the system, the user's cognitive capacities are never overloaded by too large an amount of information, and only one version or one viewpoint is visible at any moment of the interaction.

It is planned, ultimately, to give the user the possibility to do *all* the possible interactions - execution, editing, tracing, etc - on any level whatsoever, with automatic references (whenever necessary) to the other levels.

Further development includes the adaptation of the system towards sophisticated display technics, so that it would be possible to display in different windows, different aspects of the same program, and to show, during execution, in different active windows the progress of the computation.

Much work remains to be done to make the system independant of a specific language, such as LISP. To this end, I think the internal representation of the program trees can be kept, but ways have to be found to include, incrementally, compiled code (for compiled languages such as Pascal or Ada) and to maintain a strict correspondance between it and the tree like representation of it. A special facility to 'generate' editors for a given language should also be included.

To increase the efficient use of the system, the notion of minimally necessary information should be further developed.

At the time of this writing, only the preliminary implementations are finished: I have re-written most of the LISP system to handle the program trees, a prototype version of the editor and a limited version of the evaluator are running. Practical experience in the use of the PE for the construction of larger programs will be gained over the next months, as it is intended as a tool for the further development of the PE itself.

Acknowledgements: Support for this work is provided in part by the Agence pour le Developpement de l'Informatique, contract no. ADI 81/331, and in part by the Centre National de la Recherche Scientifique, contract no. ATP 4097.

7.0 REFERENCES

[ADDA 81]

M. ADDA, *Un Editeur Spécialisé*, mémoire de DEA, Université Paris 6, mai 1981

[CHAILLOUX 80]

J. CHAILLOUX, *Le Modèle VLISP : Description, Implémentation et Evaluation*, thèse de 3ème cycle, Rapport de recherche LITP 80-20, April 80

[GREUSSAY 78]

P. GREUSSAY, *Contribution à la définition interprétative et à l'implémentation des lambda-langages*, Thèse d'Etat. Rapport de recherche LITP 78-2, Feb. 1978.

[GREUSSAY 78]

P. GREUSSAY, *Le Système VLISP 16*, Ecole Polytechnique, Dec. 1978

[GREUSSAY 80]

P. GREUSSAY, *Program Understanding by Reduction Sets*, A.I.S.B.-80 Conference 1980, Amsterdam, July 1980, G-1 G-7.

[GOOSSENS 79]

D. GOOSSENS, *Meta-interpretation of recursive list-processing programs*, Proc. 6th IJCAI, Tokyo, Aug. 1979, vol. 2 pp. s7-s12

[GOOSSENS 81]

D. GOOSSENS, *La Méta-évaluation au Service de la Compréhension Automatique de Programmes*, thèse de 3ème cycle, Rapport de recherche LITP 80-52, Jan. 1981

[HAINES 65]

E.C. HAINES, *The TREET List Processing Language*, MITRE Corp., Information System Language Studies Number Eight SR-133, Bedford, Apr. 1965

[SHNEIDERMAN 80]

B. SHNEIDERMAN, *Software Psychology*, Winthrop Publishers, Cambridge Ma, 1980

[STEELE 79]

G.L. STEELE, *The Dream of a Lifetime: A Lazy Scoping Mechanism*, M.I.T., Artificial Intelligence Laboratory, Memo 527, Nov. 1979

[TEITELMAN 78]

W. TEITELMAN, *Interlisp Reference Manual*, Xerox Palo Alto Research Center, Oct. 1978

[WEINBERG 71]

G.M. WEINBERG, *The Psychology of Computer Programming*, Van Nostrand Reinhold, New York, 1971

[WERTZ 79]

H. WERTZ, *A System to Improve Incorrect Programs*, Proc. 4th International Conference on Software Engineering, Munich, R.F.A., pp 286-293, sept. 1979

[WERTZ 81]

H. WERTZ, *Some Ideas on the Educational Use of Computers*, Proc. Annual Conference of the ACM, ACM'81, Los Angeles, Nov. 1981, pp 101-107

[WERTZ 82]

H. WERTZ, *Stereotyped Program Debugging : an Aid for Novice Programmers*, International Journal for Man-Machine Studies, (1982) 16, Academic Press