(Preliminary version of book chapter, in *The Future of Computer Science Education*, A. Greening (ed.))

Towards truly educational programming environments

R. Jiménez-Peris*, C. Pareja-Flores**, M. Patiño-Martínez* and J. Á. Velázquez-Iturbide***
* Facultad de Informática, Universidad Politécnica de Madrid,
Campus de Montegancedo, 28660 Boadilla del Monte, Madrid, Spain, {rjimenez,mpatino}@fi.upm.es
** Depto. de Sistemas Informáticos y Programación, Universidad Complutense de Madrid,
Avda. Puerta de Hierro, 28015 Madrid, Spain, cpareja@eucmax.ucm.es
*** Escuela Superior de CC. Experimentales y Tecnología, Universidad Rey Juan Carlos,
Camino de Humanes 63, 28936 Móstoles, Madrid, Spain, a.velazquez@escet.urjc.es

Abstract

Software development education needs programming environments and tools (PETs) adequate to support its teaching. Professional PETs are often adequate for experienced users, but not for novices, because they do not support an adequate degree of abstraction. Therefore, it is necessary to develop PETs especifically designed to assist in software education. In this chapter, we study the past, present and future of educational PETs. Our point of view is goal-oriented, so we identify educational needs as a basis for the description of the key features PETs must exhibit. We study their past by giving an overview of the main realizations in the evolution of PETs. We also describe the present and foresee the future by focusing on current lines of research and development which are critical for an educational use: novice-friendly facilities, debugging, visualization, and support for software development and assessment (mainly with respect to correctness and efficiency).

1 Introduction

Software development is an engineering activity of an abstract nature [Bro87]. In effect, programs are very complex and sophisticated artifacts, whose nature is logical rather than physical. Their corresponding building tools, programming environments and tools (PETs), reflect this abstract nature. Software development education needs PETs adequate with an adequate degree of abstraction as to support its teaching and learning. Professional environments and tools are often adequate for experienced users, but not for novices, because they do not support a high enough degree of abstraction. To alleviate this situation, educators often find convenient or even necessary to develop PETs especifically designed to assist in software education. In this chapter, we study the past, present and future of educational PETs.

The structure of the chapter is as follows. The second section briefly gives an overview of the evolution of PETs in general, just to give a perspective on the main realizations in the area. Our point of view is goal-oriented, so we identify, in the third section, educational needs as a basis for the description of the key features PETs must exhibit. The following sections describe current lines of research and development in PETs which are critical for an educational use: novice-friendly facilities, debugging, visualization, and others; extensions for supporting development and assessment activities are also studied. Finally, we present our conclusions, mainly in relation to future trends of research and development in the field.

2 Historical Evolution

We briefly review in this section the evolution of PETs during the last decades, showing the main stages and landmarks in the development of the current state of the art. We do not make any difference between educational and professional PETs in order to show the relevant contributions to the area. In subsequent sections, we focus on educational PETs.

The state of the art in hardware is one key aspect that has determined the organization of operating systems and human-machine interface. In turn, these ones have influenced other kinds of software, in particular programming environments and programming languages. Due to this constraint, during the first decades of computer science there

were few opportunities for powerful programming environments. The batch way of working of the first operating systems only allowed the use of compilers and some low-level debugging facilities. These systems have had a deep influence on the way of working in computer science for many years, namely a sequential philosophy of working. In fact, it can even be seen today in many interactive programs, e.g. word processors or programming environments.

The invention of the interruption concept in the sixties introduced the development of time-sharing operating systems, where several users have the illusion of being the only user of the system. In addition, the development of consoles gave rise to the first full-screen editors. Software gained in complexity, giving origin to the term "software crisis".

During the seventies, the rate of new developments increased dramatically. High level languages were designed and implemented, ranging from imperative ones (mainly, Pascal and C) to declarative ones (mainly, Prolog and ML). In addition, the Unix operating system provided the programmer with a new way of thinking, based on a standard way of communicating programs by means of text files [Ker84]. Programmers could develop general purpose tools (such as *make*, *grep*, etc.) that could be used by other programmers, thus providing a reuse mechanism. In this way, a new class of programming environment emerged, toolkit programming environments. There also were a number of developments that, limited during the seventies to research or internal production, set the basis for the software of the next two decades until today: window-based user interfaces, the mouse and object-oriented programming (Smalltalk).

The early eighties are the reference for the explosion of PETs. The birth of personal computers takes place early in this decade, and they are quickly popularized by their low prices. From the point of view of education, it meant that computer resources became available to small institutions, teachers and students. These cheap computers are most of the time wasting their computing power, simply waiting for user input. As a consequence, computers can dedicate more computing resources to input-output, so the most important part of a program moves from its functionality to its user interface. In perspective, this process produced a change of paradigm (yet uncompleted), from batch based processing to interactive processing; in other words, from computing to interaction.

Although some excellent programming environments were built in the seventies, mainly Lisp [San78] and Smalltalk ones, the increasing importance of the user interface fosters their extension in the eighties. Language-based environments are integrated, providing a set of tools under a single program, simplifying the programming task. Tools are tailored to a programming language, providing programming aids for the coding phase of program development: typically, a full-screen multi-window editor, a compiler or interpreter, a debugger and support for other mundane tasks (e.g. file handling). Two landmarks in language-based programming environments were Logo for educational software [Pap80], and Turbo Pascal for professionals.

Other category of programming tools, debuggers, was also improved, giving place to the so-called high-level debuggers. They are usually integrated in a language-based programming environment, and allow the programmer to debug programs by thinking in a way closer to his/her mental models than previously, when low level implementation details had to be known.

Due to the increasing attention to users, programmers also received attention as users of programming environments. Syntax-directed editors, such as the Cornell Synthesizer [TR81], offer a new way of programmer-computer interaction, and make use of computing power for more powerful operations than simple text-edition, namely edition operations (insertion, removal, etc.) based on the language syntax and semantics. Later improvements drove to editor and programming environment generators. Although they are not as widespread as the technology promised, they set the idea that the computer could make use of language features to aid the programmer. They also showed the importance of the adequacy of user interface operations to user mental models, as opposed to the criterium of only choosing operations by their power.

The eighties also saw the development of educational programming environments. Usually called programming tutors, they are integrated environments which lack some of the tools of professional PETs, but they provide assistance to the novice student [AR85]. Assistance can vary from a good support for error explanation to advice on how to proceed in program design (intelligent tutors). They have not succeeded as expected, especially their artificial intelligence behaviour has been limited, but they helped in making the student a customer to consider.

All these trends are expanded in the nineties in several directions, often as a consequence of the synergy of different tecnologies. For instance, debugging or assistance to students are improved by higher level aids, but also because of the dramatic improvements in graphical user interfaces. In the following sections we describe the most important

improvements and their perceived future.

3 Requirements for an Educational Use of Programming Environments and Tools

Educational PETs must fulfill the same basic requirements than any professional PET: to assist in the development of computer programs. However, they must satisfy slightly different requirements, resulting from their educational use:

- PETs must be up-to-date systems, based on up-to-date programming languages, so that students can make use of the best ones [Den88]. This feature is simply a pragmatic corollary of asking for an up-to-date university education. Although there is a range of opinions about the importance of this point, positions that reject it are marginal.

  One consequence of this requirement is that complete and good systems must be provided. This restriction is not only stated with respect to compilers or similar basic systems, but also with respect to user interfaces and the environment functionality. Thus, advanced technologies which are being used successfully in other areas (e.g. graphical user interfaces or hypertext) must be provided.
- PETs must provide a framework consistent with the mental model of students. As their mental models are very influenced by the explanations given by the teacher, PETs should at least be consistent with such explanations. As a consequence, PETs must not simply mirror the implementation (quite reasonable for professional PETs), but they must provide higher level working models. In addition, they must not complicate explanations with myriads of secondary details about the programming language, the user interface or whatever, but they must allow a direct experimentation with the topic at hand.
- PETs must emphasize assistance in understanding, not only assistance in development. As a consequence, better support must be provided for explanations, especially in relation to errors.
- PETs must support non-academic activities, such as group work, and oral and written communication, with an increasing importance in university education. As a consequence, the concept of programming environment is moving to a new concept of integral educational environment, where programming will be just one activity.

These requirements lead to PETs that exhibit different facilities that those in professional PTEs. In the next sections we give the perspectives in the fields where these requirements impose more critical consequences, namely novice-friendly facilities, higher-level debugging aids, and activities related to programs development and assessment. Finally, visualization is a technique that is can help in all these activities, so we study it by its own.

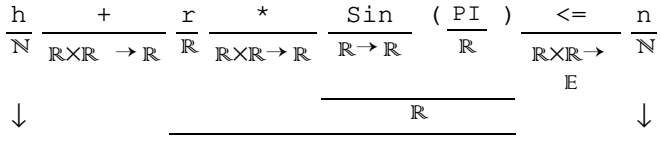
4 Novice-Friendly Facilities

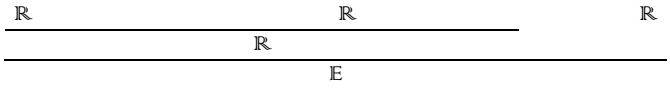
Professional programming environments are powerful enough to cover needs of expert programmers, who know and use advanced techniques oriented towards large-scale programming. On the other side, these environments lack other features, more convenient in order to make easier programming learning and teaching. Concretely, an important goal of educational PETs is to provide assistance to novice programmers in several aspects, like dealing with partial programs, helping them to identify typical mistakes, enforcing good methodological and style habits, etc. In this paragraph, we review these needs of programming students, pointing out some solutions.

4.1 Working with Partial (or Incomplete) Programs or Languages

This possibility consists in allowing the student to practice before learning all the many details related to complete programs [Webber96], such as program structure, type declaration, input and output, etc. There are several partial aspects:

- Reasoning on data types. This is a non-trivial subject for novice students, especially in languages with complex type systems, as functional polymorphic languages. Data types in (sub)expressions can be inferred from some given declarations, and explained. For instance, the following type inference shows a typical type analysis in imperative programming:





- The syntax of expressions, declaration sections and other structures can be similarly explained by means of syntax trees or diagrams.

- Another aspect related to the static semantic is the scope of the declared identifiers. Concretely, when subprograms are learned, it is convenient to make clear a few typical conflicts and situations with names, like its scope, hiding, etc.

- Running program fragments. Much flexibility is gained by allowing running program fragments, testing isolated expressions or instructions, without the need to write the full protocol of a program.

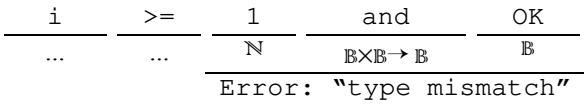
- Limitation of language functionality. A pedagogically useful possibility is to extend progressively the functionality of the language (and the environment). For example, first the environment allow working with simple expressions, later with more complex constructions and finally with objects or concurrency. An example of this feature is provided by DrScheme [F+97], where 4 levels of Scheme proficiency are provided.

- Moreover, the environment could change some of the language elements, according to some criteria. For instance, at the beginning it can be useful to relax the data type system (integer vs. real), or to change the domain of standard data types (by increasing the value of MaxInt or the precision of real numbers, or by choosing the used character set, etc.) In functional programming, it is interesting the possibility of choosing the evaluation strategy, strict or lazy.

  Other interesting feature is to use a different set of keywords, customized to the student mother language, which is constraint to English as he/she knows more about programming. Finally, the aspects related to style and methodology can be integrated (to allow or forbid the use of global variables, change the scope rules, allow or not side effects in procedure or functions, etc.), and are further explained bellow.

4.2 Advanced Help Systems and Error Diagnostics

For novice programmers, the help system is a frequently used tool, that provides them useful clues about many aspects related to the program under development, the programming language and the programming environment. When an error situation arises, an advanced error diagnosis system can suggest possible solutions, not simply identifying it. As an example, a possible reaction of the compiler to a "type mismatch" error is to offer an analysis of the operation priorities and the (sub) expressions type:



This analysis can make evident the cause of the error, like in the previous case, where the priorities of the operations have been probably misunderstood.

An example of advanced error diagnosis is the provided by the Gnat/Ada'95 environment that considers the layout of the text in order to guess the possible cause and solution of the error.

As prevention against some syntax errors, it is interesting the use of colorful highlighting and syntax-oriented editors, making obvious the structure of delimiters and programming structures. An example is the Golden Common Lisp environment, that points out the most internal unclosed parenthesis.

4.3 Style Enforcement and Other Methodological Habits

Educational PETs can promote good programming style. There is not a universally accepted style indeed, so this

feature must be custimizable, allowing several styles. Compilers can assist in enforcing aspects like a suitable indentation style (the Gnat Ada compiler offers an option to enforce this), capitalization of identifiers, etc.

A compiler can be stricter than the language itself, forbidding the use of global variables, functions with reference argument, unstructured "returns" in functions, etc. Moreover, some comments can be considered mandatory: to specify subprograms, to label the end of a programming structure, to say the mode of parameters, etc.

Related to the both, methodological habits and partial programs, an aspect in which future PETs can help the student it the development of programs by means of stepwise refinement. During this process, code and pseudocode coexist, like in the following example,

```
for i := 1 to N do begin
  k := "the index, within [i..N], of the min element in v[i..N]"
  "interchange v[i] with v[k]"
end {for}
```

but current environments cannot deal with it until a full structure of program is achieved. Educational PETs can accept this "protoprograms", handling its partial structures.

5 Debugging Aids

The most important debugging aid is the breakpoint. A breakpoint allows stopping the execution of a program in a given location specified by the programmer. There are different kinds of breakpoints:

- Step breakpoints: Execute a computation step of the program.
- Unconditional breakpoint: It stops the execution when it is going to execute a given program construction.
- Count breakpoint: However, previous breakpoints are not enough, because in repetitive processes one can be interested in the 100^{th} iteration, and it is not reasonable to press a key 100 times. This kind of breakpoint stops execution when a program construction is to be executed the n^{th} time.
- Conditional breakpoint: This kind of breakpoints is very useful when one is interested in stopping the program when it reaches certain situation, but the number of iterations is unknown. They are more complex as they require interpretative capabilities from the debugger. It stops the execution when a program construction is going to be executed and a given condition is satisfied.

In the next sections the different kind of breakpoints are particularized to each paradigm.

5.1 Debugging in Imperative Programming

Computation steps in imperative programming are programming statements. In particular, the step breakpoint executes the next program statement. A variant of this breakpoint is the skip breakpoint that executes the whole statement, even if it invokes a subprogram, while the previous one would execute the first subprogram statement. Unconditional breakpoints stop the program when it reaches the statement marked with this breakpoint. Count breakpoints stop the program when a statement is going to be executed the n^{th} time. Conditional breakpoints stop the program when it steps through a statement and certain predicate is satisfied. An example of a predicate for such breakpoint is `(count mod 24 = 0) and not found`.

Breakpoints in recursive subprograms are very tricky, especially when something is done at the return of recursive calls (i.e. after a procedure recursive call). A breakpoint in the basic case will unfold all the recursive calls. In this situation, if recursion is visualized as suggested below, that is, by showing each activation in a different window, each window will display the values of the parameters and local variables of the call. Now, the students can focus in any particular activation just setting an unconditional breakpoint in a statement of the chosen activation. The advantage of this approach is that they do not need to think the predicate of a conditional breakpoint.

Another very interesting facility for educational purposes is backward execution. It is usual that a student can become interested in what happened before reaching an interesting state during the algorithm execution.

Without backward execution it is very difficult for a student to start the execution and get the same point at a reasonable speed. This facility is not easy to implement with debuggers based on assembler generated by a compiler, but it is not too difficult in interpretative environments (high-level or low-level), for instance DynaLab [Boroni96].

5.2 Debugging in Functional Programming

Debugging in a functional language may use mechanisms similar to those explained in previous sections (e.g. breakpoints), although based on a different computational model. The operational semantics of functional programs is a reduction or rewriting process, where an arbitrary expression is repeatedly rewritten into another one until an expression in normal form is obtained -its value-. Particular details of this basic scheme depend on the particular programming language and its underlying evaluation strategy, either eager or lazy evaluation. The examples we include in this chapter are based on an eager implementation of the functional language Hope+ [Per89].

For example, let the following definition be for the factorial function:

```
dec fact : num -> num;
--- fact (n) <= if n=0 then 1 else n*fact(n-1);
```

A part of the evaluation of `fact 4` follows, where we bold the next subexpression to reduce at each step (technically speaking, the redex). In any subsequent evaluation example, se include the data type of both the original expression and its value, and we bold the redex of each intermediate expression.

```
fact 4 : num
↓
if 4=0 then 1 else 4*fact(4-1)
↓
if false then 1 else 4*fact(4-1)
↓
4*fact(4-1)
↓
4*fact(3)
↓
...
4*(3*(2*(1*1)
↓
...
↓
24 : num
```

Showing only the final result is not of much help when faced to unexpected results or evaluation errors, so debugging aids customized to functional programming must be based on this operational semantics, where the current state of a computation -of the evaluation of an expression- is the current intermediate expression [Vel94, F+97]. A minimal aid consists in showing the evolution of evaluation step by step, either by showing at every step the redex [AR94, TL92] or the whole intermediate expression [Vel94]. The latter case has been illustrated with a part of the evaluation of `fact 4`.

This way of advance is useful for small problems, but the amount of intermediate expressions is too large for more complex problems. In general, more flexible ways of advance are necessary. As in imperative programming, the use of breakpoints allows to control where to stop a computation, omitting uninteresting intermediate steps. Many of the breakpoint facilities given above for imperative programming can also be used here, such as step breakpoint or count breakpoint. Other usual aids can also be directly adapted, such as skipping a part of the computation (here, the evaluation of the current redex) or evaluating backwards [F+97]. In particular, skipping over the rewriting of a subexpression (usually the application of a function [AR94, HCD88, TL92]) provides a view similar to common tracing.

It is worth to consider how to define breakpoints, since statements do not exist in functional programming. Several solutions have been or can be proposed. A natural definition for unconditional breakpoints is the

application of a selected function [Vel94] or in general a part of a program, i.e. a subexpression [F+97].

For instance, let the computation of `fact 4` be the following. Marking the factorial function with a breakpoint, the evaluation progresses as follows:

```
fact 4 : num
↓
4 * fact 3
↓
4 * (3 * fact 2)
↓
4 * (3 * (2 * fact 1))
↓
4 * (3 * (2 * (1 * fact 0)))
↓
24 : num
```

Notice that, in a more general case, we are interested in defining conditional breakpoints that identify when to stop. Their conditions cannot be expressed in functional expressions as predicates over the variables contents, since variables do not exist in the imperative sense. Conditions about the state of a functional programming computation must refer to the current expression. Thus, a good choice of condition is that the redex matches a given pattern.

For instance, we can be interested in stopping the computation of `fact(4)` when the factorial of 0 is to be reduced (the base case). This can be achieved in a rather involved way by estimating the number of rewriting steps that will be made until that situation is achieved. Alternatively, we can advance from recursive to recursive application of `fact`, as in the previous example. A third option is simply to advance until the redex equals `fact 0`. In this example we have specified a ground expression, but more general patterns could be specified as well. Thus, we could specify to stop when a given equation is to be applied, from those defining a function. This option is especially useful for functions where some cases are trivial and very abundant.

For instance, given a usual definition of a function working on lists, we could stop when the first equation (usually the base case of an empty list) or the second equation (in this case, a non-empty list) is to be applied. More specifically, let the following definition of a function that computes the last element in a list:

```
dec length : list(alpha) -> num;
--- length (nil)    <= 0;
--- length (x::ys) <= 1 + length (ys);
```

We could specify that we define as a breakpoint those rewritings of the first equation, whose head is `length nil`. The corresponding evaluation of `length [5,2,9]` is:

```
length [5,2,9] : num
↓
1 + (1 + (1 + length []))
↓
3 : num
```

In summary, there is a trend to provide debugging aids, as versatile as those available for imperative programming and customized to functional programming. Currently there are some early steps in this direction. Thus, DrScheme [F+97] and WinHIPE [Vel98b] give aids in the style of those described above for the functional languages Scheme and Hope+, respectively. Similar interactive debugging tools for lazy programming languages, such as Haskell, still are missing.

6 Other activities

Support to education must be provided for every activity of software development. In fact, programming in the small is only a part of software development education. Programming in the large, and more generally software

engineering, must be an integral part of computer scientists education. Furthermore, there is an increasing stream that advocates for the integration of software engineering activities from the very beginning. Consequently, programming environments and tools must be extended to cover other software development activities. There are such a variety of useful environments and tools as different activities, but only a few are mature enough to consider a widely accepted integration in the curriculum. The main such activities are analysis and design, and testing.

In the previous sections, we have described in detail facilities for coding activities. Although there are other programming activities (even software engineering activities) which also can benefit from aids in educational PETs, we cannot describe them exhaustively here. Therefore, we outline facilities that educational PETs can provide for the activities most closely related to coding: support to program correctness and efficiency, and program design and analysis. In particular, the first two activities can be made either in a theoretical way or in an experimental way: verification vs. testing, and complexity analysis vs. performance measures.

6.1 Program correctness

Program correctness can be guaranteed either during development or *a posteriori*. The first trend tries to gradually develop programs from their specification through a number of steps. Depending on whether the specification is executable, this process is called program synthesis or program transformation. Program synthesis is a formalization of a well known educational method to develop correct programs: stepwise development by Dijkstra, Wirth and others. Program transformation is based on the assumption that program development is usually an iterative process, where programs are developed through several versions.

Both synthesis and transformation are particularized in a set of informal methodologies or formal systems, but there is seldom any computer assistance to them. In fact, efforts to provide assistance to this process are nowadays restricted to the research community. However, support to these activities should be included in educational PETs.

With respect to the *a posteriori* guarantee of program correctness, there are two approaches: verification and testing. The strongest educational stream advocates for verification as the primary way to ensure program correctness. Verification guarantees correctness, but it is not supported by programming environments, so it is prone to human error. As in program synthesis and transformation, we expect efforts to integrate facilities to support program verification.

Testing is more intuitive to use, but it is hardly applied in a systematic way and it only allows detecting existing errors, not missing functionality. It has traditionally been neglected from programming education until software engineering courses. While intellectually consistent, this position is not realistic with respect to several facts. First, students (and teachers) test their programs at some moment, on an intuitive basis and without a solid foundation and planning. Second, verification is a well-known activity for certain classes of algorithms, but there are practical problems to extend these techniques to larger programs. In some cases, even verification techniques are useless, such as in user interfaces. Third, verification is seldom used in practice at the industry. Finally, there is one more practical reason to teach testing from an educational point of view: assignment grading systems are becoming extended. Many of these systems are based on testing, so it is consistent to teach students those techniques upon which their assignments are to be graded.

The above reasons make necessary to teach testing techniques from the beginning. In addition, it would be desirable to have available some testing tools in programming environments. Some practical tools are: test data generator, file comparator, and test manager. A test data generator generates test data from specifications. A file comparator compares expected results to actual results. The test manager handles the testing process in a consistent and meaningful way.

There are few educational PETs supporting testing. Algolib [VCL96] is a library of modules which simplify students several tasks, mainly algorithm testing and time measurement. In particular, procedures are available to input and output data from or to different channels. After developing an algorithm, students must integrate it in a program declaration and to include several library calls for input and output. In this way, students with not deal directly with the input/output features of the language (here, Modula-2). For instance, in the figure, input data are being asked for the problem of dividing a set of integers (represented by an array) into two subsets with the same sum. The input process is generated by the library call

```
Inputs.ProduceIntVectorKeyboard (s, "set", 1, N, Outputs.HVector);
```

where the student specifies the name of the array in the program (*s*), the name to visualize for facilitating the input (*set*), the index of the arrays and the format of the array display (horizontal).

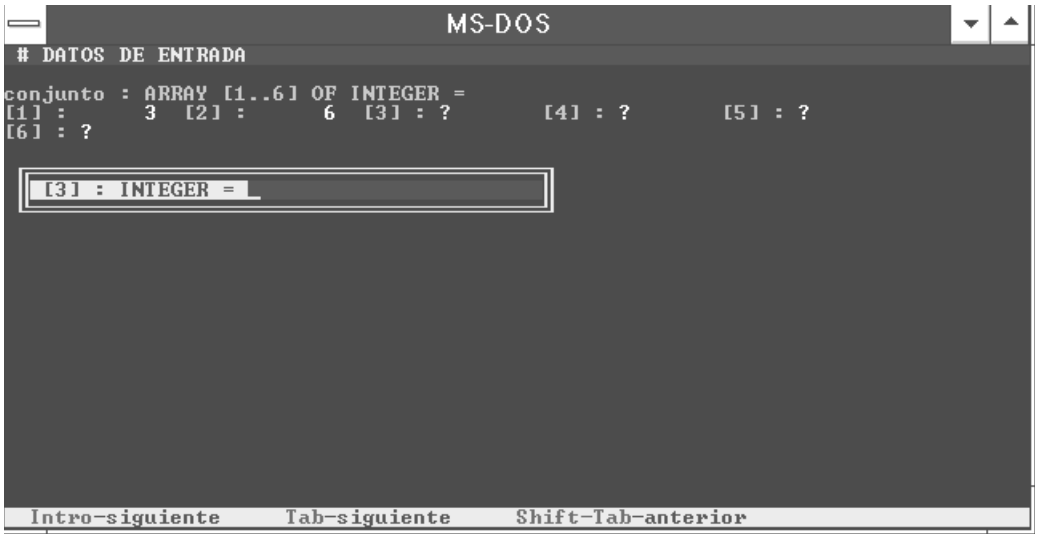


Fig. 1: Array Input

Trainer [Webber96] also includes some facilities for black-box testing. In Trainer, the student burden is even lighter than with Algolib: it must simply develop the algorithm, and the environment integrates it in a program and tests it. However, it is restricted to the set of examples and input data fixed in advance by the instructor.

6.2 Performance measurement

Performance measurement is a common exercise in algorithm analysis. Commercial PETs include some facilities for time measurement, but they are too naive. For instance, a well-known fact [HS90] is that the accuracy of the clock included in many PETs is very low. In addition, the only facility provided by most languages is a procedure `GetTime` that provides the current time. As a consequence, the programmer must worry about looping the algorithm in order to reach a duration that the clock can detect, and also about computing the difference between the initial and the final time. All these details should be hidden to students at the beginning and they should always be simplified.

Algolib [VCL96] simplifies these tasks with a module for performance measurement. With respect to time measurement, it provides much meaningful and abstract procedures: `StartTime` and `EndTime`. The former marks the start of timing; the latter has an input parameter where the student can specify the number of iterations the algorithm has been executed and an output parameter returns the elapsed time in hundredths of a second. It also provides some other procedures to organize the collection of other measures related to a total number of events and the maximum number of events at the same time. These measures allow counting, for instance, the total number of executions of a given instruction or operation, and the maximum depth of recursion.

6.3 Analysis and design

Analysis and design is one of the first phases that are immediately considered when transiting from small programs to medium size ones. CASE systems [CACM92, IEEE88] usually support design and analysis, usually based on graphical notations corresponding to structured design methods. The main tools included are: diagram editors, diagram analysis tools, data dictionary, documentation tools, forms definition tools, and code generators. If the diagram editor is tailored to some particular diagram, diagram analysis tools can be integrated into the editor. Code generators are tools necessary to keep consistency between the design and the application code, hence their educational importance.

There are a number of experiences in the use of CASE tool in design or software engineering courses (see e.g. [GL96, Gor94]). There are some well-known practical problems for their use, such as high cost, difficulty of

learning and support from commercial houses. In spite of these difficulties, some elements are key to any non-trivial software development activity, mainly graphical tools and code generation. We foresee that these facilities will be integrated in programming environments, ideally with novice-friendly facilities. For instance, one important issue for these extended programming environments will be whether they use graphical representations, which are independent from any method or class of diagrams.

7 Visualization

Nowadays, an increasing number of programming environments has the label "visual". However the term visual is overloaded with different meanings, although all of them are related to graphical representations. Several categories can be established among visual environments.

The most extended is the one related to GUIs (Graphical User Interface) programming. In this category the program editor is integrated with the resource editor (interface elements editor). The editor can open a "form" of the application, and the code associated to elements of the GUI can be accessed "visually" by clicking with the mouse on the element. Well-known examples are Borland Delphi and Microsoft Visual C++.

Iconic languages like Authorware form another category. In these languages each control structure is defined by an icon, subprograms are represented by icons, etc. They are purely "visual", and they are interesting in programming teaching. They are very intuitive, so they are very adequate for introductory courses, CS0, to introduce programming in high schools, etc. However, they are not powerful enough for non-introductory courses, i.e. CS1.

Environments that deal with non-visual languages but represent part of the behavior "visually" form another category. Many program animators can be included here; for instance, imperative programming environments that present graphically dynamic data structures. One of their advantages is that visualizations are automatically generated from the source code of the program, in contrast with algorithm animators that require a lot of work to prepare a particular animation. Liberating users of programming environments from the task of codifying visualizations makes those environments useful to assist explanations during lectures, as student notes, and to help students to debug their programming projects. Future educational programming environments should be visual in order to assist in all these tasks.

This section will center on the visual facilities this kind of environments should offer. Some visual facilities are useful for programming languages from different paradigms (e.g. recursion is used for imperative, functional and logic programming), while others are highly coupled with a particular paradigm: dynamic memory in imperative programming, expression rewriting in functional programming and search trees in logic programming.

7.1 Visualization in Imperative Programming

Two aspects of imperative programs should be visualized during program execution: the state of the data and its control flow. Debuggers visualize control flow highlighting the statement in execution and displaying the stack of pending calls. Those types of visualization are useful but they do not display recursion properly. Several elements of the state of the program that are currently neglected should be taken into account in order to provide truly educational programming environments; for instance dynamic memory, parameter passing mechanisms and variable scope and visibility. In the next section, it is described how recursion should be visualized. Then, methods to visualize dynamic memory are presented. Finally, visualizations of other elements are discussed.

Recursion Visualization

Many students find difficult learning recursion. The main reason is that they do not have enough abstraction ability, so they need to concretize recursive algorithms in order to "see" them. Visualization plays a key role here as it can show different aspects of the execution of a recursive algorithm, showing the idea of the repetitive process behind recursion.

Professional imperative programming environments already provide a display of recursion, but very limited.

They just show the state of the call stack, that is, the pending calls in the stack including the current call. Some of them allow inspecting the arguments of the call. The main deficiency is that they just show one window with the code of the recursive subprogram. If students step the execution, they just see the cursor dancing up (when stepping into recursive calls) and down, but always over the same code, what results very confusing. Recursive procedures are still more difficult to understand than recursive functions as they modify their environment (reading or writing or changing arguments passed by reference). A very simple recursive procedure is the one that writes in reverse order a line in the input. This simple procedure results to the beginner very difficult to understand, particularly because it makes things both before the recursive call and after it.

An intuitive visualization of recursion progress consists in displaying the state of each call in a separate window, perhaps overlapping all of them. This state will include:

- The text of the recursive subprogram.
- The cursor highlighting the current statement in execution.
- The local variables and their value in that call.
- The parameters of the call and their value.

This visualization gives students more intuition because they can "see" each activation of a subprogram, as there is a window per activation. They can also observe how parameters change with the progress of calls, and so they "see" the repetitive process behind recursion. Additionally, each window is identified with a number indicating its position in the call stack.

There are other important details that help in understanding a recursive function. In some languages like Modula-2, there is an instruction to return the value of a function. In addition to this, this instruction returns control to the caller, so the student cannot see the value returned by the function, if it is calculated within an expression. This makes difficult to understand the function. The only way would be to evaluate the expression just before executing the return instruction. This can be solved showing the result returned by the function, when the return instruction is executed and before closing the window corresponding to the call (see Fig. 2).

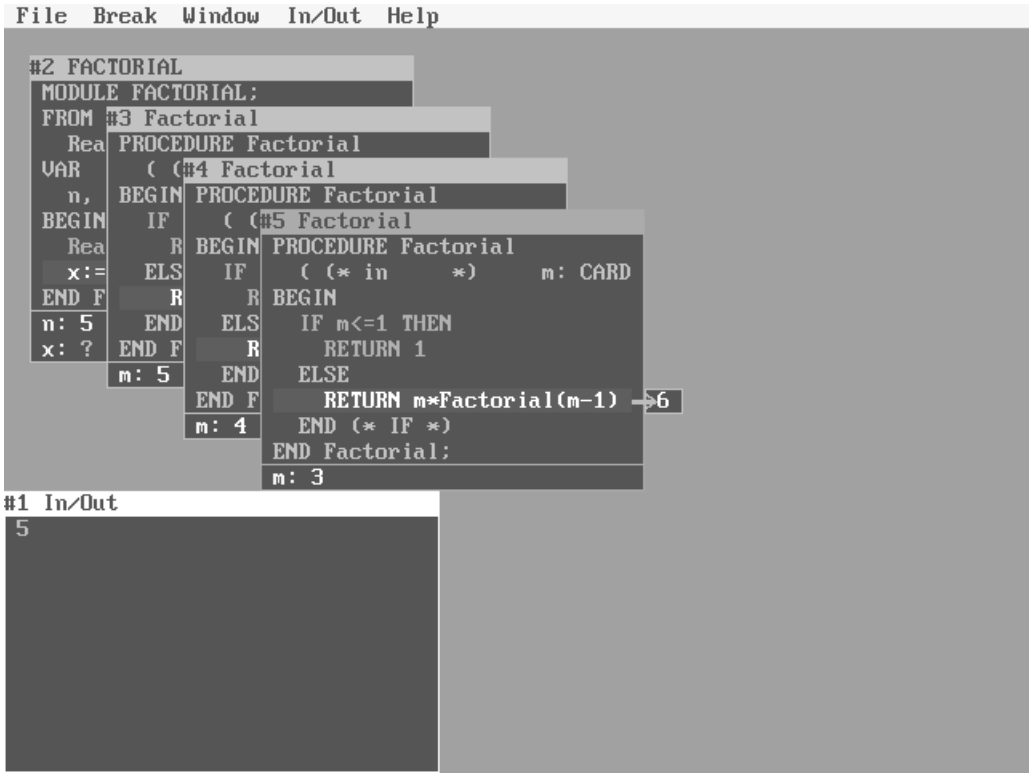


Fig. 2: Recursive Factorial

Until now, we have focused on the dynamic aspects of the behavior of recursive functions. However there is another view that is also interesting from a pedagogical point of view.

Another interesting view of recursion is that showing the tree of calls that a recursive subprogram generates.

That tree contains both pending and finished calls together with the value returned and their parameters. This view helps to see how many calls are generated, and in case of multiple recursion in which order are executed, without taken into account the code. In this way it can be seen how a value is constructed by the different calls. This is especially interesting in the case of multiple recursion, that is, generating multiple recursive calls in each call, e.g. the recursive version of the Fibonacci function.

Dynamic Memory Visualization

Dynamic memory is one of the hardest topics for CS2 students. The difficulty is partly conceptual (although this can be helped with better ways of introducing the topic as the Locker Metaphor [JimParPatVel97]), but it is also due to poor facilities to represent dynamic memory in existing environments. Most debuggers only show memory addresses as pointer values, although they allow to traverse a chain of pointers. This view is non-intuitive at all for beginners that are used to a more abstract view of dynamic memory, that is, the graphics drawn by the instructor.

There are many works on displaying dynamic structures, although only a few programming environments have this facility. Our opinion is that future educational environments will have specialized visualizers for structures of different nature: linear, linear double-linked, linear circular, trees, dags, graphs, etc. The reason is that the aesthetic criteria depend on what is going to be visualized (in fact, there are some conferences every year devoted to graph drawing). A dynamic memory cell with two pointers must be visualized differently depending on whether it corresponds to a double-linked list, to a general list or to a binary tree.

We will next describe pedagogical ways to show some typical structures.

1. Linear Structures

The visualization of a particular linear structure is the simplest one. Pictures like the one shown in Fig. X are clear enough. The problem comes when there is more than one pointer and more than one list. For instance, when an auxiliary pointer is used to traverse a list, the pointer pointing to the head of the list (we will call it the root) must be given priority, that is, it must appear at the beginning of the list all the time, and following it the elements of the list. The auxiliary pointer must appear below or above the pointed cell. This scene is depicted in Fig. 3.

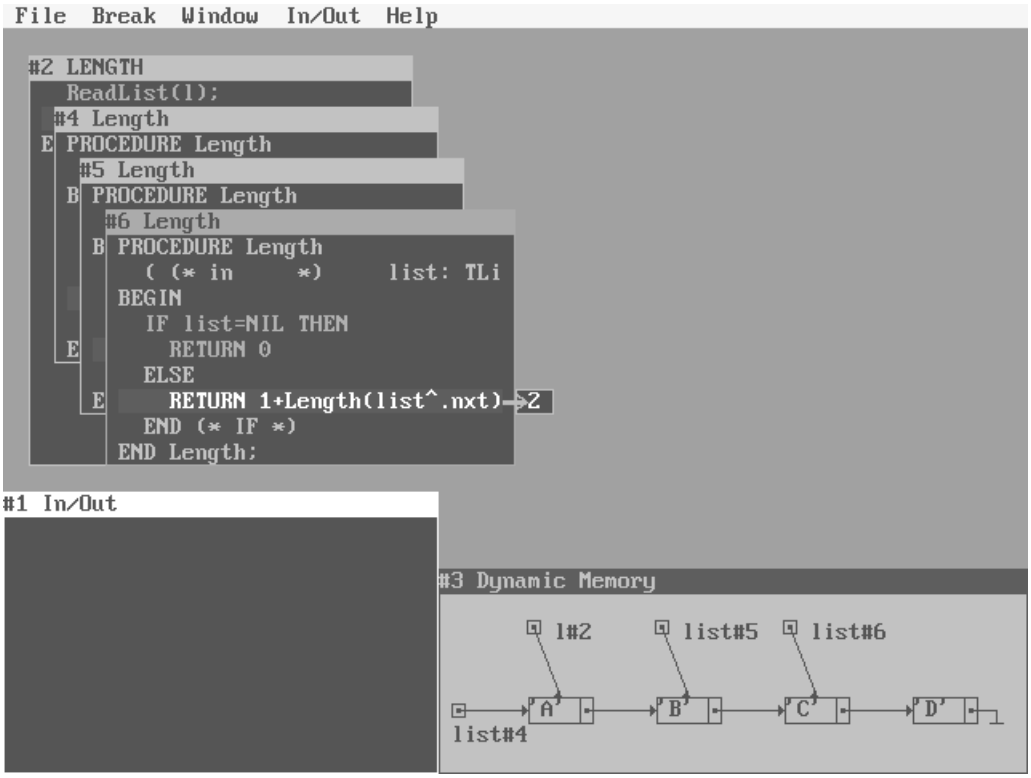


Fig. 3: Length of a list

When there are several lists (for example, to concatenate two existing lists), it is important to minimize changes on the screen between successive visualizations to give a clear picture of what is happening. If many changes are introduced in a single step it is difficult to follow them.

Circular and double-linked lists are particular cases of linear structures that should be also visualized (see Fig. 4).

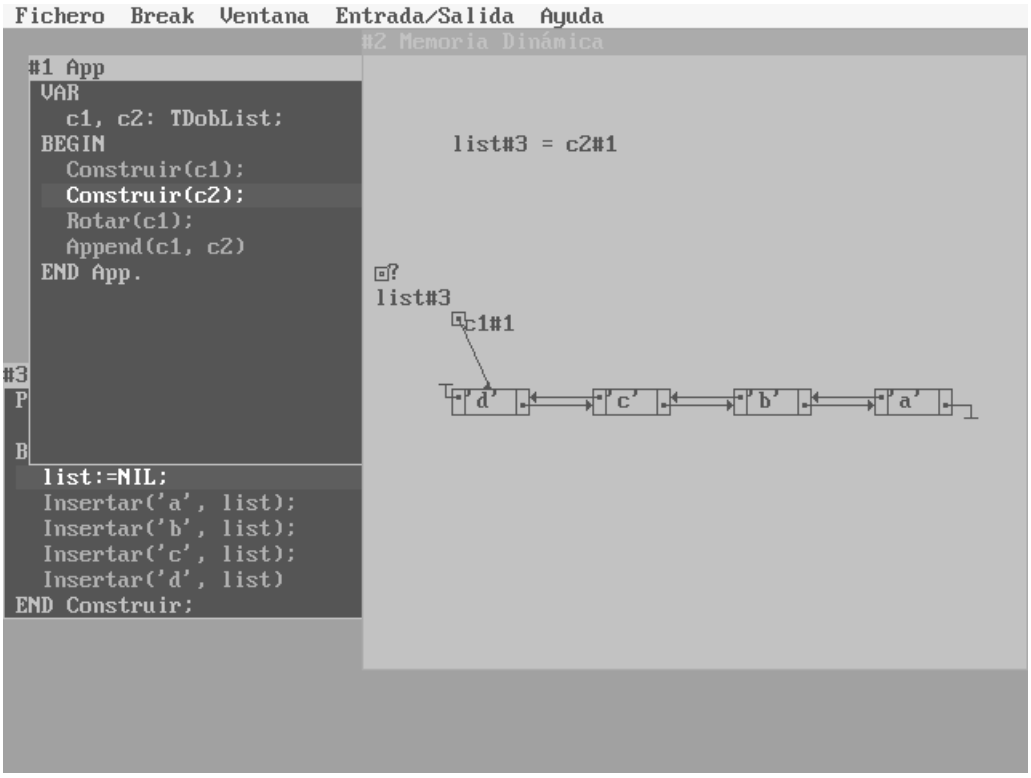


Fig. 4: Double-Linked list

2. Non-linear structures

Visualization of non-linear structures is more difficult than linear ones. The simplest ones are trees and general lists. Let's start with binary trees. They can be displayed in many ways according to different aesthetics [Wetherell79, Vaucher80, Reingold81]. In a pedagogical programming environment the aesthetics to be chosen must have the following features: intuitive, easy to read and compact (See Fig. 5).

Other non-linear structures like graphs and their variants are more difficult to represent, a good criteria would be to minimize the number of edge crossings, but other criteria could be used [DiBattista94].

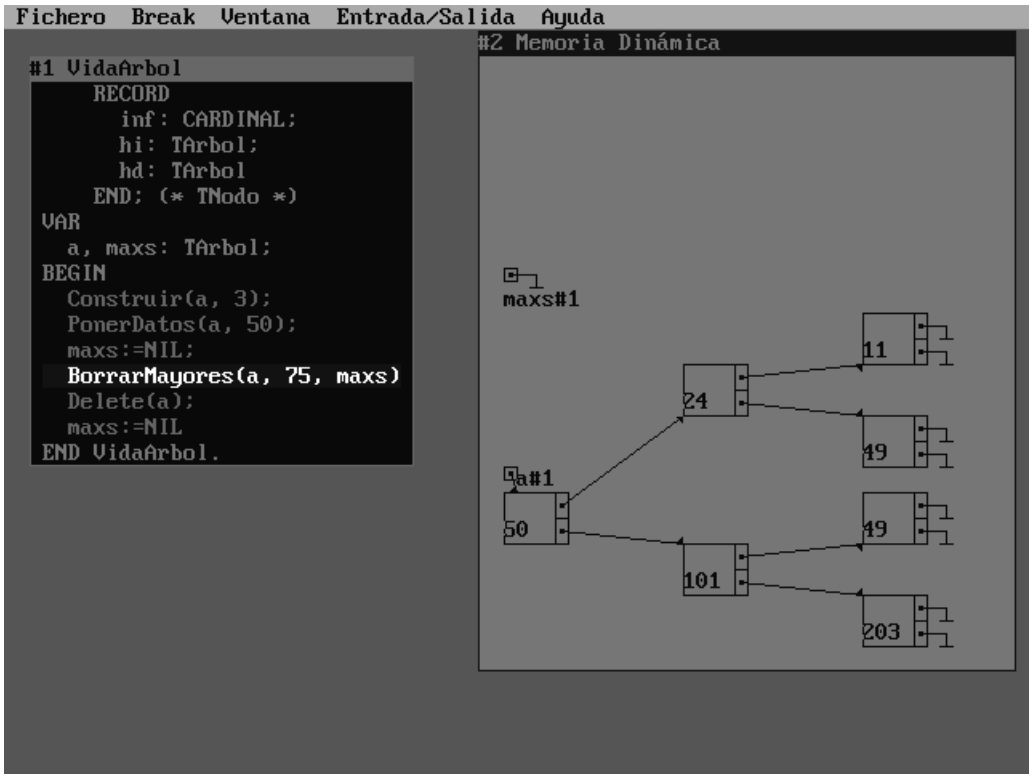


Fig. 5: Binary Tree

Visualization of other elements

In addition to recursion and dynamic memory, there are other elements that need to be visualized: non-dynamic variables and input/output.

1. Non-dynamic variables.

Non-dynamic variables can be classified depending on their types: basic types and structured types. Basic types can be shown very easily just by displaying their values (123, 'c', -3.14159, etc.). Structured types present more difficulties, as there are two things to show: values and their position within the structure. The two main structured types in imperative programming are arrays and records. Both can be shown as tuples, but this can be misleading for beginners, as they will see values of different types represented in the same way. For arrays the notation used in lectures can be adequate (unidimensional or bidimensional grids, see Fig. X??NUM), but the problem comes with arrays of more than two dimensions, that cannot be visualized on the screen in a practical way. A possible approach is to show the first two dimensions graphically and then show more dimensions, by selecting a particular position of the array. An alternative it is to show just the first simple type field of the structured type, and allow the user to choose the displayed field. Array indexes must label array elements, so students can easily find a particular element by just clicking on it.

Remarks made for arrays are applicable to records, showing field names instead of indices. In general, large structured types cannot be shown properly on screens, fisheye views can help to show large structures.

2. Input/output

Standard input and output is one of the elements worst visualized in commercial programming environments. The only visualization of input and output is the current state of the application screen. Input characters still unread appear on this screen, input and output is mixed, etc. The environment must distinguish exactly between input and output to be helpful to the student in his/her first steps, and it must indicate clearly how many characters from the input have been read.

There are two ways of distinguishing between input and output (in VisMod [JPP98,JP96] both are available). The first one is to use different colors for input characters than for output characters, but display them interleaved in the same window (see Fig. 8). This approach has the advantage that the students can see how the application screen will be, but they are able to distinguish input from output. The second method is to split the input/output window into two windows, one for input and other for output (see Fig. 7). The latter method is useful at the very beginning when the student has difficulties to distinguish among them. However it is useful to be able to change from one mode to the other (as it is the case with VisMod).

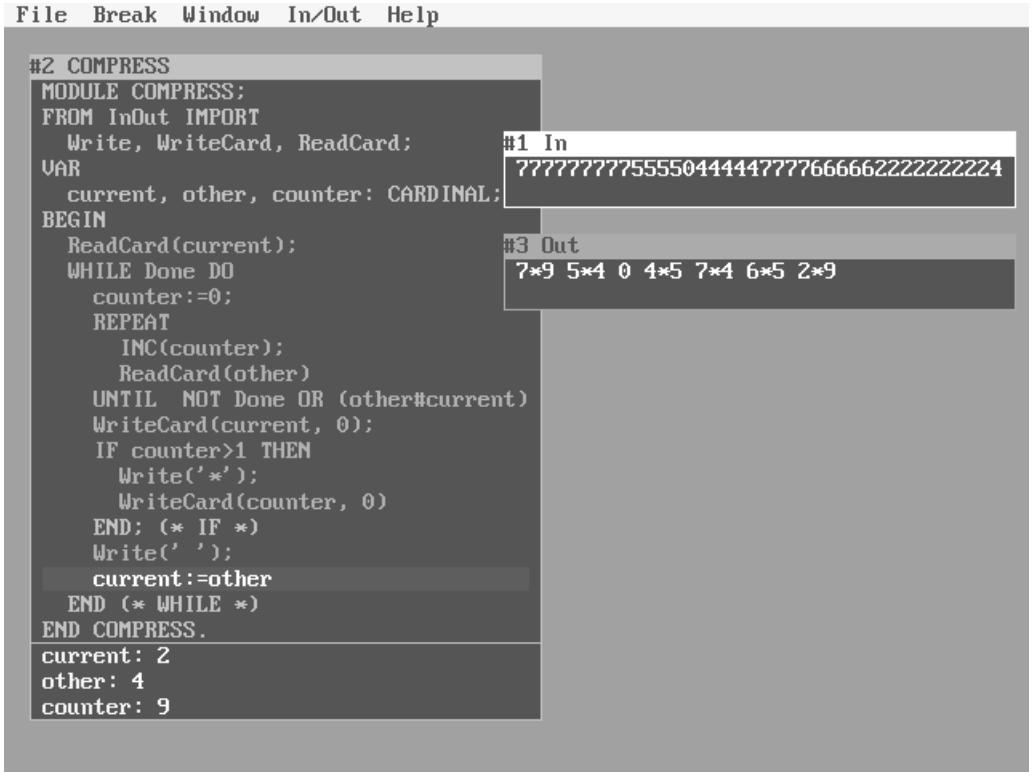


Fig. 7: Run-Length Encoding. Splitted Input/Output Window.

Another important element to visualize is to indicate how many characters from the input have been consumed, so the student will be able to track a read error. This can be achieved by highlighting the consumed characters.

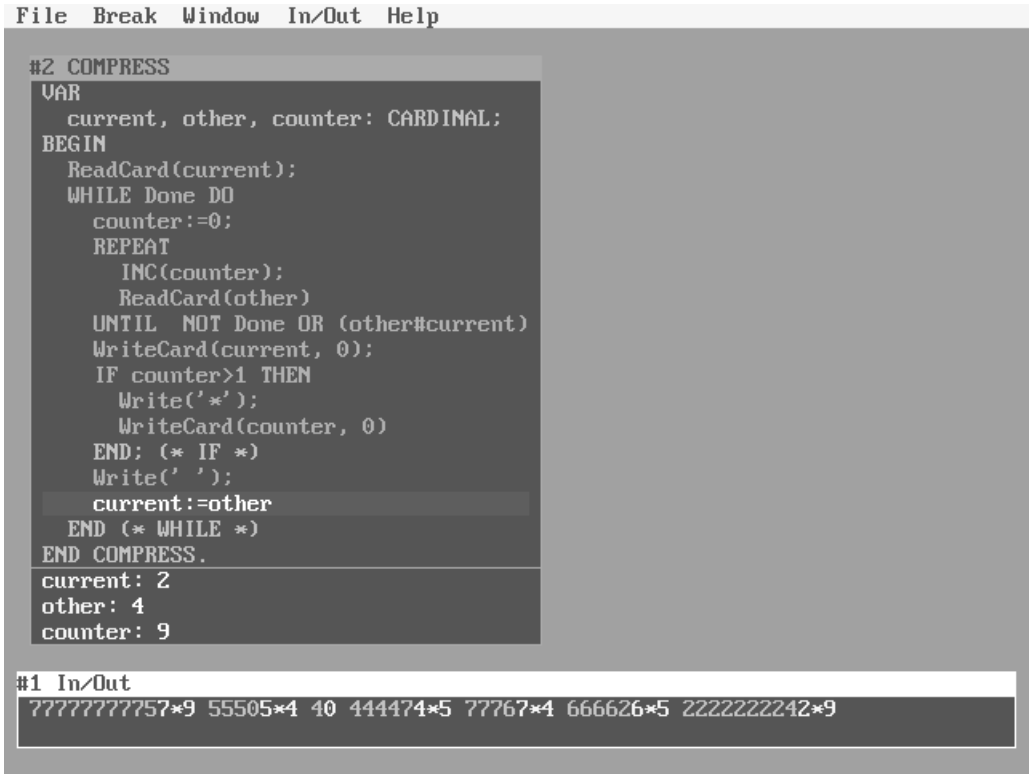


Fig. 8: Run-Length Encoding (Single Input/Output Window)

7.2 Visualization in Functional Programming

There are a number of issues that have been considered for visualization in functional programs. With respect to visualization of static aspects, textual pretty-printing is the most usual mechanism for visualizing programs or expressions in an understandable way. There are also some experiences in editing and displaying functional programs graphically [R+88]. Usually, function applications are represented as boxes, with a label identifying the function and a set of input and output arrows that represent argument and results respectively. Although this can be nice for novices, text seems to be a more adequate representation than graphics for most elements of functional programs. This is especially true in modern functional languages [Hud89], where a large number of syntactic elements have been included to provide a more powerful and friendly set of constructs: infix operators, partial application, symbolic identifiers, etc.

More interesting applications of graphics are in relation to some of the topics dealt in the visualization of imperative programs. However, these visualizations are much simpler then in imperative programming, since elements such as parameter modes, control stack, etc. do not exist in pure functional languages. Anyway, visualizations at a more conceptual level still are necessary. Thus, recursion in functional programs can be understood easily; as an example, see the progress of the recursive evaluation of the factorial function in the debugging section. However, general mechanisms such as displaying the recursion tree still are very useful for complex functions, such as non-linear recursive ones.

Other field where visualizations can be made at a conceptual level are data structures. Lists are represented with a standard list notation, e.g. `[1,2,3]`. However, other structures as trees can be understood much better with drawing than with text. Therefore, a mixed visualization of text and graphics can be successfully used for these structures [J+96].

The following figure shows one of these mixed visualizations. It corresponds to the evaluation of an expression to reverse (to mirror) the nodes in a binary tree. The program follows:

```
data tree(anything) == Empty
                    ++ Node(tree(anything)#anything#tree(anything));

dec mirror : tree(alpha) -> tree(alpha);
```

```
--- mirror (Empty)          <= Empty;
--- mirror (Node(ls,x,rs)) <= Node(mirror(rs),x,mirror(ls));
```

The figure illustrates the moment when the root element has been kept as the root of the result and the two subtrees are going to be mirrored.

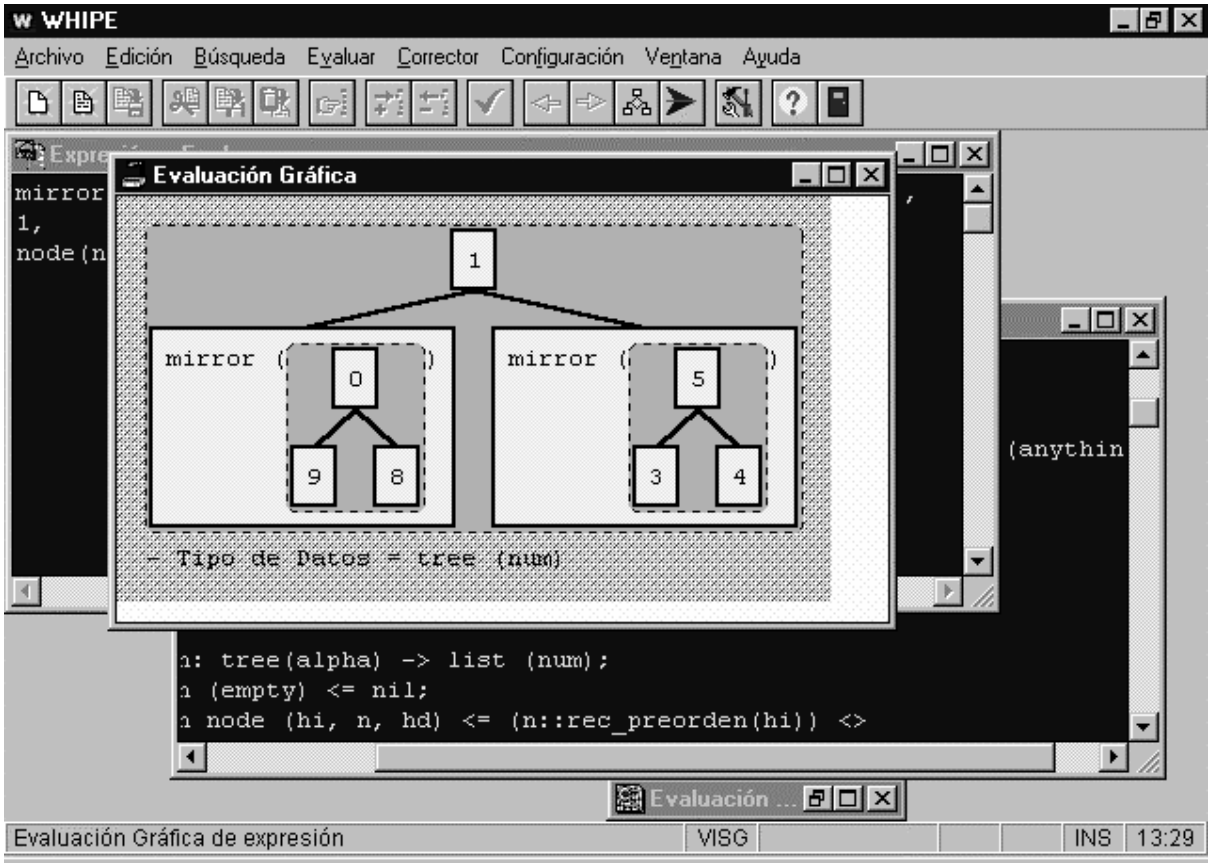


Fig. 9: Mirror of a tree

Even with graphical visualizations there problems for understanding complex programs. An important matter is scalability of graphics, i.e. scalability of the visualization of large expressions. Some trials have been made by applying different kinds of filters, so that expressions are displayed in a simplified way. We can highlight fisheye views [Vel98a] as a filtering technique to remove unimportant parts of an expression. These views suppress those parts in an expression which are far away in the abstract syntax tree of an expression from either the root or the redex of the expression. Suppressed parts are represented by elision '...'. The student must only set up a threshold that expresses the amount of syntactic nesting to show around the root and the redex. In the following figure, the redex is the left subtree, so the direct descendants of the right subtree have been elided.

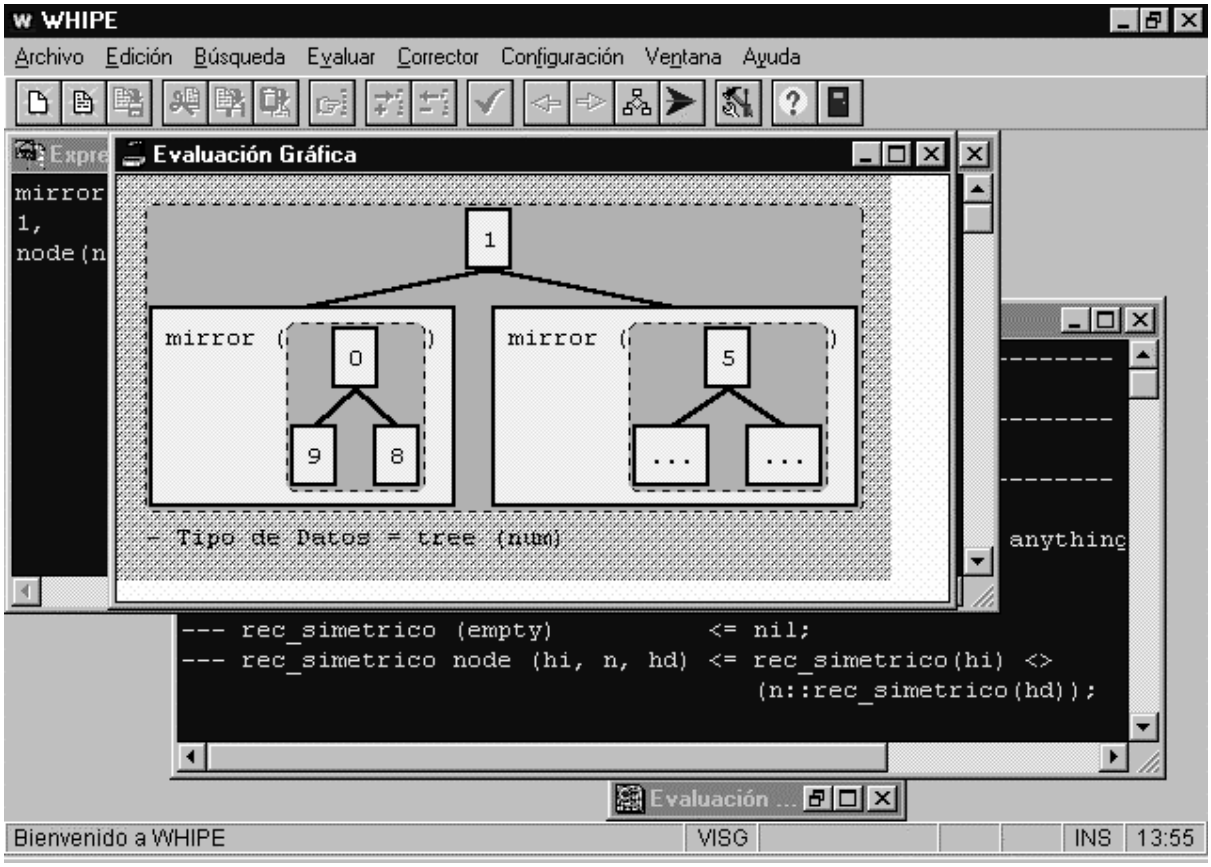


Fig. 10: Mirror of a tree

The previous visualizations show different parts of functional programs in a static way. However, dynamic aspects of expression evaluation can also benefit from visualization. Again, the most basic visualization aids consist in highlighting certain parts of the rewriting process. Hary et al. [HCD88] use several colors to highlight different pieces of information: the determination of the next redex, or the context where a computation is being made (e.g. a branch in a conditional expression).

The first task in a rewriting step is to look up the redex. A useful aid to understand properly this situation is to highlight the current redex, as made above, but it is more didactic to show dynamically the process of determining it [HDC93]. Some other useful aids can also be given about how a rewriting step is performed. For instance, to inform about failures and successes in pattern matching -with its corresponding variable bindings- is especially useful with lazy evaluation [AR94].

An intermediate expression during the evaluation of `fact(4)` is:

```
4*(if 3=0 then 1 else 3*fact(3-1))
```

where number four will be multiplied by the result of completing the computation of the factorial of three. The redex is `3=0`. A graphical explanation of the determination of the redex will show the whole expression and the dynamic evolution of a tuning process from the whole expression through subexpressions until the redex is focused. We can represent this process in several lines, where the bold subexpression is the current candidate to redex; in the final expression, the marked expression is the true redex.

`4*(if 3=0 then 1 else 3*fact(3-1))`

`4*(`**`if 3=0 then 1 else 3*fact(3-1)`**`)`

`4*(if` **`3=0`** `then 1 else 3*fact(3-1))`

More elaborate presentations display graphically the advance of evaluation. It is common to show the evolution of rewriting by means of figures. Although there are a variety of formats, they usually reflect temporal evolution in a top-down diagram, highlighting subexpressions by means of indentation or nesting within their enclosing expressions [AR94, TL92]. Finally, as evaluation goes on, information about previous steps can be kept [TL92, Vel94] or removed [AR94, F+97, HDC88]. It seems to be more adequate for novice students to keep the evaluation history, because in this way they can study it carefully.

Another set of important visualizing features is related to error information. Some errors can be visualized statically, mainly those related to static semantics. Syntax and type errors must identify clearly the text source where the error was detected, with meaningful messages. Showing static dependencies that produce these errors are a fortunate way of explaining them [F+97]. Some dynamic errors can also be detected using advanced program analysis tools, such as the one used in DrScheme, where it is deduced the set of possible values that a primitive is to be applied [F+97].

8 Conclusions

Educational PETs have different requirements than professional ones. These requirements are leading their current evolution, and they will continue influencing it in the future.

Some of these improvements come from new technologies. In particular, the most influencing one on PETs is visualization. It provides a more informative and intuitive view of different issues involved in programming (recursion, data types, etc.) than more classical presentations. In addition, visualization aids are an important part of any other improvement in a PET.

Good explanations given to students are an important part of PETs, mainly in relation to errors.

Educational PETs must be consistent with concepts introduced in programming courses and, therefore, with mental models of students. Higher levels of abstraction are mainly reflected in debugging. Mechanisms involved in program execution must be adequately shown, particularized to the programming paradigm.

References

[AR85] J. R. Anderson and B. J. Reiser, "The Lisp Tutor", *Byte*, April 1985, pp. 159-175.

[AR94] M. Auguston and J. Reinfelds, "A visual Miranda machine", *Software Education Conference (SRIG-ET'94)*, IEEE Computer Society Press, pp. 198-203.

[Boroni96] C. M. Boroni, T. J. Eneboe, F. W. Goosey, J. A. Ross, R. J. Ross, "Dancing with DynaLab: Endearing the science of computing to students", *SIGCSE'96*, pp. 135-139, March 1996.

[Bro87] F. P. Brooks, Jr., "No silver bullet: Essence and accidents of software engineering", *Computer*, vol. 20, n. 4, pp. 10-19, April 1987.

[CACM89] Several authors, "A debate on teaching computing science", *Communications of the ACM*, vol. 32, n. 12, pp. 1397-1414, December 1989.

[CACM92] *Communications of the ACM*, monographic issue on *CASE in the 90's*, vol. 35, n. 4, April 1992.

[Den88] P. J. Denning et al., *Computing as a Discipline*, ACM Press, 1988.

[DiBattista94] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis, "Algorithms for drawing graphs: An annotated bibliography", *Computational Geometry: Theory and Applications*, vol. 4, n. 5, pp. 235-282. 1994.

[F+97] R. B. Findler, C. Flanagan, M. Flatt, S. Krishnamurthi and M. Felleisen, "DrScheme: A pedagogic programming environment for Scheme", *Programming Languages: Implementations, Logics, and Programs*, LNCS 1292, Springer-Verlag, 1997, pp. 369-388.

[GL96] M. J. Granger and J. C. Little, "Integrating CASE tools into the CS/CIS curriculum", *Integrating Technology into Computer Science Education (ITiCSE'96)*, pp. 130-132

[Gor94] P. Gorman, "CASE tool usage for the educational environment", *Proceedings of the Software Education Conference (SRIG-ET'94)*, IEEE Computer Society Press, pp. 359-360

[HCD95]J. M. Hary, L. A. Cohan and M. J. Darnell, "User's preferences among different techniques for displaying the evaluation of Lisp functions in an interactive debugger", *Human Factors in Computing Systems (CHI'88)*, ACM Press, pp. 45-50.

[HS90] E. Horowitz y S. Sahni, *Fundamentals of Data Structures in Pascal*, 3rd ed., Computer Science Press, 1990

[Hud89] P. Hudak, "Conception, evolution and application of functional programming languages", *ACM Computing Surveys*, vol. 21, n. 3, pp. 354-411, September 1989.

[IEEE88] *IEEE Software*, monograph on *CASE Technologies*, vol. 5, n. 2, March 1988.

[JP96] R. Jiménez-Peris and M. Patiño-Martínez, "A Modula-2 interpreter/visualizer", ACM *SIGCSE/SIGCUE Conference on Integrating Technology into Computer Science Education (ITiCSE'96)*, ACM Press, p. 232.

[JPP98] R. Jiménez-Peris, M. Patiño-Martínez, and J. Pacios-Martínez, "VisMod: A novice-friendly visual debugger and programming environment", in preparation.

[J+96] R. Jiménez-Peris, C. Pareja-Flores, M. Patiño-Martínez and J. A. Velázquez-Iturbide, "Graphical visualization of the evaluation of functional programs", ACM *SIGCSE/SIGCUE Conference on Integrating Technology into Computer Science Education (ITiCSE'96)*, ACM Press, pp. 36-38.

[J+97] R. Jiménez-Peris, C. Pareja-Flores, M. Patiño-Martínez, and J. A. Velázquez-Iturbide, "The locker metaphor to teach dynamic memory", *SIGCSE'97*, pp. 169-173, San Jose, Feb. 1997.

[KP84] B. W. Kernighan y R. Pike, *The UNIX Programming Environment*, Prentice-Hall, 1984.

[Pap80] S. Papert, *Mindstorms: Children, Computers and Powerful Ideas*, Basic Books, 1980.

[Per89 N. Perry, N. Perry, *Hope*$^{+}$, Technical Report IC/FPR/LANG/2.5.1/7, Dept. of Computing, Imperial College, University of London, October 1989

[Reingold81] E. Reingold and J. Tilford, "Tidier drawing of trees", *IEEE Transactions on Software Engineering*, vol. 7, n. 2, pp. 223-228. 1981.

[R+88] B. J. Reiser, P. Friedmann, J. Gevins, D. Y. Kimberg, M. Ranney and A. Romero, "A graphical programming language for an intelligent Lisp tutor", *Human Factors in Computing Systems (CHI'88)*, pp. 39-44.

[San78] E. Sandewall, "Programming in an interactive environment: The 'LISP' experience", *ACM Computing Surveys*, vol. 10, n. 1, pp. 35-71, March 1978.

[Sne] J. L. A. van de Snepscheut, "Mechanized support for stepwise refinement", Technical Report 187-0, California Institute of Technology.

[TL92] D. S. Touretzky and P. Lee, "Visualizing evaluation in applicative languages", *Communications of the ACM*, vol. 35, nº. 10, October 1992, pp. 49-59.

[TR81] T. Teitelbaum and T. Reps, "The Cornell Program Synthesizer: A syntax-directed programming environment", *Communications of the ACM*, vol. 24, n. 9, pp. 563-573, Sept. 1981.

[Vaucher80] J. Vaucher, "Pretty printing of trees", *Software Practice and Experience*, vol. 10, n. 7, pp. 553-561. 1980.

[VCL96] J. A. Velázquez Iturbide, A. Mª. Cosías Rincón and N. Llorente Fernández, "An inexpensive programming toolset for algorithm development", *SIGCSE/SIGCUE Conference on Integrating Technology into Computer Science Education (ITiCSE'96)*, ACM Press, p. 231

[Vel94] J. A. Velázquez Iturbide, "Improving functional programming environments for education", in *Man-Machine Communication for Educational Systems Design*, M. D. Brouwer-Janse y T. L. Harrington (eds.), Springer-Verlag, 1994, pp. 325-332.

[Vel98a] J. A. Velázquez Iturbide, "Automatic simplification of the visualization of functional expressions by means of fisheye views", *Joint Conference on Declarative Programming (APPIA-GULP-PRODE 1998)*, July 1998. Also available as a technical report at ftp://ftp.escet.urjc.es/publications/tr98-10.ps

[Vel98b]J. A. Velázquez Iturbide, WinHIPE, available at ftp://ftp.escet.urjc.es/programacion/WinHIPE

[Webber96] Adam Brooks Webber, "The Pascal Trainer", *27th Technical Symposium on Computer Science Education*, 1996, pp. 261-265.

[Wetherell79] C. Wetherell and A. Shannon, "Tidy drawing of trees", *IEEE Transactions on Software Engineering*, vol. 5, n. 5, pp. 514-520. 1979.