

Using an interpreter to teach introductory programming

M. Harman, S. Danicic

*School of Computing, University of North London,
Eden Grove, London, N7 8DB, UK*

Abstract

Inexperienced students are overwhelmed by the steep learning curve caused by the large number of programming tools and concepts that have to be mastered before even the simplest program can be executed. Such tools clearly enhance the productivity of more experienced programmers, but to a novice, they are merely treated as “something else to learn”.

We have noticed that a large number of potentially good programmers give up in these circumstances. We have addressed the problem by implementing our own interpreter for a subset of Modula-2. The Interpreter allows us to focus specifically upon the essentials of programming technique and allows students to experience the confidence boost of writing their first programs *before* requiring them to master more sophisticated programming environments.

After an initial period using the interpreter, students have overcome the initial “confidence” barrier and can migrate to the use of an integrated program development environment, where they appreciate the need for tool support and the unavoidable complexity it entails.

1 Introduction

We have spent four years teaching Modula-2 as a first language on a Post Graduate Diploma course at the University of North London. We use two books on the course, [4] and [5].

The post graduates are mature students who have had widely varying levels of previous experience in computing. As our course is a conversion course, we do not assume any prior knowledge of programming. However, many of the students on the course *do* have experience in computing and this results in a cohort with very diverse skills and expectations.

The main problem we have experienced is that of our novice programmers losing their confidence within the first few weeks of the course. This is a situation from which some students never recover. We found that it was extremely hard to find the correct level at which to pitch an introductory course in such a way as to motivate those more advanced students whilst, at the same time, allowing the less advanced students to keep up with the material and practical work.

284 Software Engineering in Higher Education

We started to look for an approach to teaching which would maintain the confidence and morale of students for whom the technology, concepts and jargon were unfamiliar and intimidating.

2 Teaching Programming Using a Compiler

Students get a tremendous feeling of satisfaction and achievement when they witness the successful execution of their very first program, particularly if this is a non-trivial program (one which, perhaps, works out their tax bill as opposed to merely printing “hello world” on the screen in front of them). Sadly, before students can experience this sense of achievement, they have to master a number of, rather non-trivial processes. Specifically, students will have had to have:

- Mastered the basics of the editor,
- Learnt the syntax and purpose of declarations,
- Understood the basic concept of a module.
(Using Modula-2 this is complicated by the need to import basic input and output procedures in order to perform elementary I/O),

- Learnt about the compilation process.

This will, at least, involve:

- The distinction between source and object code,
- The linking process,
- A basic understanding of files and directories.

In an attempt to circumvent these initial hurdles, we would tell students:

Don’t worry about linking, this is something that has to be done, but it’s not possible to explain it properly at this stage.

and, instead of explaining the purpose of declarations and importation of I/O procedures we would say:

You just have to type these things in every time you write a program.

They do have a purpose, but we’ll explain what it is later on in the course.

When teaching undergraduate students in the eighteen to twenty two year age range, these explanations seem to be perfectly adequate and the students progress quite happily. For the mature students on the post-graduate course these explanations were far from satisfactory. Many students felt that they were struggling to keep up with the material and yet they had the impression that what they were finding so hard was merely the “tip of the iceberg”.

This feeling of inadequacy was exacerbated by the more experienced students on the course, who *did* understand the parts of the process we were forced to gloss over, and immediately set about writing far more elaborate programs than were required for the initial exercises.

3 Teaching Programming Using an Interpreter

The problems associated with using a compiler to teach programming to mature novice programmers fall into two categories:

1. The steep technological learning curve,
2. The inappropriate nature of “burying the issues” of linking, declarations, importation and so on when presenting the information.

We decided that if we implemented an interpreter for a suitable subset of Modula-2, we should be able to address both these problems.

The first problem is solved by the technological simplicity of the interpretation mechanism. Instead of learning to use an editor, operating system, mouse and programming environment, students simply type a single command at the DOS prompt and they are “in Modula-2”. From then on the student is sitting at an abstract machine which only knows about Modula-2; the student does not need to know how to use an editor. Of course, we did require some non-Modula-2 commands to enable the student to load, save and print their programs, these are described in section 4.3.

The second problem was circumvented using the interpreter since there was no need to implement the syntax for declarations, imports etc. As the students did not encounter these issues they did not cause a problem.

Clearly, there is a price to be paid for simplicity: although the student is spared the necessity of an editor, they do have to *re-type* incorrect lines. This caused a great deal of frustration, *but it certainly did not destroy students’ confidence*. Moreover, when we subsequently moved on, introducing students to the compiler and the complexities it involved, they were in a far better position to appreciate the *motivation* for editing facilities, separate compilation, declaration and other features of the compilation system which would have been so daunting had they not negotiated the initial hurdle of writing their first program.

4 The Design of the Interpreter

The Modula-2 interpreter is written in (turbo) Pascal, using as few non-standard features as possible (some non-standard features were required for disk-access). We wrote our own tokeniser and recursive-descent parser using standard techniques [1, 2], rather than using parser-generating technology, as this makes the code easy to port from machine to machine. The source code for the interpreter occupies 110K (approximately 4500 lines of Pascal) and the compiled executable occupies 50K.

The current implementation runs on any IBM compatible PC. On typing the name of the executable, **MI**, the student is “in Modula-2” and can immediately type a command. For example

```
WriteString("Hello, world");
```

will produce the familiar message. The terminating semicolon is necessary because the LALR(1) parser maintains a buffer containing the next token for one symbol look ahead. Conceptually, we still think of the semicolon as a *separator*, but as the statements are entered in interactive mode, the semicolon denotes the fact that more statements are to come.

It is also possible to make assignments to variables, which remain active until overwritten or a program is executed.

286 Software Engineering in Higher Education

The sequence:

```
x:=1;  
y:=2;  
WriteInt(x+y,5);
```

produces the result 3 on the display.

4.1 Writing Programs Using the Interpreter

There is always a “current program” in memory whilst the interpreter is running. Initially this program is empty. To add to the program, the student uses line numbers. These line numbers have *no bearing* upon the program’s semantics (syntactically a line number has the same effect as a space character). Line numbers serve merely to allow code to be conveniently inserted and deleted.

For example, both

```
10 x := 1;
```

and

```
1 x  
2 :=  
3 1  
4 ;
```

are valid programs and assign 1 to **x** when executed.

4.2 The Subset of Modula-2

We anticipate that having used the interpreter for long enough to have mastered the fundamentals of control flow the students are confident enough to migrate to the full compiler implementation of the language. Thus the cut off point for our subset of the language comes with procedures. The interpreter allows integers, literal strings and arrays of integers (single dimension only). It provides all Modula-2’s loop and conditional constructs and the I/O procedures **ReadInt**, **WriteInt**, **WriteLn** and **WriteString** from the **InOut** library. An interpreter program is simply a sequence of statements.

4.3 Extra-Linguistic Features

In order to load and save programs and to log console sessions the interpreter provides a number of extra-linguistic features. When teaching, these can be introduced naturally as they are required, and are all written in lower case so that the student can distinguish between bona fide Modula-2 and the few non-Modula-2 commands required. Amongst these, the students’ favourite is **quit**, but we also provide **load<file_name>**, **save<file_name>**, **run**, **list**, **startlog** and **stoplog**. These last two start and stop console logging. Between the issue of a **startlog** command and the issue of a **stoplog** command, everything that appears on the screen is also copied to a time-date stamped log file.

We also provide a command **renumber n m**, which serves to alter the line numbering of the program from line *n* in steps of *m* lines. Both *n* and *m* are optional, their defaults both being 10.

We provide the students with the interpreter and a set of notes [3], specifically tailored for use with the interpreter.

5 Other Benefits of the Interpreter Approach

Having designed and implemented our interpreter for Modula-2, we became aware of a number of additional benefits that accrue from the use of our own “home-grown” system. We were able to exploit the fact that we had the source code of a simple small interpreted language available to tinker with.

5.1 Copyright Issues

Many of our students either had a computer at home or had access to a computer on which they would have been able to do much of their work. Sadly, using the compiler this option was closed, because these computers did not have the Modula-2 compiler. A number of students decided to purchase the compiler for their machines at home, and whilst this provided a solution for them, it often forced them to decide between books and software. In any case it was not a possible solution for those students who only had access to somebody else’s computer.

Since we had implemented the interpreter ourselves, we were able to give it away free to students to use. Also, as the interpreter is very small (50K), many students were able to persuade friends or colleagues at work to make space for it on their machines. This greatly increased the “hands on” time available to our novice programmers.

5.2 Reducing the Number of Syntax Errors

Using the compiler, students would frequently become overwhelmed by the number of syntax errors produced in the compilation phase. Many of these errors are, of course, “knock on” effects, producing an artificially high number of reported errors. This fact does not, however, appear to diminish the dismay experienced by a student who is told that a ten line program has produced thirty three errors!

Using the interpreter students are immediately alerted the moment they deviate from the syntactic rules of the language.

5.3 Tailoring the Error Message

As well as the sheer number of errors produced by the compiler during the parsing phase, we persistently found that students were mystified by the nature of the message itself.

For example the **WriteInt** procedure in the module **InOut** is used to display an integer value. It takes *two* parameters. The second is the amount of space in which the first parameter is to be displayed. Students often forget the second parameter, and this draws an error message such as:

Expected Parameters

This is not the fault of the compiler, which treats **WriteInt** in the same general manner in which it handles all procedures. Sadly, it makes no sense whatsoever to a novice programmer who has yet to meet procedural abstraction.

Using the interpreter we are able to produce more appropriate error messages because we know the psychology of our restricted range of users and because we are dealing with a small subset of the overall language. The interpreter responds to the same syntax error with the message:

You need to use TWO numbers in a WriteInt statement

288 Software Engineering in Higher Education

5.4 Encouraging Experimentation

The Edit-Compile-Link-Run cycle imposed by a compiler often leads to a very long period of elapsed time between the discovery of an error and the second run of a modified program in which it is corrected (particularly on older PCs). We found that a lot of valuable staff time was wasted watching students' programs compile and link.

Using the interpreter it was possible to encourage the students to *experiment* with the system. For example, the student might ask:

"What happens if I read in the value X twice?"

Using a compiler the lecturer is forced to answer:

"The second time the value is read in it will overwrite the first value, since you are using the same variable name."

It would be nice to be in a position to say:

"Try it and see what happens."

thus encouraging students to seek an answer to their own questions experimentally whilst improving their problem solving skills. Using the interpreter this *is* possible, and the student will get immediate feed-back. Using the compiler, the effort involved in editing, compiling, linking and running a simple test programs to answer questions such as these is prohibitive.

5.5 Console Logging

As mentioned in section 4.3, we provided facilities for the student to log and exact a replica of the information that appears on the screen during an interpreter session. This is convenient for submission of sample runs of programs as part of a coursework. The first coursework we set was to read in twelve values for rain fall readings and to produce a star plot together with maximum, minimum and average values. It was possible to produce a solution to this coursework on the interpreter, and we subsequently used it as a vehicle for motivating the use of procedures and the value of migration to the compiler. By this stage the students were in a suitable position to appreciate the benefits offered by the compiler.

6 Problems with the Interpreter Approach

In this paper we have extolled the virtues of the interpretive style in general and of our own interpreter in particular. We do believe that for mature students who are new to computing this approach does allow the lecturer to focus upon the essentials of programming techniques

Of course, there are some drawbacks to our approach, chiefly the frustration felt by the more experienced students at their inability to "play" with the more advanced features that they know the language has.

In responding, we made no apology to these more advanced students. Our course was, after all, a conversion course, assuming no prior knowledge of computing. In any case, we found it quite possible to set problems which *could* be solved using the subset of Modula-2 available on the interpreter, but which were sufficiently challenging to keep the more experienced students busy.

Software Engineering in Higher Education 289

There was also a problem when it came to students migrating to the compiler. Some students said that they would have preferred to use a single system from the outset, whilst others hankered for the simple, care-free days of life with the interpreter.

We also should admit to a few technical *disadvantages* of the interpretive style. If the student fails to close the quotation mark in a string, then the interpreter will interpret everything that follows as a part of the (unclosed) string. This leads to students wondering why the **run** command appears to have no effect, no matter how often they type it. Our remedy was to provide a prompt, so that one can easily see when the interpreter is ready to execute a command.

In addition, it is possible for the interpreter to run out of memory (largely due to our use of a rather crude garbage collection algorithm). Should this occur, the interpreter issues a warning, and gives students time to ensure that their disk is loaded so that the system can save their work and carry on as well as possible.

On the whole we felt that our introduction of an interpreter produced a higher proportion of proficient students and that less students appeared to give up in the early stages of the course. Sadly we still found that there were some students who simply did not “get it” when it came to programming, but there certainly were less students in this category.

Overall the system was used for a period of two years by approximately two hundred students. A free copy of the object code for the interpreter is available to other academic institutions. For details e-mail m.harman@uk.ac.unl or write to the address at the head of this paper.

7 Conclusion

We have found that using a compiler to teach mature students who are new to programming presents the students with such a steep learning curve that many lose confidence and simply “give up”.

This problem appears to arise solely because of the necessary complexity of compiler technology, which requires that students learn about editing and linking before being able to execute a simple program.

We advocate an interpretive approach where the students first exposure to the programming language is through the use of a simple interpreter. Once students are fluent in the basic constructs of the language they should then migrate to the use of a compiler.

The migration phase can be used to explain the motivation for many of the facilities offered by the compiler and the programming language.

We have implemented an interpreter for a subset of Modula-2 and used the approach we advocate on our introductory programming course delivered on the Post Graduate Diploma in Computing at the School of Computing, University of North London.

Acknowledgement

We gratefully acknowledge the help and advice of Kathy Newstead, Carol Kliem and Ray Jones.

290 Software Engineering in Higher Education

References

- [1] A. Aho, R. Sethi and J.D. Ullman.
Compilers, Principles, Techniques and Tools. Addison-Wesley, 1986.
- [2] R. Bornat.
Understanding and Writing Compilers, A Do-It-Yourself Guide. MacMillan, 1979.
- [3] K. Kliem and M. Harman.
An Introduction to Modula-2. Course notes, School of Computing, University of North London, 1992.
- [4] C. Sadler and S. Eisenbach.
Program Design Using Modula-2, Addison Wesley, 1989.
- [5] H. Schildt.
Modula-2 Made Easy. Osbourne McGraw-Hill, 1986.