A Programming Languages Course for Freshmen

J. Ángel Velázquez-Iturbide
Universidad Rey Juan Carlos
C/ Tulipán s/n, 28933 Móstoles,
Madrid, Spain
angel.velazquez@urjc.es

ABSTRACT
Programming languages are a part of the core of computer science. Courses on programming languages are typically offered to junior or senior students, and textbooks are based on this assumption. However, our computer science curriculum offers the programming languages course in the first year. This unusual situation led us to design it from an untypical approach. In this paper, we first analyze and classify proposals for the programming languages course into different pure and hybrid approaches. Then, we describe a course for freshmen based on four pure approaches, and justify the main choices made. Finally, we identify the software used for laboratories and outline our experience after teaching it for seven years.

Categories and Subject Descriptors
K.3.2 [**Computers and education**]: Computer and information science education – *computer science education.*

General Terms
Languages, Theory.

Keywords
Programming languages, formal grammars, language description, programming paradigms, functional programming, recursion.

1. INTRODUCTION
The topic of programming languages is a part of the core of computer science. It played a relevant role in all the curricula recommendations delivered by the ACM or the IEEE-CS since the first *Curriculum'68* [2]. Recent joint curricular recommendations of the ACM and the IEEE-CS identified several "areas" which structure the body of knowledge of the discipline. The list of areas has grown since the first proposal made by the Denning Report [4] up to 14 in the latest version, *Computing Curricula 2001* [12]. Programming languages has always been one of these areas.

Internationally reputed curricular recommendations are a valuable tool for the design of particular curricula. However, each country has specific features that constrain the way of organizing their studies. In Spain, the curriculum of a discipline offered by a university is the result of a trade-off. On the one hand, the university must at least offer a number of credits of the core subject matters established by the Government. On the other hand, the university may offer supplementary credits of the core as well as mandatory and optional courses defined according to the profile of the University and the faculty. Any proposal of a new curriculum follows a well-established process: (1) the curriculum is designed by a Center after consulting the departments involved; (2) it must be approved by the University Council; (3) the Universities Council of the Nation must deliver a (positive) report; and (4) the curriculum is published in the Official Bulletin of the Nation. This scheme has a number of advantages, e.g. a minimum degree of coherence among all the universities is guaranteed. However, it also has a number of disadvantages, e.g. the process to change a curriculum is very rigid.

The Universidad Rey Juan Carlos is a young university, now seven years old. It offered studies of computer science since the very first year. The curriculum was designed by an external committee, so the teachers of computer science thereafter hired by the university did not have the opportunity to elaborate on it. The curriculum had a few weak points that would recommend a light reform, but the priorities of the new university postponed it.

The curriculum establishes the features of the "Foundations of programming languages" course. The course is scheduled to last for fifteen weeks, with three lecture hours per week and two supervised laboratory hours per week. However, some flexibility is allowed, so that some weeks may be released from the lab component.

This course is both a strong and a weak feature of the curriculum. It is a strong feature because programming languages are marginal in the official core. Consequently, our curriculum is closer to international recommendations than most Spanish universities. However, it is a weak feature, because the course is offered in the second semester of the first year! Notice that the programming languages course is more typically offered as an intermediate or advanced course in the third or fourth year.

Our problem was how to teach the programming languages course to freshmen. The paper presents our design of the course and our experience. In the second section we first analyze and classify proposals for the programming languages course into different pure and hybrid approaches. In section 3, we describe a course for freshmen based on four pure approaches, and justify the choices made with respect to the factors that most influenced its design. Finally, we identify the software used for laboratories and outline our experience after teaching it for seven years.

ITiCSE'05, June 27–29, 2005, Monte de Caparica, Portugal.

2. APPROACHES TO TEACHING PROGRAMMING LANGUAGES

Since *Curriculum'68*, several issues on programming languages have received attention in the different curriculum recommendations: particular programming languages, language implementation, etc. It is formative to study (or to browse, at least) such recommendations, even though the large number of topics can be discouraging for the teacher.

In this section, we try to organize different contributions. Firstly, we identify "pure" approaches to the programming languages course. Secondly, we describe their implementation, usually as hybrid approaches. Finally, we briefly discuss the issue of which programming languages and paradigms to use for the course.

2.1 Pure Approaches

Probably, the best study on approaches to the programming languages course was given by King [7]. He made a study of 15 textbooks and found 10 different goals. Furthermore, he identified 3 approaches on which these textbooks were based and discussed a number of issues. We have extended his classification up to 5 approaches. Although most books and courses follow a hybrid approach, it is clarifying to distinguish the following pure ones:

- Catalogue approach. It provides a survey of several programming languages. This approach has several advantages: the student acquires a good education in several languages, it allows studying the interaction among different constructs of a language, and the languages may be studied in chronological order. However, it also exhibits disadvantages: there is much redundancy in studying similar features in different languages, and there is no guarantee that the student acquires a solid education.
- Descriptive (or anatomic [5]) approach. Programming languages have many common elements which can be grouped into categories and studied independently. Typical examples are: data types, hiding mechanisms, execution control, etc. The advantages and disadvantages of this approach are roughly the opposite of the previous one.
- Paradigm approach. Although each language has different characteristics and constructs, it is based on a basic model of computation called programming paradigm. The paradigm approach is an evolution of the descriptive approach described above since it generalizes language constructs and groups them consistently. Typical examples of paradigms are functional, logic and imperative programming.
- Formal approach. It studies the foundations of programming languages, mainly their syntax and semantics. The main advantage of this approach is that the student acquires a solid conceptual background. However, it has the risk of being too formal and therefore keeping far from the study of the programming languages themselves.
- Implementation approach. It comprises language processing topics. This approach is usually adopted jointly with the descriptive one, so that the run-time mechanisms that support each language construct are also described. This allows estimating the computational cost of each construct. However, the student may associate each concept with a particular implementation; this approach may also be in contradiction with the idea that a high level language should be understandable independently from its implementation.

2.2 Implementation of Pure Approaches

The formal and implementation approaches are the basis of two well known and established courses: computation theory and language processors. They are not studied in this paper as standalone courses, but we consider their integration into the programming languages course.

Despite these "pure" approaches, it is more common to adopt a hybrid one, formed by a combination of several approaches. For instance, we have explained that a descriptive course may address the implementation of language constructs. The descriptive and paradigm approaches also are commonly complemented by a small catalogue of selected languages. It is also common to find a descriptive part based on the imperative paradigm, followed by a second part based on the catalogue or the paradigm approaches. Finally, the formal approach can complement the descriptive or implementation ones.

From a historical point of view, the catalogue approach was the most common in the first years of computing curricula. However, it has almost universally been abandoned, with some interesting exceptions, such as the experience by Maurer [9] on a subfamily of four C-like languages.

The trend has been towards giving more importance to the foundations of programming languages, mainly elements and paradigms. After this evolution, it seems that the two most common organizations are:

- Descriptive approach, complemented with some languages or paradigms. It is the most common organization, according to currently available textbooks.
- Descriptive approach, illustrated by means of interpreters of some selected languages. Interpreters and paradigms can be combined in two symmetrical ways: either implementing an interpreter in each paradigm [14], or implementing an interpreter for one language of each paradigm; the latter approach can be adopted with an imperative language [6] or, more probably, with a functional language [1].

There are few proposals for a holistic approach. One exception is Wick and Stevenson's proposal [17], which combines the descriptive, formal, paradigm and implementation approaches into a "reductionistic" one.

2.3 Choice of Languages and Paradigms

Courses on programming languages do not simply consist in the study of one or several languages, but their study usually is a part of the course. The selection of these languages rises some questions, that we cannot discuss here.

A related issue is the selection of programming paradigms. Not all the paradigms can be equally useful for a course on programming languages for freshmen. Firstly, some paradigms are richer to illustrate language elements than others. Secondly, some paradigms can be more adequate to freshmen than others. There is not a catalogue of paradigms classified by suitable academic year, but it is important to use a more objective criterion than just the personal opinion of faculty.

We have used the *Computing Curricula 2001* [12] as an objective basis to identify feasible paradigms. They identify the paradigms that have succeeded in CS1: procedural, object-oriented, functional, algorithmic notations, and low-level languages. The last two choices are useful for CS1 but not for a programming languages course, thus the remaining choices are procedural, object-oriented, and functional. An additional choice, not cited by *Computing Curricula 2001*, consists in the use of tiny languages [8][10]. These languages can be *ad hoc* designed for a specific domain or embedded into operating systems or applications.

3. OUR PROPOSAL

We discarded the catalogue approach because it would only contribute to a memorization effort by freshmen. Consequently, our course is based on the remaining four approaches:

- Formal approach. Formal grammars and syntax notations are given in depth. Language semantics is simply introduced.
- Implementation approach. Only basic concepts are given.
- Descriptive approach. Basic language elements are reviewed.
- Paradigm approach. A programming paradigm is given in depth (functional programming) but others are just sketched.

Table 1 contains the contents of the course we offer.

There are several major factors to consider for the design of a programming languages course. Firstly, the course when it is offered to students determines to a large extent the knowledge and maturity of students. Secondly, the existence of related courses, such as courses on automata theory or programming methodology, may recommend removing some overlapping topics. Thirdly, the specialization profile of faculty can foster the choice of a given topic instead of another one. Fourthly, a course with so many different topics must guarantee coherence among them. Finally, time constraints typically limit the number of topics to consider.

In the following subsections we justify the adequacy of the choices made with respect to these factors.

Table 1. Syllabus of our programming languages course

PART I. INTRODUCTION

Chapter 1. General issues
Computer and programming languages. Elements, properties and history of programming languages. Classifications.

PART II. SYNTACTIC FOUNDATIONS OF PROGRAMMING LANGUAGES

Chapter 2. Grammars and formal languages
Alphabets, symbols and chains. Languages. Grammars. Derivation of sentences. Recursive grammars. Classification of grammars: Chomsky's hierarchy. Abstract machines.

Chapter 3. Regular grammars
Definition. Uses and limitations. Regular expressions and regular languages. Finite-state automata.

Chapter 4. Context-free grammars
Definition. Uses and limitations. Parsing trees. The ambiguity problem and its removal.

PART III. DESCRIPTION AND PROCESSING OF PROGRAMMING LANGUAGES

Chapter 5. Language processors
Abstract (or virtual) machines. Classes of processors. Stages in language processing. Concrete vs. abstract syntax.

Chapter 6. Lexical and syntactic notations
Lexical and syntactic elements. Regular definitions. Syntax notations: BNF, EBNF, syntax charts.

Chapter 7. Semantics
Semantics. Classes of semantics. Static semantics. Binding.

PART IV. THE FUNCTIONAL PARADIGM

Chapter 8. Basic elements
Function definition and application. Programs and expressions. Overview of functional languages. Built-in types, values and operations. The conditional expression.

Chapter 9. Advanced elements
Operational semantics: term rewriting. Recursive functions. Local definitions.

Chapter 10. Functional data types
Constructors. Equations and patterns. Pattern matching.

PART V. ELEMENTS OF PROGRAMMING LANGUAGES

Chapter 11. Lexical and syntactical elements
Identifiers. Numbers. Characters. Comments. Delimeters. Notations for expressions. Program structure and blocks.

Chapter 12. Data types
Recursive types. Parametric types: polymorphism. Polymorphic functions. Type systems. Type checking and inference. Type equivalence. Type conversion. Overloading.

PART VI. PROGRAMMING PARADIGMS

Chapter 13. Other paradigms and languages
Imperative paradigms. Logic paradigm. Motivation of concurrency. Other computer languages: mark-up languages.

3.1 Maturity of Students

One major concern was the fact that the course is offered to freshmen. A single approach could not be used because of the freshmen's lack of knowledge of programming languages. A variety of contents from the different approaches must be selected in order to give them a comprehensible and rich view of programming languages.

The lack of maturity and capability of students to understand certain topics was a bottleneck for course organization. The different topics can be given with varying degrees of depth, but always making sure that freshmen can master them. We found that some topics are especially difficult to understand, even formulated in the simplest way. This mainly applies to:

- Semantics of programming languages.
- Implementation of programming languages.
- Some programming paradigms, such as concurrency.

Consequently, these topics were included in a summarized way, so that students could achieve a global view of them and

understand the main issues involved. The rest of the topics could potentially be taught more deeply, but without forgetting that they were offered to freshmen.

In terms of Bloom's taxonomy [2], the three topics above can be mastered at the knowledge level, or even comprehension level. However, for the rest of topics, we can expect students to achieve the application and analysis levels, at least.

3.2 Overlapping with Other Courses

Some topics are also offered in other courses, either in the same or in a subsequent year. Consequently, these topics can be removed or dealt with more shallowly. The most probable conflicts are:

- Imperative programming, either procedural or object-oriented.
- Grammars and formal languages.
- Language processors.

In our case, there is an annual course on programming methodology based on the imperative paradigm, but the other two topics are not included in the curriculum. The programming methodology course is offered in the first academic year.

Consequently, we removed the imperative paradigm, except for its use in some illustrating examples, mainly in part V. However, we kept chapters on grammars and formal languages, and on language processors.

3.3 Preferences and Specialization of Faculty

This factor is important in order to choose among equally eligible options, or to give broader coverage of some topics. In particular, faculty can be more familiar with some paradigms than with others. This was over-riding for our choice of the programming paradigm.

In subsection 2.3 we discussed suitable paradigms for freshmen, and we concluded that *Computing Curricula 2001* fosters the selection of the procedural, object-oriented, and functional paradigms. We discarded the procedural paradigm as it is concurrently taught in the programming methodology courses. Finally, our specialization gave priority to the functional paradigm over object-orientation. The reader can find many experiences in the literature, but we recommend a monograph on functional programming in education [13].

Functional programming is a paradigm with several advantages, such as short definition of languages, simple and concise programs, high level of abstraction, etc. However, its main advantage for us is richness of elements. This allows us to deal with many aspects of programming languages (e.g. data types, recursion, polymorphism, etc.) in a natural and easy way.

The use of tiny languages is another attractive choice in a course for freshmen. However, we also discarded them in favor of the functional paradigm because they have fewer language elements.

3.4 Coherence and Unifying Themes

A key issue in a course based on several approaches is to provide contents coherence. A network of relationships among the different parts makes possible their coherent integration.

Part III (description and processing of languages) is the pragmatic continuation of part II (formal grammars). Thus, EBNF and syntactic charts are introduced in part III as more adequate notations for language description than pure grammar definitions. Language processing is given at a conceptual level, but the role of regular and syntax-free grammars in the architecture of language processors is highlighted.

Parts IV and V are both based on a functional language, which is described with the tools given in part III, mainly EBNF and type constraints.

Parts IV and V are also related because they are based on the same language. In order to provide more homogeneity, language elements studied in part V are introduced in a universal way, but they are mainly illustrated with the functional paradigm.

Last, but not least, recursion is adopted as a recurring theme during the course. In effect, it is found in grammars, functions and data types. The recurrent presentation of this topic fosters deeper understanding by students.

The "pure" definition of recursion is given early in the course, but its three instantiations enumerated above are studied later. For each instantiation, the mechanisms that accompany a recursive definition are clearly identified, in particular representation of information and operational semantics [16]. For instance, recursive grammars represent sentences as strings of terminal symbols, and its operating semantics is defined in terms of derivation of sentences. However, recursive functions represent information as expressions, and its operating semantics is defined in terms of term rewriting.

3.5 Time Constraints

As a final factor, time constraints limit the depth of study of those topics that could be studied longer. A global view of the course schedule is given in Table 2.

Table 2 Schedule of the course

Part	Theory #hours	Lab #hours
Part I	5	–
Part II	14	8
Part III	10	–
Part IV	12	6
Part V	8	6
Part VI	4	2

In part II (formal grammars), regular and context-free grammars are the only ones studied in depth because of their importance for language description and processing.

Moreover, only one paradigm can be studied in depth. Even so, the lack of time limits the presentation of functional programming (parts IV and V) to the core elements of the paradigm. Other elements, important for the functional programmer, can not be addressed (e.g. higher-order, lazy evaluation, or currification). However, this is not a serious drawback since the aim of including functional programming in the course is teaching the essentials of a new paradigm as well as illustrating language elements.

4. LABORATORY COURSEWARE

A course on programming languages must have a laboratory component. The laboratory schedule includes sessions for those parts of the course where problem solving can be faced, mainly formal grammars and functional programming. Laboratory tools were selected carefully so that they are adequate for freshmen to exercise non-trivial concepts; simple user interaction and visualization facilities are of great help here. There are a number of tools that fulfill these requirements. For formal grammars, we require simulators that allow at least manipulating regular expressions, finite automata, context-free grammars and derivation trees. Our final selection was JFLAP [11]. For functional programming, we require a programming environment that shows term rewriting as the operational semantics. Our final selection was WinHIPE [15].

5. EXPERIENCE

We have been teaching this course for seven years. Although the basic structure has roughly been constant, it was refined according to our experience. In particular, the emphasis on recursion was introduced after several years as we noticed student problems with this concept. We consider that we have succeeded, at least in eliminating the magical connotation of recursion.

A major change was the relative order of chapters on formal grammars and the functional paradigm. During the first year, they were given in reverse order. However, students had problems in understanding the syntax of functional declarations that led us to teach in the first place formal grammars (and therefore syntax notations such as EBNF). Thus, a foundation to declare syntax was laid and then used to introduce functional programming.

The literature classifies the main difficulties for teaching functional programming into syntactical, conceptual and "psychological" problems [13]. In our approach, the two former kinds of problems are avoided, but the latter remains. As freshmen learn concurrently the functional and one imperative language, they get the idea that functional is an exotic, useless paradigm.

6. CONCLUSION

We have described a course on programming languages for freshmen. It comprises elements from four different approaches. We have described the contents of the course, and we have explained the factors that led us to its current design. The experience has been very positive both for teachers and for students. As the Denning report sought for CS1, we consider that our course illustrates that it is feasible to offer some traditionally intermediate or advanced matters in introductory courses.

7. ACKNOWLEDGMENTS

This work is supported by the research project TIN2004-07568 of the Spanish Ministry of Education and Science.

8. REFERENCES

[1] Abelson, H., and Sussman, G.J. *Structure and Interpretation of Computer Programs*. MIT Press, 2ª ed., 1996.

[2] Bloom, B., Furst, E., Hill, W., and Krathwohl, D.R. *Taxonomy of Educational Objectives: Handbook I, The Cognitive Domain*. Addison-Wesley, 1959.

[3] Curriculum Committee on Computer Science. Curriculum '68: Recommendations for academic programs in computer science. *Comm. ACM, 11*, 3 (March 1968), 151-197.

[4] Denning, P. *et al. Computing as a Discipline*. ACM Press, New York, 1988.

[5] Fischer, A.E., and Grodzinsky, F.S. *The Anatomy of Programming Languages*. Prentice-Hall, 1993.

[6] Kamin, S.N. *Programming Languages: An Interpreter-Based Approach*. Addison-Wesley, 1990.

[7] King, K.N. The evolution of the programming languages course. In *23rd SIGCSE Technical Symposium on Computer Science Education (SIGCSE'92)*. ACM Press, New York, 1992, 213-219.

[8] Kolesar, M.V., and Allan, V.H. Teaching computer science concepts and problem solving with a spreadsheet. In *26th SIGCSE Technical Symposium on Computer Science Education (SIGCSE'95)*. ACM Press, New York, 1995, 10-13.

[9] Maurer, W.D. The comparative programming languages course: A new chain of development. In *33rd SIGCSE Technical Symposium on Computer Science Education (SIGCSE 2002)*. ACM Press, New York, 2002, 336-340.

[10] Popyack, J.L., and Herrmann, N. Why everyone should know how to program a computer. In *IFIP World Conference on Computers in Education VI (WCCE'95)*. Chapman & Hall, 1995, 603-612.

[11] Hung, T., and Rodger, S.H. Increasing visualization and interaction in the automata theory course. In *31st SIGCSE Technical Symposium on Computer Science Education (SIGCSE 2000)*. ACM Press, New York, 2000, 6-10.

[12] The Joint Task Force on Computing Curricula IEEE-CS/ACM: Computing Curricula 2001– Computer Science, http://www.computer.org/education/cc2001/final, 2001.

[13] Thomson, S., and Wadler, P. (eds.) Functional programming in education. *Journal of Functional Programming, 3*, 1 (1993).

[14] Tucker, A.B., and Noonan, R.E. Integrating formal models into the programming languages course. In *33rd SIGCSE Technical Symposium on Computer Science Education (SIGCSE 2002)*. ACM Press, New York, 2002, 346-350.

[15] Velázquez-Iturbide, J.Á. Improving functional programming environments for education. In M. D. Brouwer-Hanse y T. Harrington (eds.), *Man-Machine Communication for Educational Systems Design*. Springer-Verlag, NATO ASI Series F 124, 1994, 325-332.

[16] Velázquez-Iturbide, J.Á. Recursion in gradual steps (is recursion really that difficult?). In *31st SIGCSE Technical Symposium on Computer Science Education (SIGCSE 2000)*. ACM Press, New York, 2000, 310-314.

[17] Wick, M.R., and Stevenson, D.E. A reductionistic approach to a course on programming languages. In *32nd SIGCSE Technical Symposium on Computer Science Education (SIGCSE 2001)*. ACM Press, New York, 2001, 253-257.