X-Compiler: Yet Another Integrated Novice Programming Environment

Georgios Evangelidis[*], Vassilios Dagdilelis[**], Maria Satratzemi[*], Vassilios Efopoulos[*]
[(*)]*Department of Applied Informatics,* [(**)]*Department of Educational and Social Policy*
University of Macedonia, Thessaloniki, Greece
{gevan, dagdil, maya, efop}@uom.gr

Abstract

This paper presents a simple programming language, called X, and an educational programming environment, called X-Compiler, designed to introduce students to programming. X-Compiler can be used to edit, compile, debug and run programs written in X, a subset of Pascal. X-Compiler could be didactically interesting because of the following features: (a) users can watch the intermediate steps of the execution of a program: source code compilation, correspondence of source and pseudo-assembly code during execution, registers content, and intermediate values of user and temporary system variables; also, they can edit the produced pseudo-assembly code and re-execute it, (b) there are many detailed and explanatory messages that can guide novice programmers when debugging their programs and, in general, help them write better programs.

1. Introduction

For many years now, the most common methodology [4] for teaching principles of programming languages is based on the use of a general purpose programming language, like C or Pascal, and a commercial programming environment. But, this approach does not appear to be didactically effective, since novice programmers are expected to become familiar at the same time with many concepts related both to the structure and operation of information systems and the programming techniques. Du Boulay [2], for example, mentions the following general factors that hinder the learning process:

1. The way students understand and control the "mental" machine that executes their programs and its relationship with the actual machine (the hardware).
2. The rules of the programming language (the syntax and semantics of the language affect and/or extend the behavior of the mental machine).
3. The need to first comprehend the language control structures.
4. The need to master a problem solving technique (students have to design, implement, test, and debug a program using a predefined set of tools).

Below, we further elaborate on the factors mentioned above:

- Program execution is a kind of mechanism that accomplishes a certain task. It may be very hard for students to distinguish between the program and the mechanism it describes.
- A computer (hardware) and a programming environment (software) comprise together a mechanism that is used to create other mechanisms, that is, programs. Students should be aware of the way such a computing system operates, i.e., what exactly happens in the internal of a computing system, or they may develop their own theories of operation that are usually insufficient and erroneous.
- In most programming environments, it is usually hard for students to understand the information presented on the computer screen. This information may refer to a previous interaction between the student and the programming environment, to the input the student has just entered, or to the output of the execution of some code.
- Students have to master procedures regarding the editing, loading and saving of programs.
- The internals of the "mental" machine are usually hidden. Students cannot "see" what is happening during the compilation and/or execution of their code.
- The code produced by students has to adhere to strict syntactic and semantic rules or it cannot be "understood" by the system. This is a source of frustration for students because they usually tend to lend human characteristics to the system.
- The use of English words as keywords is another potential source of anomaly because it lends the system a kind of intelligence. Also, these words have different meanings when used in programs and when in natural language.
- Most programming environments do not provide adequate and user-friendly error handling and reporting [7].

Considering the above-mentioned problems, we believe that it is essential to develop alternative didactic methodologies for introducing students to programming. This can be aided by the use of specially designed programming environments. In this paper we introduce a

programming language we have designed and we call X, and its corresponding programming environment. In Section 2, we present the basic design principles we used when designing our programming environment (X-Compiler). The specifications of the languages (X and pseudo-assembly) are given in Section 3. In Section 4 we describe the programming environment. Finally, Section 5 gives an insight on the didactic use of X-Compiler. We conclude with a summary.

2. Design Principles

Programming environments for novice programmers should be effective tools for achieving certain didactic goals. Below, we list a number of principles [12] we considered essential while developing the programming environment for X.

Minimalism. The programming language should be as simple as possible. We avoided the use of types and we don't require variables to be declared before use. Also, the programming environment does not present unnecessary information.

Simplicity. Novice programmers are asked to program a mental machine they barely know and understand. This is a machine whose nature is determined, or better, implied by the programming language. It is essential that the mental machine is as simple as possible, i.e., it should consist of a small number of components that interact in a well-defined and clear manner [3, 9, 13].

Stepped execution and control through visual feedback. Instant feedback can help novice programmers implement and debug their programs. A graphical debugger is useful even for correct programs: it can help novice programmers understand the way their programs work. A programming environment should help novice programmers test, debug, and execute their programs [8]. It is essential that the programming environment include a low-level debugger and a code execution tracer together with data visualization [7, 10, 11, 14].

3. Languages X and pseudo-assembly

We have designed a Pascal-like language, called X. The language supports the **assignment**, **if ... then**, **while ... do**, **read**, **write** and **compound** statements. Identifiers and numbers are integers and all, possibly nested, arithmetic expressions evaluate to integers. X supports only three relational operators: **>, =,** and **<>**. A comment is text enclosed in curly brackets ({}). In Table 1 that follows you can find the full specification of X.

Table 1. Specification of X

program	BEGIN **{statement;}*** END.
statement	**id** := **expr** | READ **id** | WRITE **expr** | IF **rel_expr** THEN **statement** | WHILE **rel_expr** DO **statement** | BEGIN **{statement;}*** END
id	any string consisting of letters, digits and underscore and starts with a letter
expr	**id** | **number** | **expr op expr** | **(expr)**
op	+ | - | * | /
number	any long integer between
rel_expr	**expr rel_op expr**
rel_op	> | = | <>
comments	anything enclosed in curly brackets

The assembly language used is a pseudo-assembly that runs on a virtual machine with two registers and includes the basic LOAD, STORE, COMPARE, JUMP, ADD, etc., instructions needed to implement the source language. In Table 2 below you can find a detailed description of the instructions of the pseudo-assembly.

Table 2. The pseudo-assembly used in X-Compiler

instruction	Explanation
BLOCK v	**declare** an integer variable or a memory position with name v
LOAD(r, v)	**store** contents of memory location v in register r (r can be 0 or 1)
LOADN(r, n)	**store** number n in register r
STORE(r, v)	**store** contents of register r to memory location v
ADD_R(r1, r2, r3) **SUB_R(r1, r2, r3)** **MUL_R(r1, r2, r3)** **DIV_R(r1, r2, r3)**	**add/subtract/multiply/divide** contents of registers r1 and r2 and store the result in register r3
CMP(r1, r2, r3)	**compare** the contents of registers r1 and r2 and store the result in register r3; the result is –1 if r1 < r2, 1 if r1 > r2, and 0 if r1 = r2
INC(r)	**increment** the contents of register r
DEC(r)	**decrement** the contents of register r
NEG(r)	**negate** the contents of register r
my_label:	**declare** a label with the name my_label
JUMP_ZERO(r, lb)	**jump** to lb if contents of r = 0
JUMP_NEG(r, lb)	**jump** to lb if contents of r < 0
JUMP_POS(r, lb)	**jump** to lb if contents of r > 0
JUMP(lb)	**unconditionally jump** to lb
READ(r)	**store** user input to register r
WRITE(r)	**print** contents of register r to the output window

4. X-Compiler programming environment

The X-Compiler programming environment has been implemented on the Microsoft Windows platform using Macromedia Director 7 and the compiler construction tools LEX and YACC [1].

X-Compiler allows users to edit, debug, and execute their programs. It consists of five windows (1-source-code, 2-assembly-code, 3-system-registers&tempvars, 4-user-vars, 5-output), and has two modes of operation (novice and advanced) (see Figure 1). In the novice mode only windows 1 and 5 are active, whereas in the advanced mode all windows are active. Of course, users can activate or deactivate any window any time. The provided menu-bar and window-specific toolbars allow the intuitive use of the programming environment (open, save, and edit source or assembly code, compile, execute, or step-execute either type of code, arrange windows, get help on the operation of the programming environment or the X-language). Here, we should mention that double-clicking any keyword, operator, or delimiter on the source code window provides help on the specific feature of the X-language.

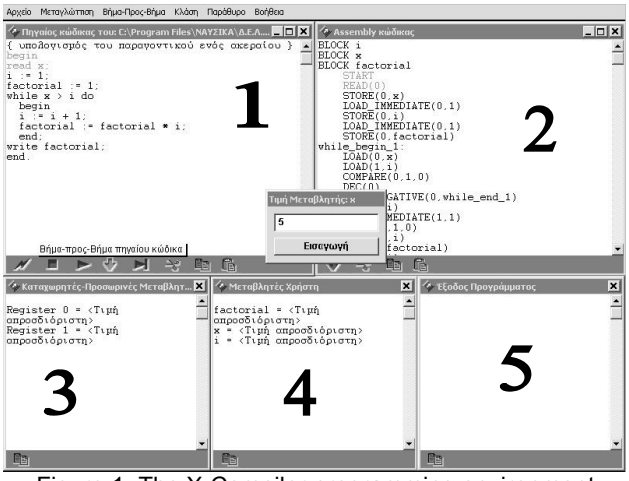

Figure 1. The X-Compiler programming environment

During compilation, syntactic errors in the source code trigger a pop-up window that contains two drop-down lists, one for the detected errors and one for the warnings issued by the compiler. Users can choose the list element they desire to get an explanation of the type of the error or warning. At the same time the appropriate line of the source code is highlighted.

Once users succeed in compiling their code they can either execute it or step-execute it so that they are able to examine what actually happens during execution. For each source code statement the corresponding assembly code statement(s) are highlighted and at the same time the appropriate system registers, temporary variables, and user variables get updated if necessary. Input statements are handled by using a pop-up window that allows users to enter the desired value for their integer variables (see in the center of Figure 1). The output window displays the output generated by the WRITE statements of the user programs.

An interesting feature of the assembly code window is the ability to edit/alter the compiler produced assembly code that corresponds to a given source code fragment and execute it. Since the compiler produces non-optimized assembly code this feature can allow teachers to guide their students in manually optimizing their assembly code. Alternatively, users can write their assembly programs from scratch.

5. Didactic features of the X- Compiler

The didactic objective of X-Compiler is to offer students a lightweight programming environment with simple high level and pseudo-assembly languages and clarify the phases of compilation and program execution that usually constitute a "black box" in professional programming environments [3, 4].

X-Compiler offers interesting didactic features. Users get detailed feedback on the errors encountered during compilation, and are always aware of everything that happens to the internals of the mental machine during program execution (by seeing the correspondence between source and assembly code, the intermediate values of the machine registers, the system generated temporary variables, their own variables, and the contents of the output window). Moreover, users can alter the produced assembly code and then execute it.

We provide teachers and students with the appropriate manuals that contain a series of educational activities on the use of X-Compiler. We have designed the included activities based on the findings of the research community and our teaching experience on the difficulties encountered by students that are novice programmers.

For example, the following case of “cognitive transfer” [5, 6] could be a potential source of difficulties for novice programmers trying to solve problems in a traditional programming environment. Some students may believe that the following code computes the area of a parallelogram:

```
area := base * height;
read(base);
read(height);
write(area);
```

They will be surprised to realize that **area** is not computed correctly. In the X-Compiler programming environment they can see why the above program is not correct by observing the intermediate values of their variables.

Now, consider the code fragment below that swaps the values of variables A and B.

```
TEMP:=A
A:=B;
B:=TEMP;
```

The teacher can observe that one can get the same effect without using the extra variable TEMP, as shown in the following code:

A:=A+B;
B:=A-B;
A:=A-B;

Students can examine the intermediate values of the variables and understand why this solution is correct. The teacher could then show that this solution is slower (because it uses more assembly instructions than the previous solution) and also it does not always work correctly (when we have integer addition underflow or overflow).

Those two examples demonstrate the didactic capabilities of our programming environment. Students can not only examine whether their programs produce the correct output, but also discover easily and fast the syntactic and semantic errors they make.

6. Summary

X-Compiler is already being used by the Greek Ministry of Education in a number of secondary education schools and in the entry level university courses on programming we teach (especially its assembly language features). Currently, we are in the process of implementing some additions to the software concerning, (a) a small extension of X to include strings the procedures, and (b) the creation of a "smart" advisor on the logical errors made by students (see first example with the calculation of the area of a parallelogram).

7. References

[1] A. V. Aho, R. Sethi, and J. D. Ullman, "Compilers: principles, techniques, tools", Addison-Wesley, 1988.

[2] B. Du Boulay, "Some Difficulties Of Learning To Program", *Studying The Novice Programmer*, E. Soloway and J. Sprohrer (Eds.), Lawrence Erlbaum Associates, 1989, pp. 283-300.

[3] B. Du Boulay, T. O'Shea, and J. Monk, "The Black Box Inside the Glass Box: Presenting Computing Concepts to Novices", *Studying The Novice Programmer*, E. Soloway and J. Sprohrer (Eds.), Lawrence Erlbaum Associates, 1989, pp. 431-446.

[4] P. Brusilovsky et al, "Mini-languages: a way to learn programming principle", *Education and Information Technologies*, 2, 1997, pp. 65-83.

[5] V. Dagdilelis, "Conceptions des eleves a propos des notions fontamentales de la programmation informatique en classe de Troisieme", Memoire D.E.A., Universite Joseph FOURIER, Grenoble, France, 1986.

[6] V. Dagdilelis, "La validation en programmation: a propos de conceptions des etudiants", actes V Ecole d'ete de Didactique des Mathematiques et de l'Informatique, Plestin-les-Greves, France, 1989.

[7] S. N. Freund and E. S. Roberts, "THETIS: An ANSI C programming environment designed for introductory use", *ACM SIGSCE '96*, Philadelphia, PA, USA, pp. 300-304, 1996.

[8] C. DiGiano, R. Baecker, and A. Marcus, "Software visualization for Debugging", *Communications of the ACM*, Vol. 40, No. 4, pp. 44-54, 1997.

[9] P. Mendelsohn, T.R.G. Green, P. Brna, "Programming Languages in Education: The Search for an Easy Start", *Psychology of Programming*, J. Hoc, T. Green, R. Samurcay, and D. Gilmore (Eds.), Academic Press, 175-200, 1990.

[10] S. Mukherjia and J. Stasko, "Applying Algorithm Animation Techniques for Program Tracing, Debugging, and Understanding", *IEEE 0279-5257/93*, pp. 456-465, 1993.

[11] S. Mukherjia and J. Stasko, "Toward Visual Debugging: Integrating Algorithm Animation Capabilities within a Source-Level Debugger", *ACM Transactions on Computer-Human Interaction*, Vol. 1, No. 3, pp. 215-244, 1994.

[12] J. F. Pane and B. A. Myers, "Usability Issues in the Design of Novice Programming Systems", *Technical Report CMU-CS-96-132*, School of Computer Science, Carnegie Mellon University, 1996.

[13] M. Ruckert and R. Halpern, "Educational C", *ACM SIGSCE Bulletin*, pp. 6-9, 1993.

[14] R. S. Sangwan, J. F. Korsh, and P. S. LaFollette, "A System for Program Visualization in the Classroom", *ACM SIGSCE '98*, Atlanta, GA, USA, pp. 272-276, 1998.