A system for program visualization and problem-solving path assessment of novice programmers

Maria Satratzemi*, Vassilios Dagdilelis+, Georgios Evangelidis*
*Department of Applied Informatics, +Department of Educational and Social Policy
University of Macedonia, GR-54006 Thessaloniki, Greece
{ maya, dagdil, gevan}@uom.gr

Abstract

This paper describes an educational programming environment, called AnimPascal. AnimPascal is a program animator that incorporates the ability to record problem-solving paths followed by students. The aim of AnimPascal is to help students understand the phases of developing, verifying, debugging, and executing a program. Also, by recording the different versions of student programs, it can help teachers discover student conceptions about programming. In this paper we describe how our system works and present some empirical results concerning student conceptions when trying to solve a problem of algorithmic or programming nature. Finally, we present our plans for further extensions to our software.

1 Introduction

Over the past decades a great number of research studies have been conducted on the comprehension difficulties students meet when they are taught introductory programming concepts [3], [6], [7]. These studies came upon important factors that can negatively affect the process of learning programming, and explain why existing languages, programming environments, and teaching methods are effective or not.

Some of the reported difficulties met by novice programmers are summarized below [3], [6]:

- Difficulties arising from the way novice programmers perceive the general properties of the "notional" machine (the one they learn to control) and its relationship with the physical machine (the computer).
- Difficulties attributed to the syntax and semantics of programming languages that are supposed to constitute an extension of the properties and the behavior of the notional machine.
- The need to understand the established programming structures.
- The need to learn how to design, develop, verify, and debug a program when given certain tools.
- The fact that program editors, compilers, and debuggers are usually designed to be used by professional programmers. So, they pose extra burdens to novice programmers.
- Programming environments are not capable of offering visualization of program execution. So, the details of program execution remain "hidden" and students tend to perceive it as something that has to do with data input and data output. The lack of visual feedback makes the understanding of language semantics hard.

The results of these studies and the advancement of technology led to the development of programming environments that support the effective teaching of programming. Such environments include systems with compilers with enhanced diagnostic capabilities, and systems that focus on program animation. The most representative tools of these types of environments are THETIS [7] and DYNALAB [2]. THETIS provides improved error reporting, strengthened syntactic restrictions, run-time error detection and debugging and visualization tools. DYNALAB is a program animator.

Based on the above findings we developed an educational programming environment, called AnimPascal[1]. It is a program animator [9] and the characteristic that differentiates it from similar systems is its build in capability of recording student actions (recordability). We consider these actions as steps in the problem-solving paths followed by students, i.e. as important data to understand the particular way each student follows to solve a problem. Thus, AnimPascal has a dual goal: (i) to help novice programmers develop, verify, debug, and execute their program, and, (ii) to help teachers detect misconceptions of their students about programming.

We believe that knowledge of the path followed by students to the solution of a problem is, usually, extremely valuable information to someone wishing to explore student conceptions about programming and problem solving techniques. The capability to systematically record such paths can open up interesting new possibilities for exploring the conceptions of students. Our educational programming environment systematically records the actions of students, thus offering teachers with invaluable information about the path to the solution followed by the students, the steps backward, the repeated tries, the mistakes, and the

[1] This research is being funded by the EU and the Greek Ministry of Education.

hesitations. We designed an educational programming environment that records and stores what is didactically essential.

2 An overview of AnimPascal

AnimPascal has the following features:

- Ability to edit and compile standard Pascal programs.
- Dynamic visualization of program execution.
- Recording of different versions of user programs and associated compilation outputs.

The software has a simple and functional GUI. The main window consists of six components: menu, toolbar, and four areas (*Source Area, Program Output, Display Variables, Compiler Output*) as shown in Figure 1.

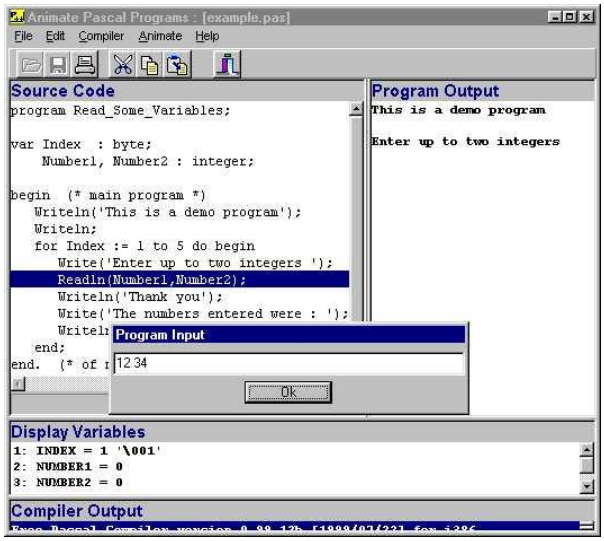

Figure 1 - Main Window and Program Visualization

It is interesting to explain the way AnimPascal informs students about errors, and warnings, hints, or notes regarding their program. In the case of errors the background of the appropriate line in the *Compiler Output* area becomes red, while in the other cases it becomes green. In all cases the font color becomes white. Students can interact with the software by clicking a red or green line and the cursor in the *Source Code* area moves to the appropriate source code line. That is, the *Compiler Output* area offers a dynamic visualization of the compilation output.

Every time students recompile their program the system automatically records the new version of the source code and the corresponding compiler output. The option *History* of menu item *Compile* presents the above information in the format shown in Figure 2.

The *History of Compilations* window can help teachers realize common mistakes and misconceptions of their students, and address them in their lectures.

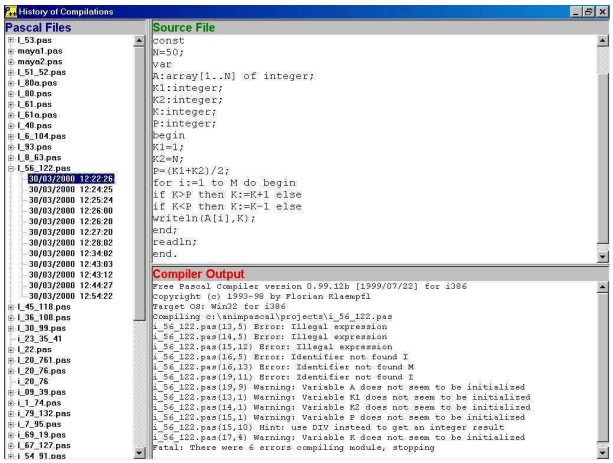

Figure 2 - Compilation History

The options of menu item *Animate* allow the dynamic visualization of program execution. The current source statement is highlighted and the result of its execution affects one the following areas:

- *Display Variables:* program variables have their values updated whenever they are assigned new values, that is, this area visualizes program memory.
- *Program Output:* whenever an output producing statement is executed, the output is appended here.
- *Program Input:* input statements cause the appearance of a window where students can type an input value that is assigned to a program variable.

AnimPascal was implemented in C++ for the MS Windows 95/98/ME platform. We used compiler Free Pascal [4] and debugger gdb [10], both under the GNU Public License. Free Pascal was used because of its compatibility with Borland Turbo Pascal, the most popular compiler for teaching Pascal. Free Pascal offers more detailed compiler output than Turbo Pascal and helpful hints for both novice and advanced programmers. It also allows the porting of AnimPascal to various platforms since it is available for many operating systems.

3 Using AnimPascal

We will briefly describe our findings from a typical laboratory class where first year students used AnimPascal.

Students were asked to implement the classic binary search algorithm, which was chosen because it is a well-known algorithm to the computer science community for its deceptive simplicity. While the algorithm appears to be simple, there are certain peculiarities one has to consider when trying to program it and verify its correctness [1]. Lesuisse [8] showed that even published versions of the algorithm contain errors, weaknesses, and special cases (for example they require the number of elements to be a power of 2).

The algorithm had been taught in the class a month and a half before the laboratory. The laboratory intended to:

- Evaluate the programming environment.
- Record student conceptions and verify the difficulties reported in the literature regarding the implementation of the binary search algorithm.
- Record the paths to the solution (or a solution) followed by students during program design, development, verification, and debugging.

After assessing the recorded student input we conclude the following:

(i) The warning messages and hints provided by AnimPascal helped students comprehend the real meaning of certain program constructs. For example, students ended up using expression (2) instead of expression (1), thus correcting the incompatible operands syntax error reported by Pascal (AnimPascal hints on using DIV instead of /).

```
Midpoint := (LowLimit+HighLimit) / 2     (1)
Midpoint := (LowLimit+HighLimit) div 2   (2)
```

So, students focused on program development and testing instead of trying to correct expression (1) based on the hypothesis that it contained a logical error.

(ii) Many students made semantic mistakes confusing array positions with array elements, i.e., i with A[i], or logical mistakes miscalculating the limits of the sub-array where the target element would lie if it existed at all. These are quite common mistakes when students use arrays.

(iii) Students usually fail to consider «extreme» cases, like arrays with no elements, or one element, or arrays that did not include the element in question.

(iv) On the contrary, many students tried to verify the correctness of their solution once it was syntactically correct. They usually attempted to avoid infinite loops by modifying the termination condition. In those cases they used a termination condition that was quite different than the one they had been taught in the class. It is clear that they were trying to test their code in all the cases they considered possible to happen.

(v) Most of the students developed algorithms with infinite loops. This happened because they replaced one of the limits with the midpoint assuming (indirectly) that the space of the search gets shorter continually. This is not always true – for example when `HighLimit-LowLimit=1` in the code fragment below:

```
LowLimit:=1;
HighLimit:=n;
...
Midpoint:=(LowLimit+HighLimit) div 2;
if v <> A[Midpoint] then
   if v > A[Midpoint] then LowLimit:= Midpoint
   else HighLimit:= Midpoint;
```

Thus, we can conclude that, students do not systematically check their algorithms and base their reasoning on their intuitions.

(vi) Lesuisse [8] noticed that some categories of algorithm implementations have repeated sections. He considered that as a case of «poor» programming analysis and made the hypothesis that these programs were a result of «mental execution». We also got programs of this type, strongly confirming Lesuisse's hypothesis. Repetition of algorithm sections shows that, once students ascertain the correct sub-array, they partition it in two by repeating the same code.

The findings of AminPascal allowed us to get an overall idea for student errors that we summarize in the following table:

1	did nothing at all	1,4%
2	solved it correctly	5,8%
3	some syntactic errors	4,3%
4	split the initial array into two parts, then searched each part sequentially	4,3%
5	did not use a loop	4,3%
6	wrong termination condition (related to point 9)	17,4%
7	computed the middle element once for the initial array but not for each resulting sub array	1,4%
8	presumed that the element existed in the array	8,7%
9	wrong computation of the index of the middle element	7,2%
10	wrong computation of the boundaries of each sub array	10,1%
11	confused the index of the array holding the number with the number itself	10,1%
12	correct termination condition, but wrong display condition	1,4%
13	used the "for" statement but computed anew its index values hoping to reduce the size of the array	1,4%
14	difficulties in formulating the termination condition (used AND) that resulted in repeated modifications of the code	2,9%

Table 1 - Summary of student errors

Table 1 directed us to a more concise analysis of the paths to the solution followed by students. This analysis provided us with qualitative data for each student. Figure 3 shows a typical path followed by a student; the timeline can provide a better understanding of the path followed by a student.

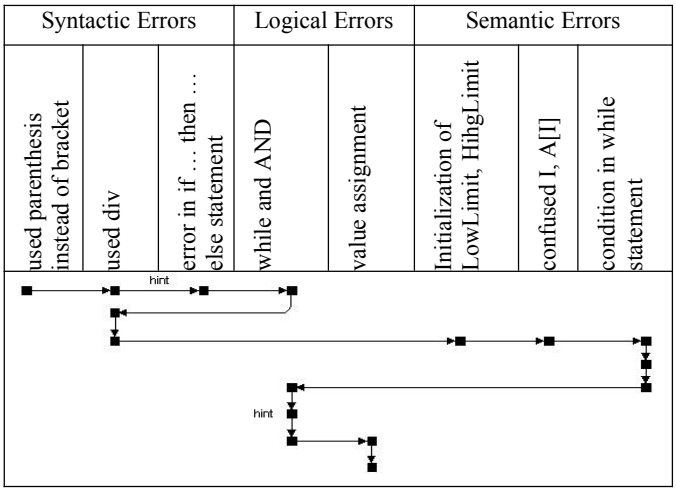

Figure 3 - Typical case of the path to a solution followed by a student and the corresponding timeline

The typical timeline presented in Figure 3 helps us establish the following:

The continuous movement of the timeline between the syntactic and logical errors clearly demonstrates the difficulties posed to the novice student programmer by the idiosyncratic nature of "real" programming languages [3], [6]. In particular, the repeated attempts of students to correct syntactic errors shows that students should be introduced to programming through languages with simple syntactic rules. Although the time spend in correcting syntactic errors is not shown in Figure 3, it is clear that programming language syntax can be a negative factor in problem solving.

Similarly, it is clear that the messages generated by the system can play an important role in the process of problem solving. In our example, the hint for using operator DIV is an effective help to the user, whereas the message about the array boundaries is incomprehensible. Students repeatedly attempt to correct their errors without understanding the meaning of the error message produced by the compiler. The above observation suggests that a systematic study of student errors and an investigation of the reasons students cannot take advantage of the compiler error messages could improve the produced error messages. Our empirical studies show that this could be achieved by using more detailed and/or translated in our language error messages.

A study of the timelines brings to light the parts of the algorithm students find hard to implement. For example, regardless of their final solution, almost all students had a hard time determining the boundaries of the sub-array to search next. The timelines indirectly indicate the way students think. It is clear that most students use simulation to develop their programs: they "run" their code in a "mental machine".

4 Conclusions

AnimPascal appears to be of great help to novice programmers. It can help them improve their ability to design, develop, verify, and debug programs. It can also offer teachers with invaluable information about the techniques used by students for program developing, verification and debugging.

The current version of AnimPascal records and reports the various versions of student programs and their corresponding compiler output. A very interesting feature, that we plan to add in a future version of our software, is the automated processing of the recorded information.

We plan to add the following two procedures:

- **Classification of the messages encountered in the compiler output**. We plan to use statistical analysis to automatically detect the most common student mistakes, the category they belong to, and the time students spend to correct them.
- **Discovery of the differences between two successive versions of a given program**. It is important to have a visual representation of the differences between two successive versions both of the student source code and the corresponding compiler generated output. A possible implementation could use a tool like diff [5]. One could then draw conclusions about the degree of comprehension of the compiler output by students, the way it influences problem solving, and student conceptions about it.

References

[1] Bentley J., Programming Pearls, Addison-Wesley, 1986.

[2] Birch, M., Boroni, C., Goosey, F., Patton, S., Poole, D., Pratt, C., Ross, R., DYNALAB: A Dynamic Computer Science Laboratory Infrastructure Featuring Program Animation, In 26th SICGSE Technical Symposium on Computer Science Education (1995), *SICGSE Bulletin*, vol. 27, pp. 29-33.

[3] Brusilovski, P., Calabrese, E., Hvorecky, J., Kouchnirenko, A. & Miller P., Mini-languages: a way to learn programming principles, *Education and Information Technologies, 2*, pp. 65-83.

[4] Canneyt M. V., Klämpfl F., Free Pascal: Users' Manual [1999]. Available Online: http://www.brain.uni-freiburg.de/~klaus/fpc/docs.html

[5] Diff, Available Online: http://www.cslab.vt.edu/manuals/diff/diff_toc.html

[6] du Boulay, B., Some Difficulties Of Learning To Program, In Studying The Novice Programmer, Soloway, E., Sprohrer, J. (Eds.), Lawrence Erlbaum Associates, 1989, pp. 283-300.

[7] Freund, S. N. & Roberts, E. S., THETIS: An ANSI C programming environment designed for introductory use, ACM SIGSCE 1996, Philadelphia, PA USA, pp. 300-304.

[8] Lesuisse R., Some Lessons Drawn from the History of the Binary Search Algorithm, *The Computer Journal*, *26*(2), 1983, pp. 154-163.

[9] Price, B., Baecker, R. & Small, I., An Introduction to Software Visualization, In Software Visualization: Programming as a multimedia Experience, In Stasko, J., Domingue, J., Brown, M. and Price, B., (Eds.), MIT Press, Cambridge, 1997, pp. 3-28.

[10] Stallman, R., and Cygnus Support, Debugging with GDB, Free Software Foundation, Boston, 1995.