DVLISP: AN EXAMPLE
OF A PROGRAMMING ENVIRONMENT

Harald WERTZ

Département Informatique
Université Paris VIII (Vincennes)
2 rue de la Liberté
93526 ST-DENIS Cedex 02

C.N.R.S.
Lab. 248 LTP
2 Place Jussieu
75005 Paris

1. INTRODUCTION

Most existing programming environments (e.g. UNIX, INTERLISP, or Apple Macintosh) are composed of a multitude of quasi-independent, specialized tools, such as editors, help facilities, documentation aids, language interpreters or compilers, linkers and loaders etc. This quasi-independence of the tools, which may be very helpful during their development, renders their cross-fertilization very difficult. By cross-fertilization we understand the possibility of one program using the knowledge of another, as, for example, the symbolic debugging facilities for compiled languages which generally use the tables constructed by the compiler.

We need a programming environment where the possibility of cross-fertilization is generalized so that each part of the environment may use knowledge of each of its other parts. In the DVLISP programming environment presented in this paper, we achieve this through a unique representation of the evolving programs. All the different tools work on this shared unique representation while adding their knowledge to this same representation.

Developing programs is a very complex and time consuming process, which rarely proceeds in a well-defined way, from the abstract specification to design and finally implementation. It is often based on some method of trial and error or successive refinement, while constructing, testing, then rejecting or modifying different algorithms,
heuristics or representations. Often the initial specification of parts of the system will be modified as a result of new insights gained during the process of interactive design and implementation.

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

The BMLISP programming environment addresses all these issues through integration of sophisticated support tools for

- documenting
- editing
- annotating
- verifying
- observing
- reading
- maintaining

programs. Each of these tools works on a unique tree-like representation of the program. This representation includes the different versions of the program, comments, assertions and examples. Naturally, the system permits the integration of special user-built tools.

The BMLISP system presented in this paper, is intended for use by novice as well as by expert programmers. It is a result of continuous work on the VLISP system [Gries77, Chailioux80], is written in the C language and in BMLISP, occupies approximately 100K and is running under all Berkeley UNIX systems.

2. METHODOLOGICAL ASPECTS OF INTERACTIVE PROGRAMMING

Our system is based on interactive programming. Let us briefly define our understanding of this notion.

Interactive programming implies that all the program development processes, from the design phase to the implementation phase, is done directly on the computer and in continuous interaction (communication) with the machine.

This interactive development of programs engenders deep changes in understanding the programming activity.
a. Programming is increasingly perceived as a conjunction of communication and programming. The construction of programs is done in collaboration with the machine, which points to inconsistencies (and errors) and permits their correction (possibly during the execution of the - as yet unfinished - program). The machine gives immediate answers to questions concerning the execution of the program and information about the system's capacities or organization. The documentation facilities of bWISP handle these needs of communication.

b. The conception of programs is decreasingly the design of new programs, but increasingly a process of successive modifications of already existing programs. Given that the programming activity is done in continuous communication with the machine, and given the possibility to experimentally test the already developed (yet possibly unfinished) parts of the intended program, there exists a real opportunity to modify similar programs, and consequently to test them incrementally, in order to measure their differences. The construction of programs through the modification of other programs is at the heart of the notion of interactive programming. It considerably speeds up their development: as with mathematical theories which are constructed on top of other (previously demonstrated) theories, it allows the construction of programs on top of other programs, which previously have (at least experimentally) shown their capacities: this is an evolutionary style of programming.

Through it, the construction of programs transforms into a task of analyzing, understanding, modifying and combining programs.

3. bWISP: A PROGRAMMING ASSISTANT

Large programs present large problems: they are difficult to write, to correct, to test and to modify. In analysing the problems related to large programs, Terry Winograd [Winograd64] introduced the notion of a complexity barrier, and proposed overcoming it with systems capable of automatically understanding programs and acting as assistants to the programmer.

Our bWISP system has been constructed with this goal in view. It has been designed to take over the programmer's task of memorizing

1. This understanding of program construction as a series of successive refinements of preexisting programs has been partially formalized and automated in Dershowitz's system [Dershowitz80].

2. A similar program is an existing program which partially satisfies the specification of the program to be constructed.
implementation details, to actively support him/her in searching and correcting errors and to offer him/her a set of evaluation tools to help understand the functioning of the program (constructed by him/herself or by other programmers).

**BWLISP** does not only offer tools for constructing correct programs, but it also offers tools for observing the execution of possibly incorrect programs, thus helping in the construction of efficient programs.

We understand the complexity barrier as a limitation of our capacities of memorization. While the program is growing, the programmer incrementally constructs a mental image of the interaction between its different parts, of the use and usefulness of its variables and of its control- and data-flow. This image, which may be very detailed for small programs, has, with the developing program, to be transformed into an increasingly abstract image. Only the global aspects of the interactions and the relationships between implementation and (formal or informal) specification are kept while more and more of the local implementation details are eliminated. This modification of the programmer's internal representation of the program is due to inherent human cognitive limitations [Diller86].

In order to circumvent these human limitations, **BWLISP** offers various documentation modules: first, a documentation module concerning the system itself. This module interactively gives information concerning the concepts, commands and instructions of **BWLISP**. Second, it possesses a local documentation module: **BWLISP** incrementally builds up descriptions of the user-defined structures. Similar to the Masterscope system [Teitelman78], a module analyzes all the functions during their definition (construction) and constructs a database of their interactions. The user can, at every moment during his/her interaction with the system, ask questions about these objects. The questions may concern the places where a given function or variable is used, the variables or functions which are used at a given place or the reasons for a given object or a given computational event. Finally, **BWLISP** has a global documentation module: this is a module which displays a control- or data-flow tree permitting the user to explore the implementation along the branches of a tree and to follow visually the execution of the program while the system is dynamically highlighting the active nodes and branches of this tree.

As the developing program grows larger, the user depends increasingly on those documentation aids. Their use has been a continuous help during the development of **BWLISP** itself.

Naturally, those documentation tools are also very useful for reading a program: together with **BWLISP**'s annotation facility, presented later, these tools permit an extremely convenient way of interactively exploring a program.
All programs contain errors. The utility of a programming environment, or of an automated assistant, depends strongly on the capacities of its error handling routines.

In **BVLISP**, each time an error occurs the user can choose among one of the following options:

- s/he can correct the program 'on the fly'. This possibility is particularly useful in the case of minor errors, such as typing errors or references to undefined (not yet defined) objects.

- s/he can enter the editor which will automatically point to the origin of the error. Let us note, that this place is not necessarily identical with the place in the program where the error has been detected. **BVLISP** has the capacity to trace the development of a program and, through a network of dependencies, find the last modification(s) which may have caused the error.

- s/he can ask the **PHENARETE** subsystem to automatically correct the erroneous program fragment.

**PHENARETE** ([Werts82, Werts85]) is a system to automatically correct and improve LISP programs. This system is composed of a database expressing knowledge about standard LISP functions, stereotypical programming knowledge and a meta-evaluation algorithm. While reading a program, **PHENARETE** (as part of **BVLISP**), constructs an internal representation of the program which must satisfy the constraints expressed in the knowledge base. If **BVLISP** encounters an error, **PHENARETE** metaleverages the program using and enlarging this knowledge base. If at any moment during this process, one or several constraints are not satisfied, the system applies a set of correction rules until it reaches a transformed version of the program which satisfies all of the constraints. When none of the constraints is violated, **PHENARETE** considers the program as correct.

- s/he can enter an inspection loop. Within this loop, the programmer has access to the entire execution environment and to the entire **BVLISP** system. s/he can interactively change the value of variables, undo effects of previous computations, examine the history of the calculation, edit functions or data structures, etc. When exiting from one loop, the computation continues in the (possibly modified) environment.

- one may, of course, abort the computation at any point.

Concerning the preventive search of errors, **BVLISP** has a capacity to accompany LISP functions by a set of *input/output* examples, whose correction is automatically checked after each modification of the
corresponding functions. These examples are informal function specifications; the associated verification mechanism assures the programmer that the modified functions still satisfy the specifications. This liberates the programmer from manual verification after each modification and automatically informs him about each difference between the specification and the result of an execution of the corresponding example function call. Another view of these examples is to consider them as notice comments: comments not only for the passive human reader of the program, but comments useful also for the interactive exploration of the program.

Another important feature of BVLISP is the possibility to attach input and/or output-assertions to LISP functions. These assertions are tested at each call and/or each exit of the related function. Here again, the user is informed of each non satisfied assertion. Naturally, those assertions are (without a theorem prover) not assertions for general program verification, but attributes verifiable for each execution.

We have generalized this mechanism in two directions: on the one hand we permit arbitrary supplementary actions at the entry and the exit of user defined functions, and, on the other hand we have adjoined mechanisms to permit those same possibilities at any arbitrary point in the program and at the evaluation of any object. With these mechanisms, the user can not only associate input or output assertions to every expression inside every program, but s/he can associate any arbitrary additional activity, allowing the construction (and integration) of a set of modules unique to BVLISP, such as automatic version maintenance of the program, annotation of a program by descriptions on various conceptual levels, monitoring of variables, path-expressions, etc.

BVLISP is superior to other programming environments not only in its original tools (such as execution tracing on data- or control-flow trees, or this general annotation facility), but most importantly in the integration of these tools in a unique framework which permits an exchange of data between the different tools and allows access to all the tools (and related data) during run-time of the program.

This general annotation facility (and the corresponding integration of the supplied tools) results from two fundamental changes in our LISP.

3. The idea of introducing assertions into programs is surprisingly old: Goldstine and Von Neumann [Goldstine68] had already proposed such instructions in conjunction with assignments. Nowadays, assertions are mainly used in verifying compilers [Floyd87] or in general program verification systems [Igarashi75], but not at all for "on the fly" verifications. Let us note that, nevertheless, some languages support instructions similar to our assertions: especially the assertion ON of PL/1 [IBM87] and the ASSERT instruction of PL/C [Zeikowski71] and of ALGOL W [Skest71].
interpreter:

- a new representation of lists permitting, whenever necessary, to attach additional information to its cells
- a new evaluator which not only computes the value of its argument, but, whenever necessary, prefixes this computation by the evaluation of the entry-activities attached to this argument and postfixes the computation by the evaluation of the exit-activities attached to the argument.

In order to illustrate some of the characteristics of the system, let us examine a scenario of a typical interaction with BVLISP. In this scenario, we will concentrate on local features, i.e.: primarily concerning individual functions. Whenever appropriate, we will indicate the corresponding global (i.e.: concerning a set of functions and data-structures) features.

4. A COMMENTED SCENARIO

```lisp
(menu, 'save, 'save, 'd delete droite, 'd delete gauche, 'w quote
BVLISP UNIX Portail pour vous servir
pour avoir des renseignements, tapez '?'
Signalez tous les bugs à [hw]
? (de dela (el liste)(cons?
? (null late))
? (e
l'objet: ? eq
eq (eq e1 e2)
teste si l'expression e1 pointe sur le meme lieu en memoire
tente si e1 et e2 sont des nombres et leur egalite.
Remene "1" si oui, "nil" si non.
Peut egalement s'ecrir (= e1 e2).
```

We see above a picture of the screen at the beginning of a session with BVLISP. The first line displays part of the line-editor menu, the default menu. Just underneath is the initial system information (in French, since this system is used in a French environment, although there would be no problem giving these texts in another language). Whenever BVLISP displays its prompt-character ("?"), it is waiting for commands by the user. Here, the user began to define the new function deIe. The line indentations, after each <return>, are, as in all LISP systems, characters typed by the user are written in boldface, system responses in normal characters and the cursor is represented by an "\".

Page 7
automatically generated by the LISP reader. This indentation gives a first, visual control of the matching parentheses.

In the middle of the third line of this new function definition, the programmer typed the '?' character. An immediate temporary interruption of the standard interaction results, and the message

'Object ?'

is displayed in a new window, followed by the activation of a documentation module, which gives a description of the given object. Here the user asked the standard information about the eq function. After having given the documentation (here: the manual entry for this function), typing any character will erase this pop-up window and permit the previously interrupted activity to continue.

Characters having an immediate effect are called short characters. Naturally, the user can define (or redefine) his/her own such characters. When using a mouse, these characters correspond to mouse clicks in specialized menus.

```
; 'w' menu, 'f' source, 'h' recue, 'd' delete droite, 'g' delete gauche, 'q' quote
? ((eq c (car liste)))
? (t (cons (car c)(delete ol (cdr liste))))
? (c
? ?
? (delete 'c(a b c))
? (undef undef)

\text{l'objet: ? delete}
\text{----- delete-----}
\text{type = EXPR}
\text{local variables = c liste}
\text{used by = delete}
\text{used by = delete}
```

The user has now finished writing his delete function. In parallel to the reading of the user input, BLYSP analyses it in order to build up an internal description. This internal description of user objects may also be obtained with the '?' command. Here, directly after defining the function, the description knows about the type of the function, the local and global variables used in it, the called functions as well as the calling functions. This information may be very useful for the interactive reading of programs written by other persons, as well as for the incremental development of one's own programs. At any moment this information about the dependencies of functions and data may be retrieved by the user. The information displayed here is the kernel of the local documentation used by modules for global information.
The test call of the `dele` function visibly computes an unintended result.

```
(+w+ menu, -r avance, -h recole, -d delete droite, -h delete gauche, -w quote
  ? (dele 'a (b a c))
  = (undef undef)
  ? (trace dele)
  = (dele)
  ? (dele 'a (b a c))
     --> (dele a (b a c))
     --> (dele a (a c))
     --> (dele a (c))
     <-> (dele nil)
     <-> dele nil
     <-> dele (undef)
     <-> dele (undef undef)
  = (undefined undef)
  ?*
```

Not knowing the reasons for this erroneous behavior, the user asked for an execution trace of the `dele` function. Note that the trace, as all other additional activities, never changes the standard behavior of the traced function: one can see in the picture, the recursive parts of the function remain recursive, the iterative parts remain iterative. This characteristic is unique to `BUIJSP` and is due to the implementation of the trace as a pre- and post-activity of function calls (this kind of tool is normally implemented as an envelope of the function definition). Here, this characteristic is very useful: it shows graphically that the algorithm of the `dele` function is correct, and that the error is created somewhere in extracting the values of the different elements of the resulting list.

Let us edit the function:

```
+ MENUS: - menu: this menu    - menu: time menus
  * - menu: edit menus  - menu: global variables
  + - menu: version menus

  <-> dele nil
  <-> dele (undefined)
  <-> dele (undefined)
  = (undefined undefined)
  ? (edit dele)
  Welcome to wonderful edit
  Editing old function, dele
  The structure has been saved as Version: 0
  Future work is on Version: 1
  = t
  ?*
```
The standard program editor is activated with the `edit` command, for editing a function, or with the `editl` command for editing a data structure. Initially, the editor shows what new, editing related, menus are available. `edit` is a structural [Donzeau-Gouge70, Donzeau-Gouge84], screen oriented, editor: the upper part of the screen is reserved for displaying the edited structure, the lower part of the screen is the standard lisp interaction window.

The editor, after having given its 'hello', informs us that it is editing an already existing version of the program (a version which hasn't been constructed using the editor) and that it names this initial version version 0. All future modifications will change exclusively the new version f of the function. This means that the editor is automatically in charge of maintaining the different versions of the edited structures, which assures the user that:

- s/he can undo all modifications
- s/he can, at any moment, go back to any previous version.

The editor constructs a tree-like representation of the program, which includes the entire history of its development. It is this tree-like representation which is also used by the evaluator, allowing the user to dynamically decide which version to evaluate.

<table>
<thead>
<tr>
<th>Structure being edited: de</th>
<th>Version: 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>((c1 list) (cond</td>
<td></td>
</tr>
<tr>
<td>((null list) nil)</td>
<td></td>
</tr>
<tr>
<td>((eq el (car list)) (de</td>
<td></td>
</tr>
<tr>
<td>(de el (cdr list)) ))</td>
<td></td>
</tr>
<tr>
<td>(f (cons (car x el) (de</td>
<td></td>
</tr>
<tr>
<td>(el (cdr list)))))</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Editing old function, de</th>
<th>The structure has been saved as Version: 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>future work is on Version: 1</td>
<td></td>
</tr>
<tr>
<td>= t</td>
<td></td>
</tr>
<tr>
<td>? (dp2l)</td>
<td></td>
</tr>
<tr>
<td>= t</td>
<td></td>
</tr>
<tr>
<td>? (mv r d r 3 d 3 r)</td>
<td></td>
</tr>
<tr>
<td>= t</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td></td>
</tr>
</tbody>
</table>

Here we present the normal editing window: the first line indicates which version of which function is actually edited, the rest of the window displays the edited structure with the cursor position indicated by an arrow ('>'). One can see that the fact that the `de` function is still traced doesn't show up in the internal representation of the function: tracing, as all other additional activities, never changes the code of the program. In this way, the user is never confronted with objects whose meaning or origin s/he doesn't know.
The editing command initializing the editing window is the `dpi` (an acronym for display) command. The following command,

```
(mv $r d r 3 d 3 r)
```

is one way of moving the editing cursor. The argument `$r` indicates a cursor motion to the right, the argument `d` indicates a cursor motion inside (down into) a box. The numerical arguments modify the previous argument, indicating the number of its repetitions. Given that the cursor is initially at the head of the edited list, its present position is the result of this cursor motion command. Naturally, if we had terminal equipment permitting analogical movements of the cursor (e.g. a light pen or a mouse) the cursor motion would be done with the help of this peripheral equipment.

In the next picture, the user has replaced the element at pointed at by the cursor, by the atom `list`. It is one of the general characteristics of the `bWISP` system that, independent of the activity one chooses has access to the entire environment and to all of the capacities of the interpreter. In `bWISP` there are no particular, different modes (e.g. an editing mode, a normal interaction mode, etc).

Given that the function has still an active trace feature, its next call is traced. The result seems correct. The last command exits the `date` function from trace mode.

---

5. Normally, the program development process is done in clearly separated steps: edition, compilation, execution. Each of these steps uses its particular environment and its particular language, distinct from the others. No information transfer from one step to another is possible. This strict separation of the different modules necessary for the program construction, and the impossibility of integrating the related activities, obliges the programmer to constantly switch among different representations, different languages and different modes.

In `bWISP`, the programmer is not confronted with this diversity of languages and environments. The information of the different modules may be used by all the others, and the programmer is constantly situated inside one unique language with permanent access to all the other facilities of the system.
DeLiSIP offers a possibility to attach a set of couples of the form <calling-example, desired-result> to a function. These examples are verified after each modification of the function. We understand these examples as informal specifications of the desired activity of the function. Their automatic verification after each modification helps the programmer in detecting new anticipated effects of his/her modification. This mechanism is of considerable use during the development of large programs, where one frequently modifies an old function in order to adapt it to new needs, without recalling the previous constraints. If this is the case, and if the constraints are expressed in the examples accompanying the function, DeLiSIP automatically recalls them to the programmer.
Structure being edited: dele
((el lists) cons
  ((null lists) null)
  (= (e plus (car lists)) (dele el (cdr lists))))
     (t (free (car lists) dele el (cdr lists))))

? (example 'dele 't)
? ((dele 't '(1 2 3 2 1)) (2 3 3 2))
? ((dele 'a b) '((a b) b (a b) (a b)))))
= dele
? (dele 'a '(a b c a))
incorrect example : (dele 'a b) '((a b) b (a b))
the calculated result is : (a b) b (a b)
the desired result was : (a b)
= (b c)
? (far-eq eqa)
= 1
? (dele 'a '(a b c))
= (b c)
?*
* undefined function: `eqa with (el (car lists)) in dele
to get (if possible) a proposition for improvement, type 'p'
to edit dele, type 'q'
to enter an inspection loop, type 't' : t
* do you want to give a value or a function?
type 'v' for value, 'f' for function (this modifies)
to get back to toplevel, type 'c'; (else type 'x')
now, please give the function : equal

Let us explain:

* In the call of the function example we associate two input/output
elements to the dele function.

* During the execution of this function dele, B, we, perceiving that
the function does not satisfy one of the examples, gives this information
to the user.

Naturally, this difference is due to the use of the eq function instead
of the equal function. Note also, that we have not exited from the
editor. To improve the function, it is sufficient to continue editing it.

The editor keeps sufficient information for allowing reconstruction of
any previous version of a structure. This implies that we can, without
any difficulty, go back to previous versions for continuing their
development. This mechanism is sufficiently general and efficient to
permit a very exploratory style of programming, where the program-
mer can easily make test modifications and then go back to a previous
version if necessary in order to continue the development of the
previously abandoned versions.

But let us continue our comment: the command sr searches the first
occurrence of its first argument in order to replace it with its second argument. Unfortunately a typing error has been introduced: instead of replacing eq by equal, we replaced it by equa.

- During the execution of the call (dele 'a (a b a c)), the evaluator realizes this error, informs us and gives us several choices: (a) entering an inspection loop, (b) editing the dele function, (c) asking to correct the error automatically, (d) physically modifying the call, (e) giving the name of another function which would replace the erroneous equa function call, (f) supplying a value which would be considered the value of the call to this nonexistent equa function.

The user has access to all of the above listed options for any errors appearing during evaluation. Note also, that these error handling facilities are not only useful when 'normal' errors occur, but also when 'intentional' errors occur, such as the execution of an unfinished program fragment, where the user can dynamically supply values for the still missing (or undeveloped) parts.

In this example, we choose option 'd', i.e.: to correct the error 'on the fly'. WILSP asks for a function name to replace the function equa, and after having been supplied with equal, the evaluation continues using the newly modified function.

- Finally, the machine delivers a correct result. Note that we didn't receive any message from the example mechanism. We conclude that the function satisfies the two examples.
In addition to the example mechanism, which is only activated after each modification of the function, we have, in VIJISP, some specification mechanisms which are tested during each execution of the function. We call these mechanisms assertions, and we distinguish between input assertions, which are verified at the entry to a function, and output assertions, which are verified at the exit of a function.

In the picture above we have an example of an input assertion associated to the `dele` function. This assertion states that the second argument of the function, `liste`, should always be a list.

During the following execution, at each entry to the `dele` function, or more precisely - after each construction of a control frame, the input assertion is evaluated. If we would have an output assertion, it would be evaluated just before the destruction of the corresponding control frame. In the example given, we can see that the assertion is not verified for the last recursive call of `dele`.

We have also given an example of a pretty print of the `dele` function to show that the external aspect of a function is independent of both the number of additional activities attached and the number of existing versions.

Note that the pretty printer, like all tools of VIJISP is loaded only at the
time the user asks for it. This assures that only those tools which are active are loaded in the working memory and no memory space is lost for non used programs.

<table>
<thead>
<tr>
<th>Structure being edited: dele</th>
</tr>
</thead>
<tbody>
<tr>
<td>(el listes) (cond</td>
</tr>
<tr>
<td>((null listes) nil)</td>
</tr>
<tr>
<td>((equal el (car listes)) (dele el (cdr listes)))</td>
</tr>
<tr>
<td>(t (cons (car listes) (dele el (cdr listes)))))</td>
</tr>
<tr>
<td>? (mv h)</td>
</tr>
<tr>
<td>= t</td>
</tr>
<tr>
<td>? (Is dele)</td>
</tr>
<tr>
<td>= t</td>
</tr>
<tr>
<td>? (mv up)</td>
</tr>
<tr>
<td>? (mv u)</td>
</tr>
<tr>
<td>= t</td>
</tr>
<tr>
<td>? (add-entree (incr appel tailrec))</td>
</tr>
<tr>
<td>= (dele el (cdr listes)) entree ((incr appel tailrec))</td>
</tr>
<tr>
<td>? (setq appel tailrec 0)</td>
</tr>
<tr>
<td>= 0</td>
</tr>
<tr>
<td>? *</td>
</tr>
</tbody>
</table>

Illegal command encountered
(mv arg1 [num1] arg2 [num2] .)
where argn = (l, r, u, d, b)
and numn is number of times
to execute command argn.
Used move within function.
l = left, r = right, u = up,
d = down, b = home.

We have generalized this notion of assertions in admitting arbitrary user defined activities at the entry and exit of functions. For example, the trace facility is implemented as such an additional activity.

A further generalization of these mechanisms also permits these activities at the entry and exit of the evaluation of any expression (list or atm). In the previous picture, we introduced the incrementation of the variable appel.tailrec as an additional entry activity at the tailrecursive call of dele. Note that NVLISP may be put into a non-careful evaluation mode, in which all those additional activities are ignored. This special mode is the evaluation mode of programs which are considered correct.

Note also that the editor, as all other tools, gives a short description of its commands if they are invoked incorrectly. Naturally, this same description can also be obtained by asking explicitly, i.e.: by typing ‘? ’ and giving ‘mv’ as argument.
In the example given above, we have introduced two more supplementary activities: the recursive call of `dele` the incrementation of the variable `appel.rec` as entry activity, and at the point of the evaluation of `nil` the printing of the values of those two counters as an exit activity. As shown in the two following calls of `dele`, those counters do not modify the computation of the `dele` function, but they are superimposed upon it. Once more, neither the external form of the program, nor its behavior during
evaluation are modified by these additional activities. As one can see in
the tracing of `dele`, the function is still tail-recursive at one call and recu-
reive at the other.

Naturally, no programming help should ever modify anything in the
behavior of the observed program. (Otherwise one would create some-
thing we might call a `Heisenbug`.)

In order to conclude this scenario, let us have a look at the presently
existing documentation of the `dele` function:

```
<table>
<thead>
<tr>
<th>Structure being edited: dele</th>
</tr>
</thead>
<tbody>
<tr>
<td>((el liste) (cond</td>
</tr>
<tr>
<td>((null liste) -&gt; nil)</td>
</tr>
<tr>
<td>((equal el (car liste)) (dele el (cdr liste))))</td>
</tr>
<tr>
<td>((cons (car liste) (dele el (cdr liste)))))</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>? (2a dele)</th>
</tr>
</thead>
<tbody>
<tr>
<td>dele</td>
</tr>
<tr>
<td>assert-end</td>
</tr>
<tr>
<td>mapcar</td>
</tr>
<tr>
<td>subst</td>
</tr>
<tr>
<td>subst1</td>
</tr>
<tr>
<td>mind</td>
</tr>
<tr>
<td>mind1</td>
</tr>
<tr>
<td>dele</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>?*</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>objet: ? dele</code></td>
</tr>
<tr>
<td>type = <code>EXPR</code></td>
</tr>
<tr>
<td>local variables = el liste</td>
</tr>
<tr>
<td>uses = dele</td>
</tr>
<tr>
<td>used by = dele</td>
</tr>
</tbody>
</table>
```

During all interaction, the system continued, in the background, to
gather information about the user defined objects. We have two forms of
documentation about the `dele` function:

- The standard documentation obtained through the `?` command, which
  is still the same as it was just after defining it.

- The tree of the function calls of `dele`. This tree contains all the func-
tions called by `dele`, during this interaction with `BYLISP`. This kind of
  tree, which is given by the `2a` command, can also be used to observe
  (in real-time) the execution of programs. In this case the display of
  the call-tree is done in such a way that one can see, at every
  moment, the active functions. This allows the user to know at each
  moment of the execution the present control path, and to visually
  recognize the most intensely used part(s) of the program. This gives
  a very precise indication of where the program needs to be improved
  or optimized.
This capacity for collecting information parallel to the evaluation is very helpful for analyzing the activities of a program after its execution. Through it, the user can obtain call-trees for any one particular execution.

We end here our scenario of the use of BVLISP. The interested reader wishing a more detailed account of its capacities is invited to consult [Wertzel].

5. CONCLUSION

In this short presentation of the BVLISP programming environment we have concentrated on local tools, used primarily when working on individual functions. The more global tools will be presented in another paper.

We have not been able to demonstrate all the facilities BVLISP offers; we haven't presented its symbolic evaluation tool, its tools for propagating the influence of modifications, its history facilities, its tools for stepwise evaluation or its tools for event-driven evaluation.

Nevertheless, increasing evidence indicates that the usefulness of a programming language does not depend exclusively on the capacities of its instructions, but rather on the capacities of the associated programming environment. Programming is no longer an activity of a sole programmer, who manages to run the program despite the machine, but is a joint activity of the couple <programmer, machine>. Programming is done with the help of the machine.

Our system gives support to a single programmer or group of programmers working on a single, non partitioned project. It is intended for use by experienced programmers as well as by novices [Wertzel] and combines, for this reason, 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 to handle multiple viewpoints of programs as well as supporting the simultaneous existence of multiple versions of a program,

- evaluator handling these multiple versions of a program, able to choose the version to evaluate and able to accompany the evaluation of an expression by an arbitrary number of additional pre- and post-activities without changing the normal evaluation behavior.

The design permits a maximum amount of flexibility and ease of extension, every standard feature of the system can be adjoined by user-
defined similar features.

The system is intended to integrate the - until now dispersed - subsystems in a unified system of interacting modules. Many of its possibilities are invoked automatically, without the user's awareness, such as the fact that modifications on one level of description are automatically propagated to all the other dependent levels.

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

Further developments include the adaptation of the system toward more sophisticated display techniques, especially for the cursor motion of the editor.

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

At the time of this writing, the implementation of the first version of wVLISP is completed. Practical experience in the use of it, by novice programmers at our University and by expert programmers at our CNRS laboratory will help us to evaluate its capacities and to define its next (and better) version.

6. REFERENCES


